Generic Compression and Recall of Signals with Application to Dolphin Whistles

by

Kevin G. Christian

B.S.E.E., University of Puerto Rico - Mayagüez Campus (1983)
S.M., Massachusetts Institute of Technology (1985)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1993

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
NOV 30 1993
LIBRARIES
ARCHIVES

© Massachusetts Institute of Technology 1993. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
June 3, 1993

Certified by ....
David H. Staelin, Professor of Electrical Engineering
Thesis Supervisor

Accepted by ....
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

Generic Compression and Recall of Signals with Application to Dolphin Whistles

by

Kevin G. Christian

Submitted to the Department of Electrical Engineering and Computer Science
on June 4, 1993, in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in
Electrical Engineering and Computer Science

Abstract

The efficient compression of a lengthy signal in order to allow quick detection of repeated elements, even though the repetitions are separated by large amounts of time, is investigated using a dolphin whistle database. Work is done in three levels. Level I does silence removal as well as conditioning of the data for further analysis. Typical compression for this level is about a factor of 12. In level II, data is further compressed by converting it into a single frequency-vs.-time trace. Assuming there is no desire to reverse the level II compression, the compression factor is typically increased an additional factor of 257 relative to level I.

The final level, III, involves irreversible compression and by itself achieves a compression factor of about 1. Reconstruction of the original signal is not possible using Level III output, although detection of repetitions is possible. In this level, each of the 1169 whistles in the database is converted into a single point in a 16-D coding space. Techniques for the creation of adequate coding spaces are presented. To find a whistle in the database, the coding space needs to be searched. Two methods, the $k-d$ tree and the expanding bucket, are implemented and evaluated for the search problem. Although the $k-d$ tree requires that a larger fraction of the database be searched to find a match, it is faster than the expanding bucket. However, the best speeds for both methods (obtained with a 6-D coding space) are comparable (35 ms to 53 ms on a SparcStation 2). Under certain budget constraints, the expanding bucket can give superior performance.

Using the coding space of level III, the maximum potential whistle vocabulary the space can hold was estimated. Single animals reproduce their own signature whistles precisely, and the associated "single" potential vocabulary was found to be over one billion unique whistles. When the possibility of copying "errors" are introduced, the "shared" potential vocabulary was just a few hundred whistles; based on very limited copying data.

As mentioned above, the particular database used in this study is composed of dolphin whistles. The techniques are generic in nature, however, facilitating their application to other fields. For example, manufacturing problems such as fault detection, vibration analysis, and diagnosis of machinery can benefit from this work. All that is required to apply the search algorithms discussed in this thesis is the creation of an appropriate coding space for the signals of interest.

Thesis Supervisor: David H. Staelin
Title: Professor of Electrical Engineering

Acknowledgments

First of all, I would like to thank Professor David Staelin for his ideas and support throughout these past few years. His guidance has been invaluable. I am also grateful to my friends in the Radio Astronomy group, both past (Bernie Szabo, Shiufun Cheung, Perry Bonanni, and Ashraf Alkhairy) and present (Phil Rosenkranz, Jack Barrett, Carlos Cabrera, and Michael Schwartz) for our numerous discussions. Special thanks to Carlos for his help in digitizing data for my thesis and investigating various ways of obtaining a coding space for dolphin whistles.

I am also grateful for the help I obtained from the people at the Woods Hole Oceanographic Institution. They always managed to make me feel as one of the group. In alphabetical order they are: Mary Ann Daher, Kurt Fristrup, Terrance Howald, Cheri Recchia, Laela Sayigh, Peter Tyack, and Bill Watkins. Thanks also for the use of the computer resources. I guess I will not be needing any more data guys.

Financial support is something I could not have done without. For that I would like to thank Professors Jeff Lang, William Siebert, James Roberge and Campbell Searle for allowing me to be a teaching assistant in their courses at different times. The Woods Hole Oceanographic Institution, the Leaders for Manufacturing Program, Graduate School at MIT, and the Department of Electrical Engineering at MIT have also provided much needed financial support.

Thanks also to my thesis committee, Professors David Staelin, Peter Elias and Peter Tyack. This thesis is all the more readable thanks to them.

Last but not least, thanks to my wife Teresa. The real power behind the throne and the one that held everything together. Thanks for keeping up the faith, now it is my chance to show you it was all worth it. And thanks to my son Brian, who may not care a whole lot about all this but who's mere presence made me care about it even more.

Contents

1 Introduction **1**

1.1 Prior Related Work . . . 2

1.2 Problem Statement . . . 6

1.3 Proposed Approach . . . 10

1.3.1 Level I . . . 11

1.3.2 Level II . . . 12

1.3.3 Level III . . . 12

1.4 Thesis Organization . . . 13

2 Level I Compression - Reversible **14**

2.1 Signal Detection . . . 15

2.1.1 Background on Signal Detectors . . . 16

2.1.2 Duration of Analysis Window . . . 18

2.1.3 Development of Signal Detector . . . 19

2.1.4 Improving the Signal Detector . . . 30

2.1.5 Endpoint Detector . . . 33

2.2 Frequency Domain Representation . . . 39

2.3 Whistle Frequency Resolution . . . 46

2.4 Additional Ideas . . . 48

2.5 Reversing Level I Compression . . . 51

3 Level II Compression **53**

3.1 Peak Detection 53

3.2 Estimating the Frequency-vs.-Time Trace 60

3.3 Reversing Level II Compression 67

3.4 Improved Endpoint Detection 70

3.4.1 Discriminant Analysis and Clustering 71

3.4.2 Cepstral Analysis 72

4 Level III Compression - Coding Space **80**

4.1 Candidate Dimensions 81

4.2 Evaluation and Selection of Coding Spaces 87

4.3 Animal Potential Vocabulary 100

4.4 Comparing Cluster Sets 106

5 Level III Compression - Detecting Repetitions **120**

5.1 General Background 121

5.1.1 Full Search Methods 123

5.1.2 Adaptive Buckets 127

5.1.3 Hash Functions 128

5.1.4 Metric Based Methods 129

5.1.5 Trees 136

5.2 $K - D$ Tree Implementation 139

5.3 Expanding Bucket Implementation 145

5.4 Experimental Results 148

6 Conclusions **172**

6.1 Summary and Conclusions 172

6.2 Future Work 176

A Source Code - Declarations **178**

B Source Code - Tracing Algorithm **183**

C Source Code - Search Algorithms **196**

Bibliography **215**

List of Figures

2-1 Histograms of signal detection measures. 24

2-2 Histograms of signal detection measures - cont. 25

2-3 Histograms of signal detection measures - cont. 26

2-4 Histogram of linear combination measure. 28

2-5 Example of zero-crossings. 31

2-6 Histogram of adaptive quant. zero-crossing count measure. 34

2-7 Endpoint detection algorithm. 36

2-8 Endpoint detection algorithm - cont. 37

2-9 Periodogram power density spectrum estimate. 41

2-10 Burg's power density spectrum estimate. 43

2-11 Sample spectrogram of dolphin whistles. 47

3-1 Example of power density spectrum for signal window. 54

3-2 Example of background power density spectrum. 55

3-3 Power density spectrum for signal window after removing background. 56

3-4 Signal segment spectrogram. 56

3-5 Example of peak selection algorithm output. 57

3-6 Example of peak selection algorithm output after cleanup. 58

3-7 Example of grid used in finding frequency vs. time trace. 63

3-8 Tracing algorithm example. 66

3-9 Tracing algorithm example (1) using real data. 68

3-10 Tracing algorithm example (2) using real data. 68

3-11 Sample spectra and cepstra for FB 35. 74

3-12 Sample spectra and cepstra for FB 62. 75

3-13 Sample spectra and cepstra for FB 153. 76

3-14 Power density spectrum from filtered cepstrum. 79

4-1 Block diagram for generating a large number of measures. 81

4-2 MSNR vs. OVLPC scatter plot. 96

4-3 AVED2 vs. AVER2 scatter plot. 97

4-4 Log(potential vocabulary) versus dimensionality plot. 103

4-5 Log("shared" potential vocabulary) versus dimensionality plot. 105

4-6 Cases when matching cluster A_1. 108

4-7 Matching clusters - example 1. 111

4-8 Matching clusters - example 2. 112

5-1 Example used to illustrate metric-based method. 129

5-2 Tree representation of a coding space partitioning. 137

5-3 Expanding bucket along a dimension (example). 146

5-4 Number of terminal nodes used by each file in the database. 150

5-5 Histogram of number of records needed to encounter nearest neighbour. 154

5-6 Histogram of number of records needed to confirm nearest neighbour. 155

5-7 Histogram of number of computations performed during search. . . . 156

5-8 Average search time as a function of dimensionality. 159

5-9 $K - D$ tree performance as a function of database size. 161

5-10 Expanding bucket performance as a function of database size. 162

5-11 Average rank of nearest neighbour versus search budget. 164

5-12 $K - D$ tree sensitivity to database organization. 165

5-13 Distance as a function of rank. 166

5-14 Distance validation - example 1. 168

5-15 Distance validation - example 2. 169

5-16 Distance validation - example 3. 170

5-17 Distance as a function of rank for 3 random whistles. 171

List of Tables

2.1 Signal detection measures. 20

2.2 Performance of signal detection measures. 23

2.3 Performance of endpoint detector. 38

2.4 Variability of repeated whistle frequencies. 48

2.5 Coding gain achieved by quantizing difference signal. 49

2.6 SNR obtained by different quantizers. 50

3.1 Cepstral peak amplitudes. 77

4.1 List of candidate dimensions (measures) for the coding space. 85

4.2 Medium database contents. 89

4.3 List of selection statistics for evaluation of coding spaces. 93

4.4 List of candidate dimensions (measures) for the coding space. 95

4.5 Correlation coefficients for selection statistics. 96

4.6 Absolute value of correlation coefficients for some dimensions. 98

4.7 Coding space dimensions (measures). 99

4.8 Cluster volume and potential vocabulary as function of dimensionality. 102

4.9 Whistle list for manual clustering. 115

4.10 Cluster set statistics per individual subject. 116

4.11 Comparison of manual cluster assignments. 117

4.12 Manual clustering for animal identification. 119

5.1 Steps in metric based method example. 131
5.2 List of dimensions used in $k - d$ tree. 145
5.3 Contents of large database (1169 records, 22 animals). 149
5.4 Average performance of search methods. 157
5.5 Performance of search algorithms as dimensionality changes. 158
5.6 Records needed for confirmation as a function of database size. 160

Chapter 1

Introduction

This thesis deals with the analysis of large data sets with the explicit purpose of being able to "detect repetitions" in these data sets. Repetitions are detected by searching the database for the nearest neighbors to the record being studied. The overall goal is to develop a generic compression system coupled with efficient storage and retrieval techniques to help in the analysis of large sets of data. These two problems, compression and storage/retrieval, can not be totally separated if efficiency and performance are to be maximized.

Other kinds of analyses, for example, finding "classes" in the data based on element attributes are beyond the scope of this thesis. Analyses of that kind have been done elsewhere (see, for example, [8] which describes a program called AUTOCLASS combining Bayesian analysis and artificial intelligence.). This work, will concentrate on the "detection of repetitions". The resulting system will be able to find repeated elements in a data set even when these repetitions are separated by a large span of time.

The system being proposed has clear applications in many fields. In manufacturing, the system can be used as a diagnostic or quality control tool. For example, the elements in the database can be engine parameters for different engine conditions (e.g. faulty spark plug, vacuum leak, etc.). By comparing the parameters of the

engine to be diagnosed against those in the database a quick diagnosis is possible. In marine biology the system can be used for identification and characterization of animal sounds. For dolphin whistles there is interest in detecting whistle repetition to aid in animal identification as well as to help answer questions such as: How unique are the signature whistles of a particular dolphin? Do these signature whistles evolve (change) during the life of the animal?

Given the availability of a large collection of dolphin whistles for study, a dolphin database has been selected as a test case for this work. Most of the signals used in this study come from a population of wild dolphins. When collecting whistles, the dolphin is captured in a net and then a contact microphone attached to make the recording. Note that for these signals there is no ambiguity regarding the identity of the animal making the whistle. Other reasons for using the dolphin whistles are their relative simplicity and high number of repetitions. However, as will be discussed later, the animal does not always repeat the same whistle all the time. Other kinds of whistles, as well as reproduction of other sounds in the animal's environment, are possible.

The rest of this chapter is organized as follows. The next section presents a discussion of prior related work. Following that, there is a section describing the problem to be addressed in this thesis and another section describing the approach taken in solving the problem. Finally, the last section describes how the thesis is organized.

1.1 Prior Related Work

The automatic speech recognition problem [30] is similar to the problem addressed in this thesis. In theory, speech recognition can be simply considered to be a pattern matching problem. The goal of a speech recognition system is to match the spoken utterance with those stored in the dictionary of the system. If we consider the dic-

tionary to be a database of utterances, then all the speech recognition system does is detect repetitions. By finding the repetition, the utterance is recognized.

Work in speech recognition differs in these important respects, however: 1) speech involves a "training" phase where words are defined, 2) speech is harmonically much more complex than are dolphin whistles, 3) words can be pronounced in a wide variety of ways with identical meaning, 4) for these reasons most speech recognition systems involve considerable ad hoc decision making (e.g. the number of states in a hidden Markov model (HMM)) and training. The work here is restricted to minimal ad hoc rule making and training, and potentially very large "vocabularies". Also, many speech recognition systems can interact with the speaker while, in this work, little cooperation can be expected from the "speaker" (dolphin).

Let us pause for a second and consider the size of the human speech recognition task described above. Being very ambitious, assume that the basic unit of comparison will be isolated whole words. (Clearly a wasteful approach since some sounds are common to many words). This unit of comparison will have an average duration of about 0.25 s. Now, if speech is sampled at 6 kHz (a 3-kHz bandwidth) using 8 bits (256 possibilities) to represent each sample (1500 samples per word), then there are theoretically 256^{1500} possibilities to be stored and searched. Compression is clearly necessary.

There are several ways to compress such a dictionary to make the recognition problem more manageable. First, limit the vocabulary, thus limiting the possibilities to be searched. Second, specialize the dictionary to fit a particular speaker. Out of all possibilities simply store those which can be produced by this particular speaker. Third, reduce the length of your basic comparison unit to fractions of a word. This effectively reduces the number of possibilities to be searched per fragment at the expense of complicating the matching procedure. Several matches are now needed before the word can be identified. Fourth, code the data differently. Instead of using raw samples to represent the data, other parameters such as power spectrum values,

cepstral coefficients, or linear-prediction coefficients can be used. In addition, these parameters may be coarsely quantized, discarding additional information. All the above methods can be combined to maximize their simplification effects.

Although the speech recognition problem has not yet been entirely solved, the field is rich with techniques proposed as solutions to the problem. Among the most popular are dynamic time warping (DTW) [15], hidden Markov models (HMMs) [23, 31], and neural nets [24]. The basic idea in DTW is to compute the "distance" between the utterance to be identified and a set of templates. In this approach, the distance between the utterance and the template represents an intrinsic quality-of-match measure. The template that best matches the utterance is selected. As the name implies, DTW adjusts (warps) the time base in order to minimize the distance between utterance and template.

HMM recognition is based on conditional probabilities. Instead of having templates for each utterance to be recognized, there will be a HMM. Given the utterance, either a word or word fragment (phoneme), the system computes the probability that the utterance was produced by a particular HMM. The model with the highest probability is selected. In order for the system to work, it must first be trained. During training each HMM is taught to recognize a specific utterance. (Basically, the parameters of the model are set so that they maximize the probability of producing a particular utterance). HMMs have a major advantage over DTW, namely, the number of computations required during the recognition phase is much less (at the expense of a very intensive training phase).

Neural networks have for some time been recognized as very good pattern classification units. It is this property which is utilized when they are used for speech recognition. However, pattern classification is not their only use. Neural nets have also been used to compute distances in DTW and HMM, thus creating hybrid speech recognition systems. As is the case with HMM, neural networks require a training period before they can be used effectively. This training represents a serious problem

in this work since this work deals with an unknown vocabulary that may continually evolve. Therefore, the system can not be "trained" in the usual speech recognition sense.

Many of the issues affecting speech recognition have parallels in this work. One such issue is the detection of the voice/silence boundary in order to make sure that the recognizer is operating on speech and not background noise. (This detection problem should not be confused with the detection of word and/or phonetic boundaries which has no direct parallel in this work). As part of this work, a "dolphin detector" has been devised to identify the silence/whistle boundary in the signal and help automate the system.

Recently, attempts have been made to improve the accuracy of speech recognition systems by introducing syntactic rules [35]. Such ideas have been extended to non-speech signals [10]. However, the amount of work involved in the development of a grammar for the signal under study clearly puts this approach beyond the scope of this work.

The speech recognition work discussed above is mainly geared towards the "detection of repetitions" aspect of this work. However, detecting repetitions is only part of the work to be done; the other part of the work involves compression and storage[1]. Let us now consider some of the prior work related to the compression and storage problem.

Related to the compression work are the fields of speech and image compression. The work in these fields can be divided in 2 areas, reversible and partially reversible compression. In reversible compression, the goal is to compress a signal (speech or image) for transmission over a (possibly) noisy channel[2] with the intention of perfectly reconstructing the original signal after transmission. In partially reversible compression the reconstruction only attempts to find a signal which is perceptually

[1] In the rest of this chapter, the term storage will include both storage and retrieval problems.

[2] Note that in the reversible case, the database can be considered a very simple noiseless channel.

similar to the original- that is, a signal that conveys the same message in the case of speech or that looks the same in the case of an image. Partially reversible compression is more applicable to this work.

One major distinction between the work done in speech and image compression and this work is that many of the techniques used in these fields fail to address issues related to transmission, leaving channel optimization as a separate task. Since the "channel" in this work is the cumulative database, channel optimization should not be left as an independent task. The techniques of speech and image compression must be combined with highly efficient methods for database storage and contents-addressable retrieval for the final system to be as good as possible.

Several (reversible) coding methods for compression have been developed over the years in information theory. Among the most popular methods are Huffman [14], arithmetic [40], and Ziv-Lempel [42] coding. Methods differ from one another in speed, memory requirements, how close to the entropy of the source the resulting code is, type of code (fixed vs. variable length), and type of source model (memoryless, adaptive, etc.), to name a few issues.

Regarding storage, one source of information is the study of the internal structure of database management systems (DBMS) [6]. This area of DBMS is concerned with how the elements of the database are stored in and retrieved from memory. Several techniques are used for storage/retrieval (accessing) of database records, the most common being hashing [27] and B-trees [5]. A number of search techniques were considered for this work and the best alternatives implemented.

1.2 Problem Statement

Broadly speaking this thesis deals with efficient compression of signals in order to detect repetitions quickly even when separated by long periods of time. The main goal is the development of a system to perform this compression and detection. In

addition, no prior training should be required in order to accomplish detection, and the system should create, and continually update, a database of "observed" signals as the processing continues. This is a complex and ambitious task dealing with two interrelated issues. The first is the selection of an appropriate representation of the compressed signal. The second is the storage of this representation to permit efficient retrieval of matches.

In order to evaluate the usefulness of this tool, it will be tested on an existing database of dolphin whistles. These whistles are narrowband with fundamental frequencies typically in the range 2-25 kHz. Whistle duration is on the order of a second. Dealing with dolphin whistles offers certain advantages over other types of signals such as speech, music, seismic, etc. First of all, the dolphin whistle is a relatively simple signal. Many investigations of whistles (e.g. [3]) suggest that most of the information is contained in the fundamental frequency of the whistle. This simplicity is not available in speech or most music. Second, dolphin whistles are repeated often by the animals. Studies conducted with wild bottlenose dolphins (*Tursiops truncatus*) [34] have shown that these dolphins tend to have a preferred whistle (also known as a signature whistle) which accounts for over 70 % of their whistles. This allows one to concentrate on the task of finding repetitions without having to worry about their existence. Such a high degree of repetition may be forced upon speech or music signals but is unlikely to be found in seismic signals.

Third, the dolphin whistle is an evolving vocabulary. As new generations of dolphins are born and old ones die, new types of whistles are created and old whistles stop being used. For example, in the same study [34], it was found that female calves will tend to develop a signature whistle very distinct from that of the mother. Also, although signature whistles have been observed to remain stable for a number of years (12) there is no conclusive evidence that once a signature whistle is developed it remains the same for the life of the dolphin. Dolphin's vocabulary may change as they meet other dolphins and imitate their whistles.

Finally, an expanding database containing approximately 1000 hours of data recorded on analog tape (reel, cassette and Hi-Fi VHS) is available at the Woods Hole Oceanographic Institution (WHOI). The database includes many high quality (strong signal-to-noise ratio, little reverberation) whistles available for this work. Also, the continuous expansion of the database increases the likelihood that the tool being developed will not only find use now but also for some time to come.

The goal of this work is to develop a system which serves as a simple interface to a compressed database of dolphin whistles. Besides the standard function of adding/deleting whistles from the database, the interface will have two main functions. First, as new whistles are added to the database, the system will identify those whistles for which a record already exists. Second, if a repetition is not found, the system will identify whistles which are close to the new whistle. The system to be developed may be compared to a spell checker for whistles. A correctly "spelled" whistle will be one which already exists in the database. If the whistle is not "spelled" correctly, alternate spellings are provided. Other types of database functions beyond the two mentioned above (e.g. range queries[3]) will not be implemented.

The goal of compressing and retrieving dolphin whistles is achieved incrementally in three major steps. With each step the amount of compression is increased at the expense of a loss in the information content of the compressed signal. This loss means that at each step one is less able to do an exact reconstruction of the original signal. Three steps, called levels below, are used.

The first level does the initial conditioning of the data followed by a reversible compression. This level is needed to reduce the amount of data to a more manageable quantity. The second level takes the output of the first and does a partially reversible compression. The third level does irreversible compression. The output of the third

[3] An example of a range query for the whistle database would be to request all whistles of duration between d_1 and d_2.

level just has enough information to permit the identification of the signal but not to reconstruct the original signal in any way.

In addition to the above, this work also produced techniques for selecting the representation of the compressed data. The representation involves the selection of the dimensions to use in the coding space. This coding space must be designed so as to capture as much of the significant underlying information in the signal as possible but a minimum of the extraneous information. In addition, it should be invariant to external factors added by the processing of the signal. The next few paragraphs will cover in more detail some of the problems associated with the development of the coding space.

One problem is the uncertainty in the location of the start and end of the whistle in time which makes it hard to assign a time reference to the whistles. This problem, as well as a desire to recognize partial whistles, increases the difficulty of the identification task. It was initially proposed to use whistle-dependent reference points in order to remove this uncertainty. Unfortunately, the reference points did not work as well as expected. Therefore, partial whistles can only be matched to other partial whistles.

Another problem is the relative variation in duration and both the absolute and relative variation in frequency of dolphin whistles. An example of the absolute variation in frequency can be observed in [33] where a dolphin reproducing specific sounds does so at a completely different absolute frequency than that of the original sound in one of the experiments. To address this problem some of the dimensions in the coding space will have to be designed with immunity to these variations.

In addition, one can not expect the data used in the development of the coding space to contain all possible whistles. Thus, the coding space must be general enough to accommodate new whistle types as they are encountered. Also, the system should be able to perform the match with a single whistle. One can not ask the dolphin to repeat the whistle a number of times in order to get a clearer version of it.

As mentioned before, this tool will be tested on an existing database of dolphin whistles. Currently, replications in this database are detected manually. For example, operators know that dolphins tend to repeat whistles from a small set. The set contains the signature whistle of the dolphin as well as copies of sounds heard by the animal, including the signature whistles of other dolphins. Given such a small set it is very simple for the operator to classify whistles just by looking at them. However, although the whistle has been classified as similar or not, a quantitative measure of similarity is still missing. There is also the strong desire to perform similar studies for larger data sets.

As part of this work, a distance measure for whistles has been developed. This distance is based on the euclidean distance of whistles in the coding space. It can be used as an independent analysis tool in the study of whistle mimicry by dolphins. It also offers a quantitative way of comparing dolphin whistles. Quantitative measures of similarity have been studied in the past [4] as a way to bring objectivity into sound comparisons. However, the method shown in [4] would not work given frequency and time shifts in the whistles.

1.3 Proposed Approach

As mentioned above, compression is done in three levels, with each level adding to the overall compression at the expense of the ability to reconstruct the original signal. Prior to level I, the dolphin whistles are digitized by the owners of the database at WHOI. The majority of the signals are digitized to 12 bits using a 50 kHz sampling rate, but that is not the only option available. The final database used in this work, contains 1169 whistles coming from 22 different animals.

1.3.1 Level I

The first step in this level is to remove all analysis windows (see below) which do not contain any interesting signals. This may correspond to windows with silence, or just background noise, recorded while waiting for the event of interest to occur. The removal of uninteresting windows is accomplished by a fast detector designed as part of this thesis. A marker is inserted to keep track of the duration of the silence so that it can be added during reconstruction.

Keep in mind that in many cases the signal one is interested in carries information in different ways, i.e. whistle duration, amplitude, etc. To accommodate situations like this one, the signal can be separated into potential information bearing streams. This would of course help with the identification of those characteristics of most importance to discriminating the signal. Initially, three information bearing streams were proposed. Namely, tone, amplitude and echo location (clicks).

The most important information stream is the tonal stream which captures information transmitted through the pitch of the whistle. This stream has been preserved as the basis for whistle identification. The amplitude stream was looked at briefly and found to be inadequate for whistle classification. As to the echo-location stream, the sampling frequencies used in this work were not high enough to study them properly. As discussed in [1], the echo-location signals coming from the dolphin may have peak energies above 100 kHz.

Finally, an analysis window (in time) was determined for the tonal stream and the whistles were converted to the frequency domain. Only magnitude information, in the form of a power spectrum, is preserved due to the (suspected) insensitivity of dolphins to phase information in sound waves. The resulting spectrum is quantized and stored in integer form.

1.3.2 Level II

In the previous level, compression is achieved by removing the silent windows and by the quantization of the power spectrum of each window. In this level further compression gains are attained by discarding selected information in the power spectra. The information kept is that which increases the chances of differentiating between distinct signals.

Based on the recommendation of scientists at Woods Hole, only the location (frequency) of the main peak in the power spectrum is preserved. Changes in this main peak as time progresses result in the frequency-vs.-time trace that gets sent to level III. Several algorithms for detecting spectral peaks were tried and evaluated. Also, an algorithm based on dynamic time warping was implemented in order to obtain a smooth trace.

1.3.3 Level III

The success of the whole system depends on the selection of an appropriate coding space. In this level the task of determining this coding space is completed. Various techniques for the evaluation of coding spaces and individual dimensions were developed for this level. In the end, a 16-dimensional coding space was selected.

Once the coding space was identified, the potential vocabulary the dolphin may have in this space was also investigated. Since this issue has not been directly addressed before, let us first expand on it. The compressed representation for a specific whistle can be considered to be a point in the coding space. As the specific details of a whistle change, one can expect a change in the point used for representing the whistle. All these points define a region whose size determines how much space is taken by a particular signature whistle. That is, although the animal is repeating the same whistle, since it does not repeat the whistle exactly, instead of getting a single point in the coding space, one ends up with a small cluster of points. The ratio

of total space volume to average cluster volume determines an upper bound[4] to how many of those clusters the space can support, which in turn determines the potential animal's vocabulary.

In addition to the vocabulary experiment, the coding space was used for the evaluation of search techniques. Two search techniques were implemented as part of level III in addition to the full search method.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 covers level I compression. The animal detector, power spectrum estimator, power spectrum quantizer, and size of analysis window are all discussed in this chapter. Chapter 3 is devoted to the level II compression and the development of the tracing algorithm. The chapter also compares several methods for detecting spectral peaks. Attempts to improve the level I animal detector using the frequency domain information in level II are also included in the chapter. The chapter closes with a look at the cepstrum of dolphin whistles.

In Chapter 4, the details on how to select a coding space are discussed. The chapter starts by listing possible dimensions for the coding space based on human knowledge about what makes a trace look like another trace. Then, measures for the quality of individual dimensions and coding spaces are discussed. A 16 dimensional coding space is finally selected, for which an estimate of the potential animal's vocabulary is found. Chapter 5 uses the coding space from Chapter 4 to study different search algorithms. The algorithms are compared based on the number of comparisons required to find the closest match, the number of comparisons needed to confirm the closest match, the total number of math operations, and the average execution time.

Finally, the conclusions reached after all this work are given in Chapter 6, which also contains some ideas for future work in this area.

[4]Only an upper bound is obtained since the clusters may not pack tightly.

Chapter 2

Level I Compression - Reversible

This chapter describes the level I compression scheme. Level I compression was designed from the start to be the most reversible compression stage. Since it involves lossy compression, exact reconstruction of the original signal is not possible. In addition to discussing the main components of the level I system, i.e. signal detection/silence removal and conversion to the frequency domain, the chapter will also cover the implementation of these components.

It is assumed that the input to the level I stage is a digitized stream of 1-D data. All that is known about the data is the sampling frequency used and the number of bits used to quantize each data point. For our specific application the sampling frequency may be 40.96 kHz, 50 kHz, or 81.92 kHz with 50 kHz being the most common. The number of bits used in the quantizer is 12 but quantized values are zero-padded to 16 bits for storage. It is assumed that the data has been properly filtered before sampling and that saturation has been avoided while maintaining a good dynamic range.

In addition to the anti-aliasing analog filter, a digital high-pass filter is also used. The filter is a 2nd order Butterworth filter with coefficients changed according to sampling frequency. The coefficients of the filter are chosen to maintain a cutoff frequency of 2 kHz irrespective of sampling rate.

2.1 Signal Detection

Signal detection refers here to the process of identifying which portions of the data stream contain signal and which contain only background noise. Several areas related to this problem can be found in the literature, for example, silence detection (identification of breaks between signal and silence), endpoint detection (beginning and end of signals) and background removal (eliminating undesirable signals). However, the intent of these areas is the same, to be able to mark each *analysis window*[1] as being produced by the process of interest or just by the background environment. The duration of this analysis window is discussed below.

Note that when dealing with endpoints, one goes from a single window or local identification to a multi-window or global identification. That is, there is a structure that spans many windows and one tries to identify where it starts and stops. Silence detection is part of endpoint detection, but silence detection does not require endpoint detection. This distinction is important since the signal detection algorithm developed in this thesis is a kind of endpoint detector and therefore it will require a single window classifier (silence detector) and some algorithm for finding the endpoints based on the output of the silence detector.

Algorithms for signal detection are as diverse as the number of applications. That is because signal detectors in general are very heuristic. The algorithm developed in this thesis is no exception. Thus, it is not only important to cover the end result, i.e. the signal detection algorithm, but also the methods used in deriving this detector. That way, if the actual detection algorithm is not applicable to a different application of this work, the method can still be used to develop a suitable detector.

It is desired that the signal detection algorithm be able to work in "real-time" fashion. This means that the signal detector is restricted either to examine each window once, or be responsible for buffering the data in case more than one analysis

[1] In general, identification is not made on a sample-by-sample basis but rather in terms of an analysis window (or frame) of constant duration.

window is required for the decision. This would allow the signal detector to be used between the A/D converter and disk storage to minimize the amount of disk space used.

2.1.1 Background on Signal Detectors

A statistical silence detector is described in [38]. This detector uses an F test with significance level P to classify data frames as being silent or nonsilent. The value of P offers direct control over the error rate of the algorithm. The algorithm also adapts and normalizes itself by continually updating the mean and variance used in the F test. The detector uses the mean and variance of a 5-dimensional parameter vector computed from the data in the frame. The parameters should be approximately normal (Gaussian) and contain information relevant to the detection being done.

The statistical silence detector does require that means and variances be properly initialized. This is accomplished by requiring the data stream to start with silent frames. For the application studied in this thesis, this requirement was found to be too restrictive. Also, there was doubt that an appropriate set of Gaussian-distributed parameters could be found. Thus, an F test based signal detector was not used.

References [18, 11] are examples of the use of silence detection to compress speech signals. In both papers, the silence segments are replaced by a reduced set of parameters (e.g. duration and amplitudes). During reconstruction these parameters are used to generate a pseudo-noise signal that replaces the original silence. In [18], signal vs. silence identification is done one sample at a time with some hysteresis added in to avoid noisy endpoints.

The algorithm in [18] keeps track of short-term vs. long-term "energy" and uses a set of heuristics to decided when to switch from signal (speech) to silence. In addition to fixed gain thresholds for identifying each sample as either signal or silence, the algorithm requires a number of fixed time-constants. These time constants are set based on the specific application of the algorithm. In order to find the correct values

for a different signal to that used in [18] a large set of data would need to be analyzed. Given the amount of overhead required, this algorithm was not used.

On the other hand, [11] has an algorithm based on short-term "energy" and zero-crossing rate computed using a 100-sample analysis window. Oversimplifying, when either of these 2 measures is larger than a threshold, the window is classified as speech and a speech segment started. Likewise, when both measures are below the thresholds the window is classified as silence. Actual implementation of the algorithm is slightly more complex since the energy threshold is shifted up and down depending on whether the transition being expected will end or start a speech segment. Also, minimum durations are required before the threshold is changed. The idea of adapting the measure thresholds is utilized by the algorithm developed in this thesis.

One final example of the use of endpoint detectors in speech processing is given in [21]. This algorithm produces a set of candidate endpoints based on the logarithm of the energy in the analysis windows. An "average" noise background level is removed from each window before processing. The set of candidate endpoints is used by the word-recognizer stage, different endpoints are tried while maximizing recognizer performance. The coupling between recognizer and detector helps in reducing some of the problems of stand-alone energy-based detectors. Such a coupling is not possible in the system developed here.

Specifically for the application being considered, there are 2 methods currently in use for signal detection. The first method was developed by Kurt Fristrup (at the Woods Hole Oceanographic Institution) for detection of endpoints in selected cuts of data. In this method, two passes of the data are required. In the first pass, energy values are computed for approximately 100 windows of data evenly spaced over the cut. Then, a histogram is made and an energy threshold is computed from it. An average power spectrum is also computed. In the second pass the endpoints are detected by comparing the energy in the current window to the threshold.

The second method, developed by Terrance Howald [13], expands on the previous algorithm. First, an average noise spectrum is computed from a first pass of the data. This noise spectrum is then multiplied by a gain set by the user and subtracted from the power spectrum computed for each window. All frequencies with power below that of the amplified noise background are set to zero power. There is also a frequency cutoff, again set by the user, which acts as a high-pass filter. The power of all frequencies below the cutoff is set to zero. If there are any frequency bins with power left in the window, then the window is considered to be signal.

The signal segment begins with the first window labelled as signal and ends after a certain number of consecutive background windows have passed. Again, the user chooses the number of windows to wait based on his knowledge of the signal. Finally, the duration of signal segment is check d against a minimum allowed length parameter. If the segment is too short, it gets discarded by the algorithm.

In addition to being a two pass algorithm, this algorithm works entirely in the frequency domain. Thus, it requires considerable more computational power than other time-domain based methods.

2.1.2 Duration of Analysis Window

Since the signal detector developed in this thesis works on a window basis, that is, classification is made one window at a time, the first task is to determine the duration of the window to use. The desired window should be large enough so that the data captured in it can produce reliable measures. On the other hand, it should be small enough to avoid joining silence and signal segments into the same window. This could lead to having entire signal segments be rejected due to the amount of silence in the window.

There is yet another constraint on window size. After signal detection, the streamed data is then converted to the frequency domain. This conversion process is also done one (non-overlapping) window at a time. In practice it is not required that

the detection window and frequency-conversion window be the same size, only that one be a multiple of the other. However, to simplify implementation it was decided to make them the same size. This adds another constraint to the window size in the form of a time resolution criterion. Since spectral frequencies are computed for each window, window duration should be set in such a way as to provide adequate sampling of the frequency in time.

A window of 512 points in duration was selected for the analysis. Depending on sampling rate, this window can be 12.5, 10.24, or 6.25 ms in duration. The slower frequency sampling rate, 12.5 ms, was still deemed acceptable to provide good resolution. Given an average signal duration larger than 500 ms, there should be little chance of a signal segment being discarded. Also, 512 samples are sufficient to compute good statistics of the data in the window.

2.1.3 Development of Signal Detector

The initial signal detection methods used in this thesis were based on power and/or the sign of the 2nd order correlation coefficient. The desired parameter was computed for each window and compared to a fixed threshold. Based on the comparison the window was labelled as signal or silence. After some mixed results with these detectors it was decided to expand the set of measures and the number of records used for testing.

Two files of 10 Mbytes each (10240 windows) were used for this study. One file was used for development and adjustment of the thresholds while the other was strictly used for testing and evaluation. The endpoints of the signal segments in each file were set manually. Windows were then classified as signal or silence based on the manually-identified endpoints. Direct manual classification at the window level was not possible. This has to be taken into account when evaluating the performance of the signal detector since it works at the window level.

Measure	Classification
power	power-based
absolute sum	power-based
amplitude range	power-based
2nd order autocorrelation coefficient	frequency-based
3rd order autocorrelation coefficient	frequency-based
4th order autocorrelation coefficient	frequency-based
5th order autocorrelation coefficient	frequency-based
number of turns	frequency-based
normal zero-crossings	frequency-based
quantized zero-crossings	frequency-based
normalized frequency measure	hybrid

Table 2.1: Signal detection measures.

The set of measures used in the study is given in Table 2.1. The measures can be classified under three categories: power based[2], frequency-domain based, and hybrid. Assuming the input signal is $x[n], n = 0, 1, ..., 511$ the mathematical definition of each measure is as follows.

1. Power-based measures. This set of measures is based on the fact that windows with signals should generally have more power content than those only containing background. Of these detectors, the log of power was used in [38].

   (a) power - simple sum of squares

$$P = \sum_{i=0}^{511} x[i]^2 \tag{2.1}$$

   (b) absolute sum - Sum of absolute value of each sample. This measure is very

[2]Instead of power, it is perhaps more appropriate to call these measures energy based. However, the difference between energy and power here is just a scaling factor which does not affect the performance of the measure.

similar to power but much less computationally intensive.

$$AS = \sum_{i=0}^{511} |x[i]| \tag{2.2}$$

(c) amplitude range - Difference between the largest and smallest amplitude in the window. Very sensitive to sample outliers.

$$AR = \max_n(x[n]) - \min_n(x[n]) \tag{2.3}$$

2. Frequency-based measures. This set of measures attempts to extract frequency information from simple time-domain measurements. In order to interpret and properly use the frequency information, knowledge about the spectral content of signal and background is required. Of these measures, the 2nd order autocorrelation coefficient, normal zero-crossings, and number of turns appeared in [38].

(a) autocorrelation coefficients - The autocorrelation function (ACF) $r(\tau)$ is the Inverse Fourier Transform of the power spectrum. Therefore, each autocorrelation coefficient represents the integral of the power spectrum of x after being scaled by a different cosine function. This is illustrated by the following equations. Let $S_{xx}(\omega)$ be the power spectrum of $x[n]$. If $x[n]$ is a real signal, then

$$\begin{aligned} r(\tau) &= \frac{1}{2\pi}\int_{-\pi}^{\pi} S_{xx}(\omega)e^{j\omega\tau}d\omega \\ &= \frac{1}{2\pi}\int_{-\pi}^{\pi} S_{xx}(\omega)\cos(\omega\tau)d\omega \end{aligned} \tag{2.4}$$

Several different autocorrelation coefficients were evaluated as pointed out

in Table 2.1. The equation used for their computation was,

$$R(\tau) = \frac{1}{512-\tau} \sum_{i=0}^{511-\tau} x[i] * x[i+\tau] \tag{2.5}$$

(b) number of turns - Measures the "jaggedness" of the data in the window. Since the signal is expected to be of higher frequency than the background, a large number of turns should indicate its presence. The evaluation of this measure was based on a counter which was incremented whenever $x[i-1] < x[i] > x[i+1]$ or $x[i-1] > x[i] < x[i+1]$ for $i = 1...510$. This measure is the same as the number of zero-crossings of the 1st derivative of $x[n]$.

(c) zero-crossings - As with the last measure, zero-crossings increase when high frequencies are present in the windowed data. To evaluate the number of zero-crossings a counter is incremented each time $\text{sign}(x[i-1]) \neq \text{sign}(x[i])$ for $i = 1, 2, ..., 511$. If either $x[i]$ or $x[i-1]$ were zero, the counter was left unaltered and the next value of i used. Two versions of the zero-crossing measure were implemented. The first is described above. The second used a rounded version of $x[i-1]$ in the comparison whose effect will be explained in more detail later.

3. hybrid - Only one hybrid measure was tried. The measure is known as the normalized frequency measure. It is calculated as follows,

$$NFM = \frac{\sum_{i=0}^{510} |x[i+1] - x[i]|}{\sqrt{\sum_{i=0}^{511} x[i]^2}} \tag{2.6}$$

The base 10 log of this measure was used in [38].

Measure	Threshold	P_{FA}	P_{MA}
power	85693.0	0.1033	0.1033
absolute sum	5054.7	0.0967	0.0969
amplitude range	70.6250	0.1187	0.1186
2nd autocorrelation coefficient	0.1570	0.1597	0.1597
3rd autocorrelation coefficient	1.7531	0.4850	0.4850
4th autocorrelation coefficient	4.5719	0.2449	0.2449
5th autocorrelation coefficient	−0.0476	0.4495	0.4494
number of turns	235.[illegible]25	0.4359	0.4402
normal zero-crossings	177.50	0.1419	0.1450
quantized zero-crossings	90.00	0.0451	0.0469
normalized frequency measure	18.2715	0.1615	0.1616

Table 2.2: Performance of signal detection measures.

Once a set of measures for signal detection was found, the next step was the computation of probability density functions (p.d.f.'s) for each one. Two p.d.f.'s were found, one for windows labelled as signal, the other for windows labelled as background. Of the 10240 windows used in the test file, 3262 were signal and 6978 were background. A good measure for the identification of signal vs. silence would show p.d.f.'s that are non-overlapping, narrow, and very far apart from each other.

The p.d.f.'s were estimated using histograms. Figures 2-1 through 2-3 show the various histograms obtained. In each plot the histogram of the populations labelled as signal is in dashes, the one for backgrounds is solid. For a given threshold T one can compute probabilities of false alarm (P_{FA}) and of miss (P_{MS}). A false alarm occurs every time a background window is identified as signal. A miss occurs when a signal window is identified as background. In Table 2.2 the threshold value which minimizes these two probabilities is shown. The table also includes the final probabilities obtained at the given threshold.

As can be seen from both the table and the figures, performance varies considerably from one measure to another. In general, power-based measures performed better than hybrid and frequency-based measures. However, the best overall per-

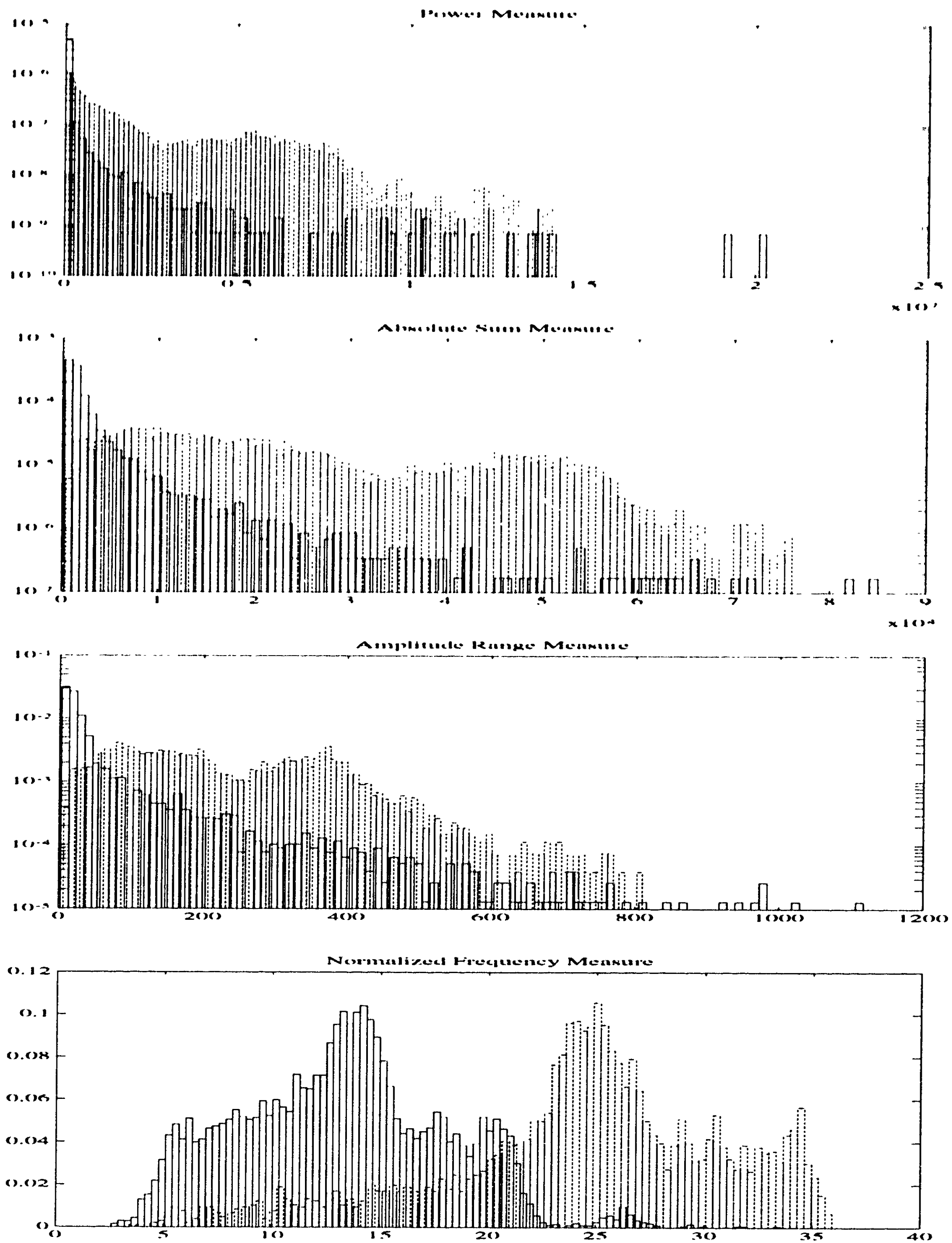


Figure 2-1: Histograms of signal detection measures. - Power-based and hybrid measures as labelled. Two distributions shown; solid lines used for signal-labelled distribution and dashed line used for background-labelled distribution.

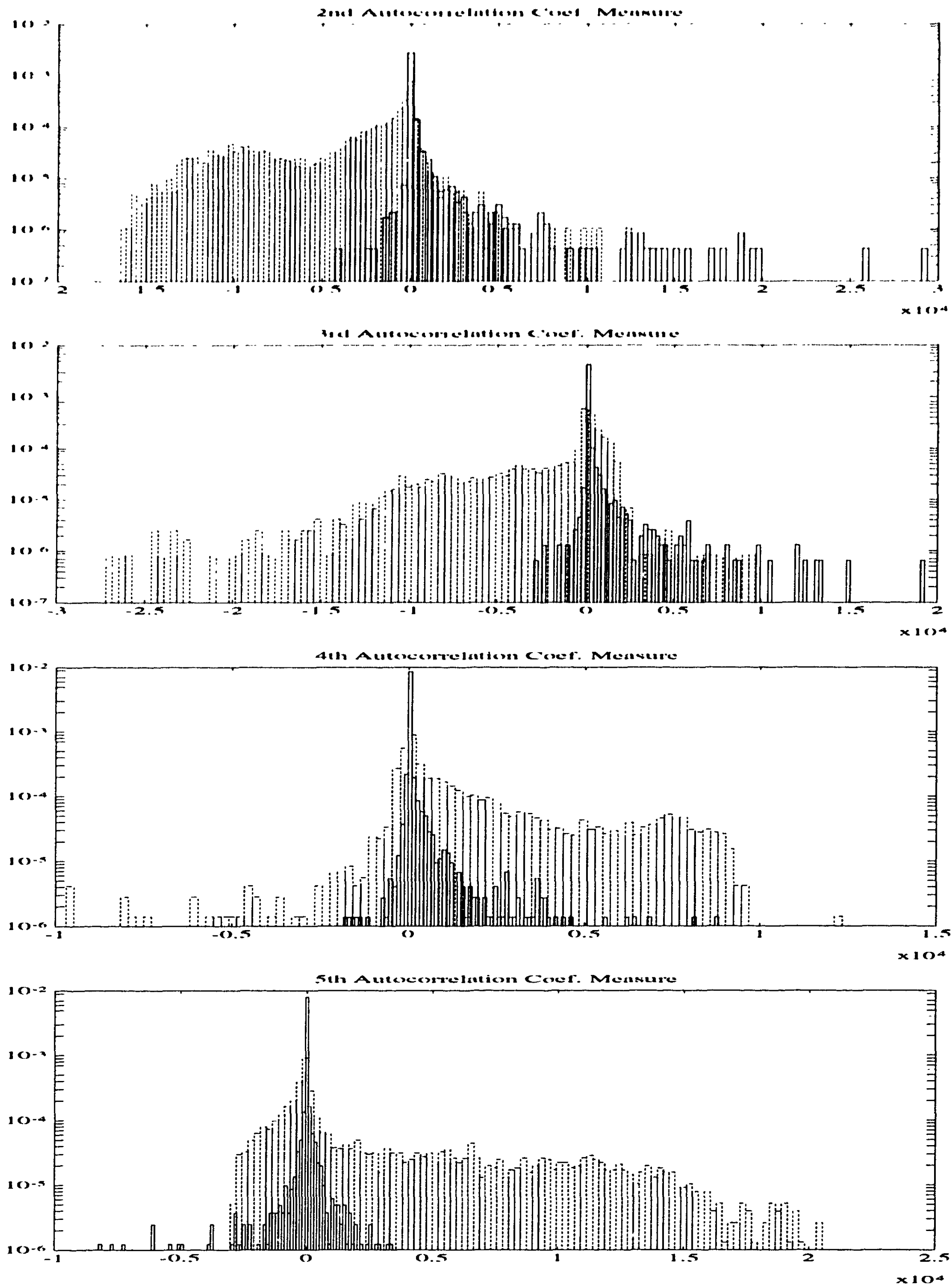


Figure 2-2: Histograms of signal detection measures. - Frequency-based measures as labelled. Two distributions shown; solid lines used for signal-labelled distribution and dashed line used for background-labelled distribution.

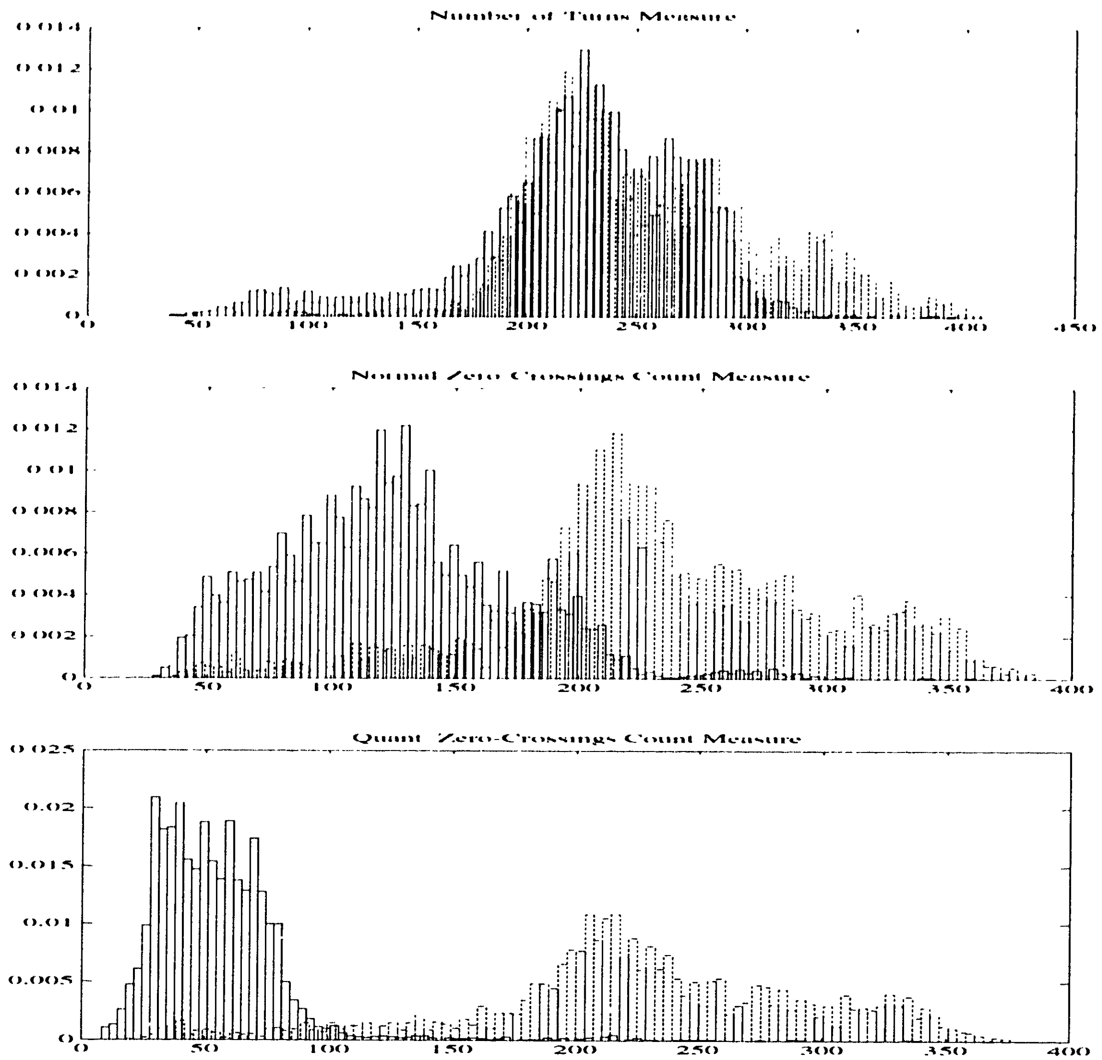


Figure 2-3: Histograms of signal detection measures. - Frequency-based measures as labelled. Two distributions shown; solid lines used for signal-labelled distribution and dashed line used for background-labelled distribution.

formance was obtained from the quantized zero-crossings count, a frequency-based measure.

After this initial testing of all the measures, work concentrated on improving the best measures found. Three different approaches were taken to the task of optimizing the signal detection measure. First, examine if linear combinations of the above measures can achieve better performance. Second, evaluate performance of decisions made based on 2 consecutive windows instead of a single one. Third, given the best measure found by the 2 previous steps, analyze how the measure parameters may be selected to achieve better performance.

Using Fisher's method [17] a linear combination of the best 2 measures was found and evaluated. The linear combination found maximizes the ratio of the distance between population means over the variance of the linear combination. Let m and y be the absolute sum and quantized zero-crossing counts, respectively. Then, the new measure z is given by $z = 2.24743 \times 10^{-3} * m + 0.999997 * y$. The optimal threshold for z was 105.976 and it resulted in a $P_{FA} = 0.0447$ and a $P_{MS} = 0.0448$.

Fisher's method assumes that the variance of the linear combination is the same independent of the population. I modified the method slightly and instead found the linear combination which optimizes the difference between population means over the sum of the population's variances. The result was a new z measure given by $z = 2.21728 \times 10^{-3} * m + 0.999997 * y$. The optimal threshold was 105.625 and the probabilities were $P_{FA} = 0.0446$ and $P_{MS} = 0.0445$. Figure 2-4 shows the two resulting p.d.f.'s for the linear combination.

The modification to Fisher's method provided only a small performance improvement to the linear combination measure. When comparing the best linear combination performance to that of the quantized zero-crossing count one can see that the difference between the two is small. Given such small improvement it did not make sense to compute a linear combination of measures. That is, the expense of evaluating two

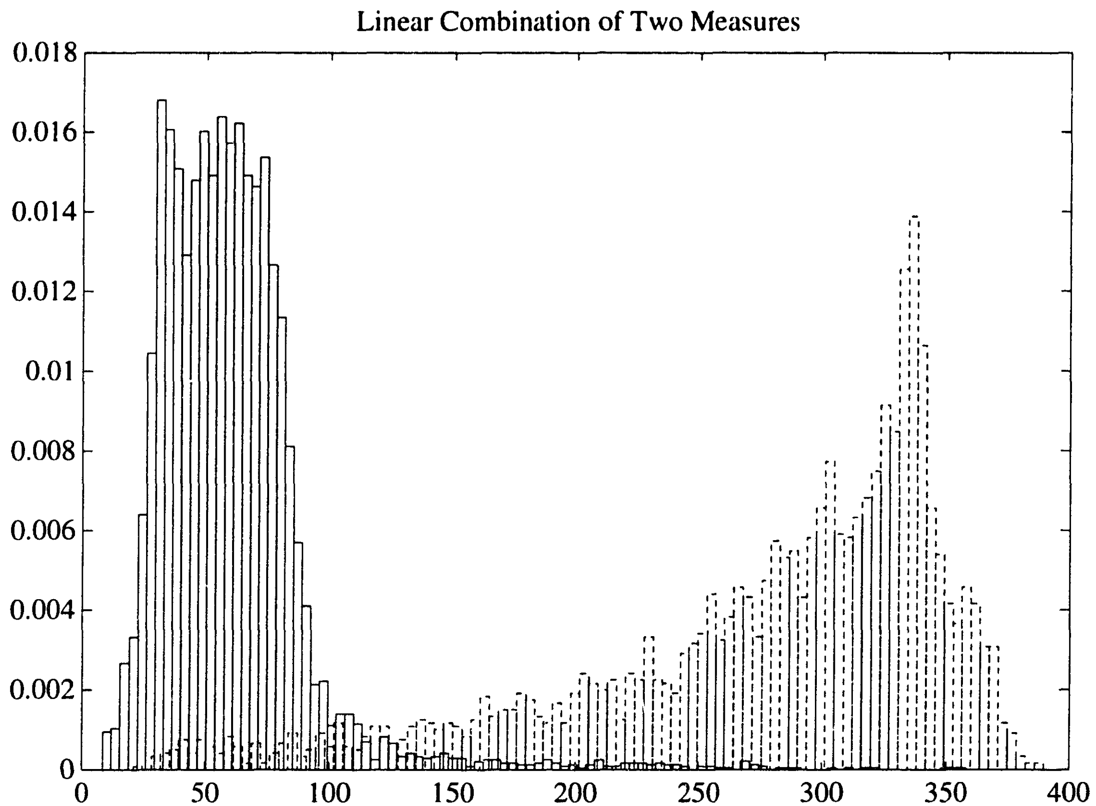


Figure 2-4: Histogram of linear combination measure. - If m is the absolute sum measure and y the quantized zero-crossings count measure then, the new measure is $2.21728 \times 10^{-3} * m + 0.999997 * y$. Two distributions shown; solid lines used for signal-labelled distribution and dashed line used for background-labelled distribution

measures and then forming a linear combination with them is not justified given the small increase in performance.

The work done in combining 2 windows at a time considered 4 different ways of combining the windows as well as forward and backwards implementations. The four functions used were the following:

1. max(·) - backwards: $z[n] = \max(m[n], m[n-1])$
   forwards: $z[n] = \max(m[n], m[n+1])$

2. min(·) - backwards: $z[n] = \min(m[n], m[n-1])$
   forwards: $z[n] = \min(m[n], m[n+1])$

3. average(·) - backwards: $z[n] = 0.5 * (m[n] + m[n-1])$
   forwards: $z[n] = 0.5 * (m[n] + m[n+1])$

4. low-pass(·) - backwards: $z[n] = 0.9 * m[n] + 0.1 * m[n-1])$
   forwards: $z[n] = 0.9 * m[n] + 0.1 * m[n+1])$

In the above, m is the value of the quantized zero-crossing count and z is the new measure being formed by combining the 2 windows. The variable n indicates the index of the window. Backwards and forwards descriptions of all functions are given for the sake of completeness. However, it should be clear that for the first 3 functions the decision between forwards and backwards just introduces a one element delay in the resulting measure. Only for the last function does a forward implementation results in different numbers than those of the backwards implementation.

Of the four functions, best results were obtained with max(·). The optimal threshold was 101 which resulted in $P_{FA} = 0.0431$ and $P_{MS} = 0.0435$. These new probabilities indicate a better detector is possible, however, gains in performance do not justify the additional algorithmic complexity.

2.1.4 Improving the Signal Detector

At this point the quantized zero-crossing rate was selected as the best measure for signal detection. The remaining discussion will concentrate on work done to increase its performance and how an endpoint detector was developed based on it.

The quantized zero-crossing count is so called because when comparing samples $x[n-1]$ and $x[n]$ to determine if a zero-crossing has occurred, the value of $x[n-1]$ is quantized to an integer[3]. Quantization is done by truncation because it is very easy to implement. That is, numbers are rounded towards zero instead of the nearest integer. Since a zero-crossing does not occur if one of the 2 samples is zero, quantizing has the effect of creating a "dead zone" around zero. All samples of magnitude less than one are quantized to zero and can no longer produce a zero-crossing.

Figure 2-5 provides examples of what is and is not a zero-crossing. Each circle in the figure represents a data sample while squares are used to mark between which samples a zero-crossing occurs. The dashed lines are used to mark the dead zone. Basically, for a zero-crossing to occur the signal must start outside the dead zone and switch signs with no samples or only zero-valued samples in between. For a signal to have a high quantized zero-crossing count it must be both of high amplitude and high frequency. These are precisely the characteristics present in dolphin's whistles.

As mentioned before, the optimal threshold T for this measure is 90.00. Clearly, this optimal value is dependent on the width of the dead zone. If the zone is made wider, the number of zero-crossings and the optimal value of T will go down. If the zone is narrowed the reverse will occur. Another way the optimal threshold will be affected is if the overall gain of the data changes. Let us say that by operator intervention or by changing the tape being analyzed one gets a stream of data with an effective gain of twice that of the tape used for setting the threshold. For the threshold T to remain optimal the gain must be accounted for by either T or the

[3]Samples are initially integers (when digitization occurs) but become floating-point values at the output of the digital high-pass filter mentioned at the beginning of the chapter.

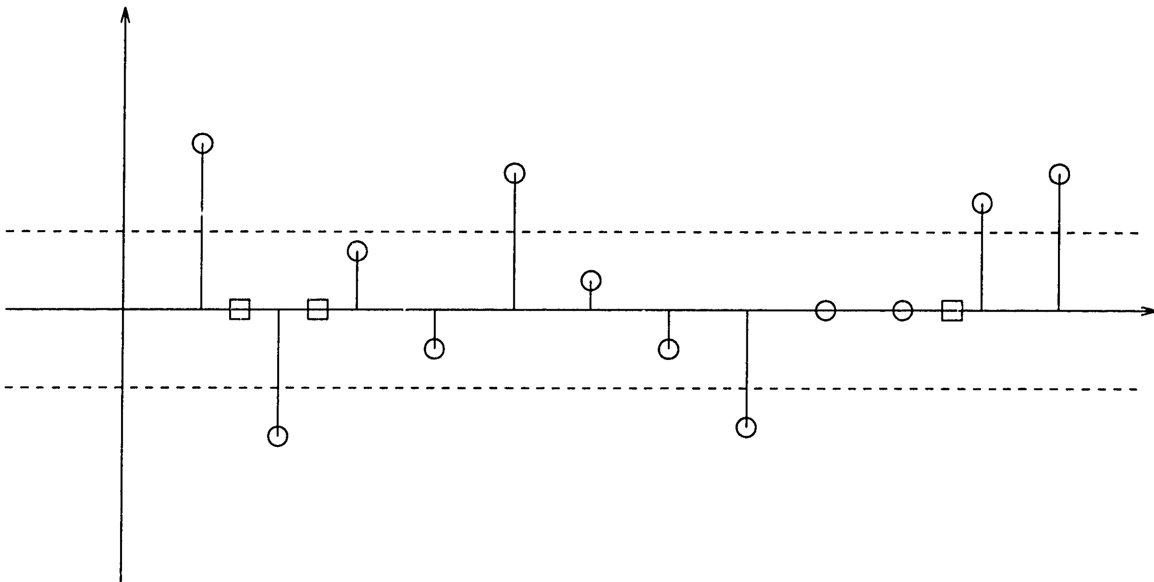


Figure 2-5: Example of zero-crossings. - Squares are used to mark in between which samples a zero-crossing occurs. The "dead zone" introduced by quantization is denoted by the dashed lines.

dead zone width. That is, if T remains at 90.00 then the zone must be twice as wide. If the zone's width is left unchanged, then a new threshold must be found which most likely will not be a simple multiple of the old threshold.

Since the relation between signal gain and dead zone width is much clearer than the relation between signal gain and threshold, it was decided to leave the value of T fixed and adjust zone width. Several experiments were conducted to investigate ways of adapting the width to improve performance. The first experiments involved setting the width of the dead zone to be a fraction of the average magnitude of the data in the window. However, this approach was improper. For example, consider a window with just background, the average signal will be small which in turn leads to a small dead zone width. However, the small zone width makes it easier for the quantized zero-crossing count to be above the threshold and have the window labelled as signal.

Use of a simple filtered average does not solve the above problem since long periods of background noise can still result in very small zone widths. However, filtering is

a step in the right direction because it bases the dead zone width on all the data observed thus far instead of the current window.

The algorithm finally used to determine the width of the dead zone does rely on a filtered value. The algorithm computes the average amplitude of each analysis window and then passes it to one of two filtered means, loM and hiM. Roughly speaking, the high mean (hiM) corresponds to the amplitude average of signal windows and the low mean (loM) to the amplitude average of background windows. By keeping two averages, the use of filtering now works when before it failed.

In order to determine which of the two filtered averages gets the computed average window amplitude the variance of loM is used (in addition to the loM and hiM values). The loM variance, σ^2_{loM}, is computed over the last 20 background windows[4]. A total of 20 analysis windows are used in order to get a good variance estimate.

Using the amplitude average of the current window, call it M, and the variance of loM, σ^2_{loM}, the mean to be updated is determined as follows. If $M \geq hiM$ then the value of hiM is updated. Similarly, if $M \leq loM$, the value of loM gets updated. When the value of M is between loM and hiM, $loM < M < hiM$, then the variance of loM and a gain G are used. If the distance between M and loM is smaller than $2G\sigma_{loM}$ then update loM, otherwise, update hiM. That is, if the mean M is less than $2G$ standard deviations away from loM, then M is used in updating loM. The value of gain G is initially 1.0 and gets increased by 1.0 for every 10 windows that get processed without a change in loM. As soon as loM changes, G is set to one again and the count reset. This is an attempt to keep the value of loM from becoming a constant.

Note that regardless of which value gets updated, loM or hiM, the window has not yet been classified. Also, the equation used to update the filtered means implements

[4] When less than 20loM values are available just use all values except the largest. The largest loM value is never used in estimating the variance. When only one value is available, the variance defaults to 1.

a simple first-order low-pass filter. For example, when updating loM, $loM = (1 - \alpha)loM + \alpha M$. The value of α used was 0.1.

Classification is based on the zero-crossing count. After the values of loM and hiM have been updated as needed, the dead zone width is set to $2loM$. The computation of the zero-crossing count then proceeds as outlined before. Histograms of the 2 populations (same two as before) given the new adaptive form of the quantized zero-crossing count measure are shown in Figure 2-6. The optimal threshold is 73, resulting in $P_{FA} = 0.0530$ and $P_{MI} = 0.0527$. These probabilities are higher than those obtained for the quantized zero crossing count measure with a fixed dead zone, 0.0451 and 0.0469.

The adaptive measure has slightly larger false alarm and miss probabilities. However, we now have immunity against signal gain changes. For example, if the signal gain is increased by a factor of 2 the optimal threshold remains the same. This is not the case with a nonadaptive measure and therefore the adaptive measure is used for the endpoint detector discussed in the next section. Also, when tested on a different data set, the adaptive measure resulted in a $P_{FA} = 0.0298$ and $P_{MI} = 0.0185$, a result not too far from optimal for the second data set. The optimum of this second set is obtained with a threshold of 75, giving a $P_{FA} = 0.0271$ and $P_{MI} = 0.0253$.

2.1.5 Endpoint Detector

Having settled on the signal detector to be used in classifying windows, then the endpoint detector was developed. A flow chart of the detector is given in Figures 2-7 and 2-8. The endpoint detector is very simple conceptually and entirely based on the signal detector discussed above. The signal detector is applied to each window until it indicates the window is not solely background noise. Then, the start of the signal segment is set $N1$ windows before the current window. The value of $N1$ is small and it determines the size of the window buffer used to store old windows. To end the signal segment it is necessary for the signal detector to find $N2$ consecutive background

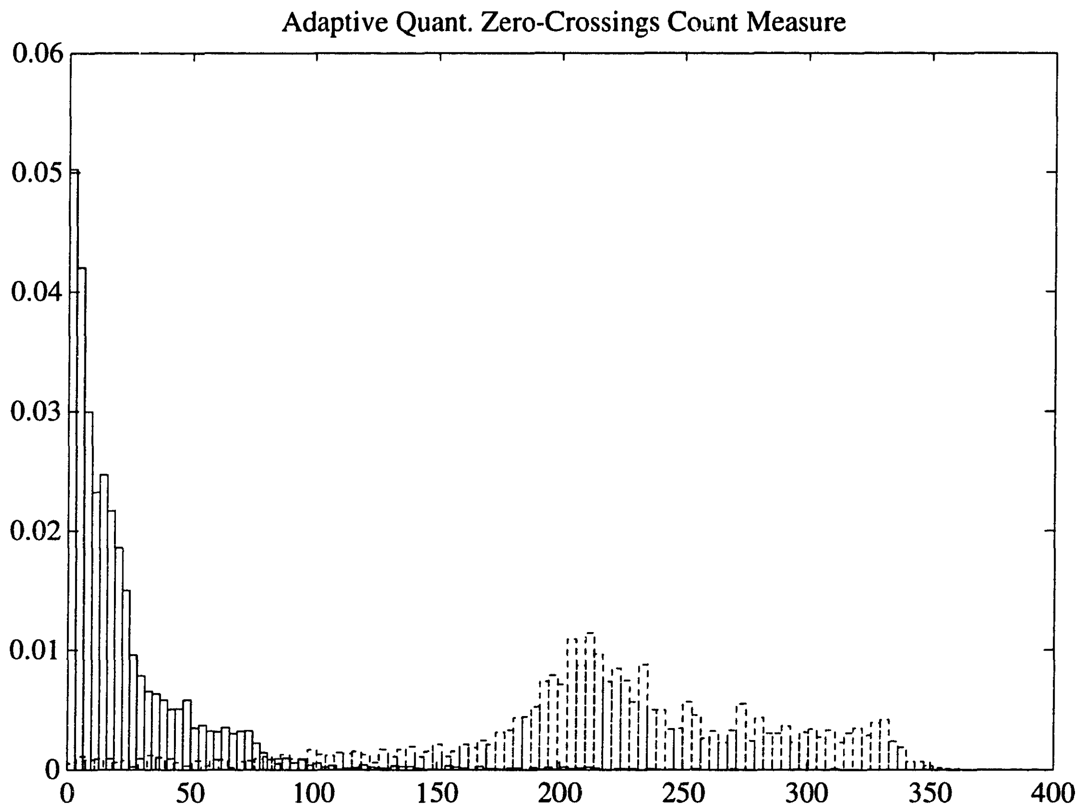


Figure 2-6: Histogram of adaptive quant. zero-crossing count measure. - Width of dead zone adapted according to a filtered low average, loM. Two distributions shown; solid lines used for signal-labelled distribution and dashed line used for background-labelled distribution

windows. After ending the segment, its duration (in windows) is computed. If the duration is less than or equal to $1.5(N1 + N2)$ then the entire segment is discarded. Otherwise, the segment is declared a valid signal segment and stored to disk.

For the current application, $N1 = N2 = 3$. That is, the segment identified as signal by the single window detector is extended 3 windows in each direction to form the signal segment. Any signal segment of duration less than 9 windows is rejected. Background segments of duration smaller than 3 windows inside a signal segment are classified as signal. The idea behind the extensions is to allow for fading of the signal. It was observed that many times the beginning and end of a whistle were weaker than the middle section. These weak signal windows may be misclassified as background. The addition of 3 windows at each end helps reduce the problem. Similarly, rejection was added to the endpoint detection algorithm to eliminate short noise bursts of high power which were classified as signal by the window-based detector. The minimum duration of a segment in order to be considered signal was determined based on knowledge of the particular application.

Evaluation of the endpoint detector was made using the same data used for the signal detector. In addition to computing the probabilities of false alarm and miss, I also kept track of how helpful the use of extensions and rejections was. Results are given in Table 2.3. In the table, wasted extensions refers to the number of windows that were background but got included in the signal segment when the segment was extended. Extension saves refers to windows that would have been misclassified as background but were correctly labelled thanks to the extensions. Similarly, invalid rejections refers to windows that were improperly rejected by the duration check while rejection saves refers to windows that would have been classified as signal when they were really background.

Table 2.3 shows 3 sets of results for each test file. Results are shown for a threshold of 73, the optimal for file 1, and a threshold of 75, the optimal for file 2. The extra set of results was obtained using a threshold of 75 but using a variable gain for the

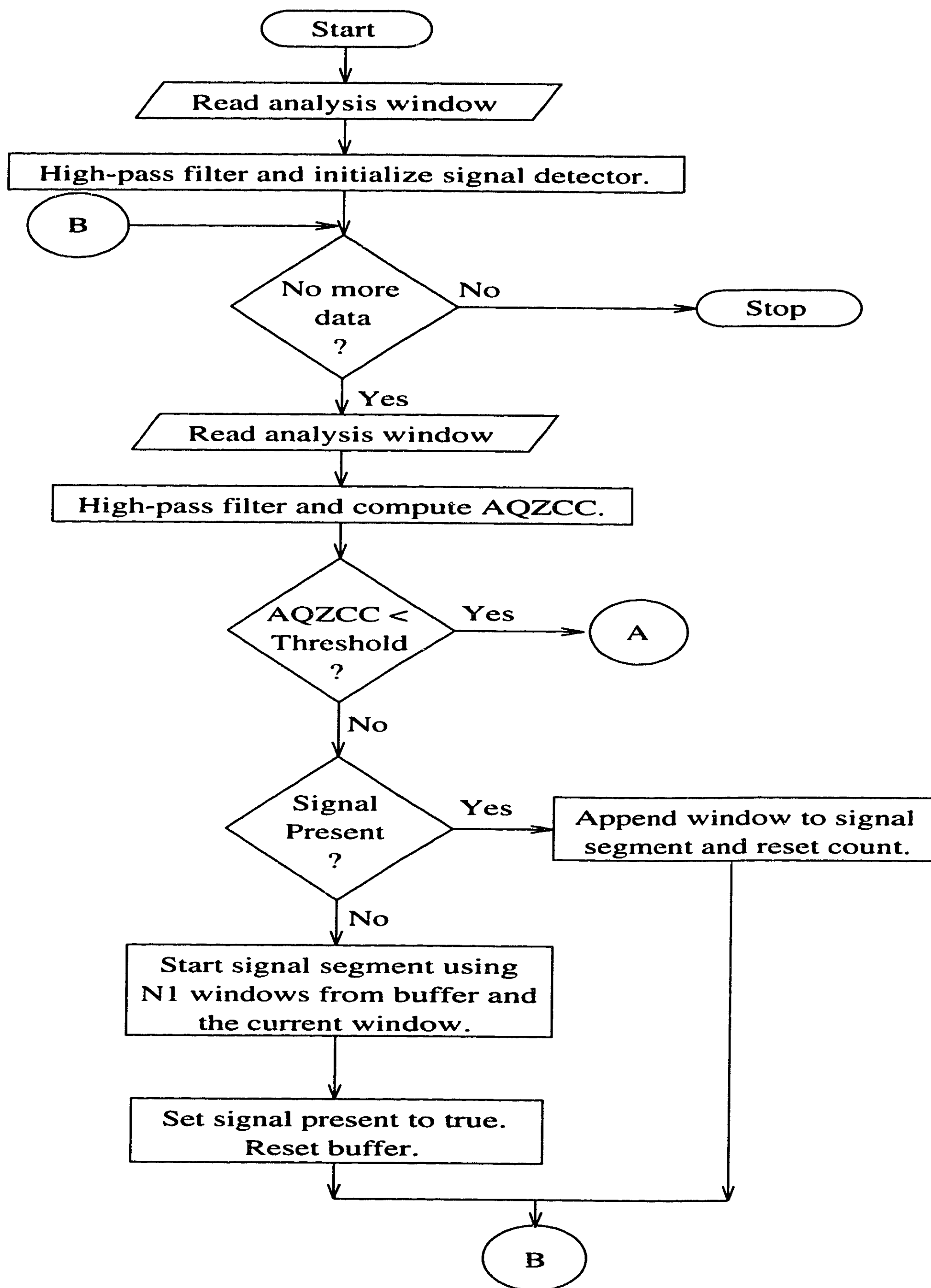


Figure 2-7: Endpoint detection algorithm. - Flow chart illustrating the main steps taken for endpoint detection. The variable AQZCC refers to the adaptive-quantized zero crossing count. A separate variable called count and a signal present flag (SPF) are also used. N1 = N2 = 3.

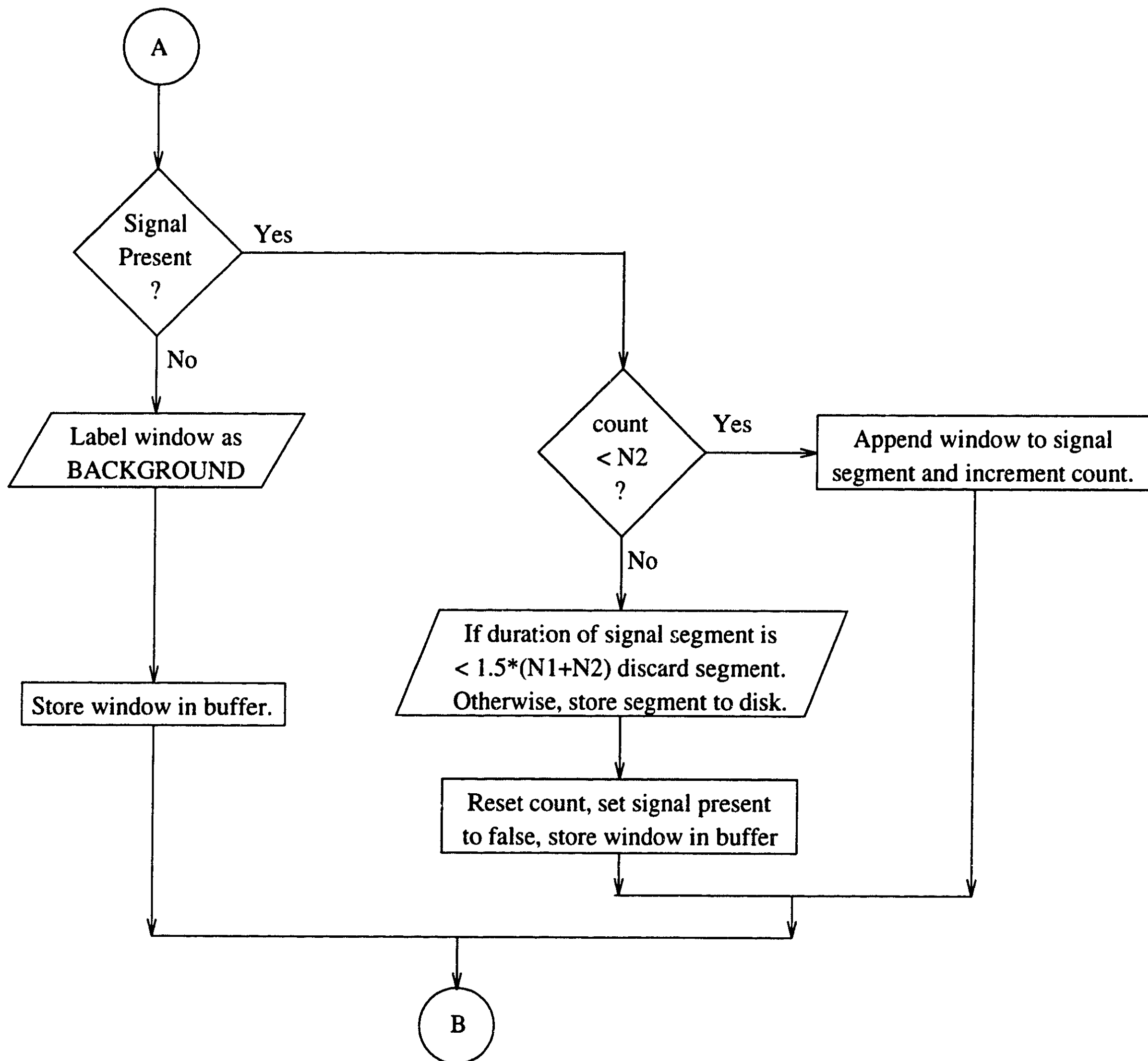


Figure 2-8: Endpoint detection algorithm. - Part 2 of the flow chart illustrating the main steps taken for endpoint detection.

Statistic	Test File 1		
	$T = 75$	$T = 73$	V. Gain
P_{FA}	0.0492	0.0546	0.0507
P_{MI}	0.0230	0.0227	0.0227
Wasted extensions	704 (0.1009)	717 (0.1028)	710 (0.1017)
Extension saves	102 (0.0313)	98 (0.0300)	100 (0.0307)
Invalid rejections	0 (0.0)	0 (0.0)	0 (0.0)
Rejection saves	50 (0.0072)	57 (0.0082)	50 (0.0072)
	Test File 2		
P_{FA}	0.0192	0.0214	0.4141
P_{MI}	0.0009	0.0009	0.0005
Wasted extensions	573 (0.0959)	613 (0.1026)	352 (0.0589)
Extension saves	104 (0.0244)	75 (0.0176)	67 (0.0157)
Invalid rejections	0 (0.0)	0 (0.0)	0 (0.0)
Rejection saves	344 (0.0576)	366 (0.0613)	218 (0.0365)

Table 2.3: Performance of endpoint detector.

data. That is, data was scaled by 0.5 for windows between 2001-4000 and by 2.0 for windows between 8000-10000. All other windows were unaffected.

As seen in the table, results were comparable for the 2 thresholds considered. Probability change observed due to the change in threshold is around 0.5 % at the most. It was decided to keep $T = 75$ for the final implementation of the detector. In regard to extensions and rejections the results show a large fraction of the extensions as being unnecessary. However, since data labelled as background is thrown out it is better to waste the extensions (basically increasing the number of false alarms) than to truncate signals. These false alarms do cause some problems in level II as will be discussed in the next chapter. All rejections were done correctly.

Results for the variable gain test were mixed. For the first file, changes in gain had almost no effect on performance. For the second file, the number of false alarms went up considerably. Apparently, the value of loM is too small resulting in a negligible dead zone and a large number of valid zero-crossings. It is clear the detector failed, but it is comforting to see that it failed on the side of caution (increased false alarm

rate). Given that such drastic gain changes are unlikely in practice, and an acceptable performance on other data sets, no further modifications were made to the detector.

In general, the detector will fail by increasing the number of false alarms instead of the number of misses. For the number of misses to increase a large value of loM is necessary. However, since average values below loM go directly into the filtered loM, a high loM is not very likely. The most likely scenario is a too small loM leading to a smaller than necessary dead zone and an increase in false alarm probability.

2.2 Frequency Domain Representation

This section describes the processing done on the output of the endpoint detector. It should be noted that conversion to the frequency domain is not necessary for every application. For the dolphin whistles scientists have used spectrograms to describe and distinguish among whistles. It is therefore logical to transform the output of the endpoint detector to the frequency domain. As seen below, this also allows for further compression of the signal.

The size of the analysis window used has already been set at 512 samples per window. This size was based on requirements imposed by the signal detector and the time resolution of frequency samples. It is entirely possible to select different window sizes for frequency analysis and signal detection but the additional complexity made this approach undesirable. Having the same window size allows detection and conversion to the frequency domain to occur concurrently (on a per window basis) instead of sequentially.

For each analysis window, a power density spectrum estimate is computed. Two different techniques were studied for power spectrum estimation. Among the classical power density spectrum estimators, the periodogram [29] was selected for its simplicity

and low computational cost. The estimate, $\hat{S}_{xx}(\omega)$ is given by the equations below,

$$\hat{S}_{xx}(\omega)_i = \Delta\mathrm{T}\frac{\left|\sum_{n=0}^{N-1} x_i[n] * w[n]e^{-jw\Delta\mathrm{T}n}\right|^2}{\sum_{n=0}^{N-1} w[n]^2} \tag{2.7}$$

$$\hat{S}_{xx}(\omega) = \frac{1}{L}\sum_{i=0}^{L-1} \hat{S}_{xx}(\omega)_i \tag{2.8}$$

In the above equations $x[n]$ is the data to be analyzed, the index n is valid from 0 to $N-1$ with $N = 512$. The function $w[n]$ is a window function. The window is normalized to unit power by the term in the denominator of Equation 2.7. The value of $\Delta\mathrm{T}$ is the time between samples. Multiplying by $\Delta\mathrm{T}$ makes the above estimate, based on a digitized signal, a better estimate of the analog power spectrum density.

To reduce the variance of the periodogram estimate the analysis window can be divided into L subwindows. A power spectrum is then computed for each subwindow and an average taken. This operation is shown in Equation 2.8. Figure 2-9 shows the spectra obtained from the periodogram for 2 different L values. The solid plot is for $L = 1$ and the dashed plot for $L = 4$.

From the more recent, model-based, power density spectrum estimators the Burg technique [20] was chosen. The Burg technique offers higher resolution than other model-based techniques while being computationally fast. On the other hand, the Burg technique can produce biased frequency estimates for sinusoidal spectra. However, this bias is less of a problem when several cycles of the signal are present in the window. Given the frequency of our signals and the window duration, no biasing problems were anticipated.

The basis of model-based power spectrum estimation is to estimate a model (difference equation) that will generate the observed data $x[n]$ when driven by white noise. In particular, auto-regressive (AR) spectrum estimation methods fit a model

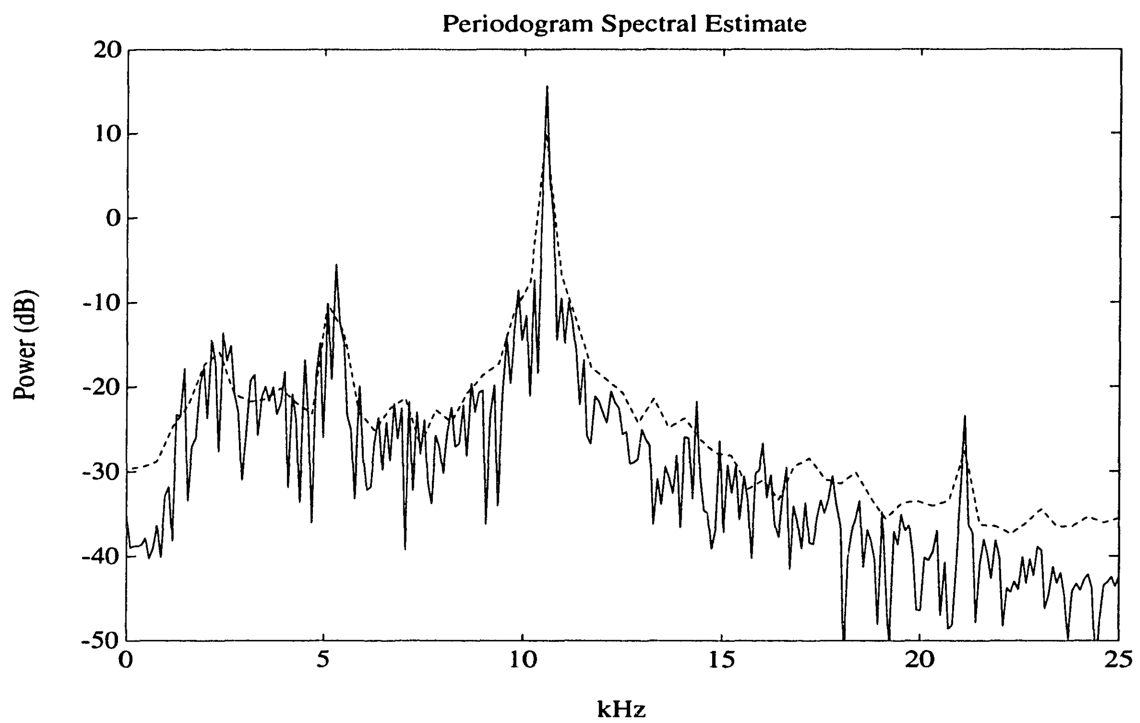


Figure 2-9: Example of periodogram power density spectrum estimate. - Two curves shown, the solid one correspond to $L = 1$. That is, no segmenting of window. Dashed curve corresponds to $L = 4$.

of the form

$$x[n] = -\sum_{j=1}^{M} a_j * x[n-j] + \mathrm{W}[n]$$

where $\mathrm{W}[n]$ is a white noise sequence of variance σ_{W}^2, M is the model order and a_j for $j = 1, 2, ..., M$ are the model parameters. The problem is to determine all a_j and σ_{W}^2. For the purpose of identifying model parameters, M is assumed known. In practice, M is another variable to be set based on the expected number of sinusoidal peaks in the data. Automatic model order selection algorithms exist based on measures such as the Akaike Information Criterion [19].

The Burg technique identifies the model by minimizing the sum of squares of the backwards and forward model errors. The minimization is subject to the constraint that the AR parameters satisfy the Levinson recursion for all model orders from 1 to M. More information regarding Burg's and other spectral estimation techniques can be obtained from [19, 26]. Given the $a_{M,j}$, that is, the a_j elements for an order M model, and σ_{W}^2, the Burg's spectral estimate is

$$\hat{S}_{xx}(\omega)_{\mathrm{Burg}} = \frac{\sigma_{\mathrm{W}}^2 \, \Delta\mathrm{T}}{\left| \sum_{k=0}^{M} a_{M,k} \mathrm{e}^{-j\omega\Delta\mathrm{T}n} \right|^2} \tag{2.9}$$

Figure 2-10 shows the same spectra as that of Figure 2-9 as obtained by the Burg technique. The model order used was 6. Clearly, the Burg technique produces a much cleaner estimate of the power spectrum than the periodogram. On the other hand, for the window size used the computational burden of the Burg technique was nearly twice that of the periodogram. This computational burden is even higher if an automatic model order selection algorithm is added.

Thinking in terms of what the power spectrum was going to be used for, it became clear that using the Burg technique was overkill. From either of the 2 spectral estimates it is fairly simple to estimate the main frequency of the signal. Since this main frequency is what gets preserved in later stages either method is good enough. Sec-

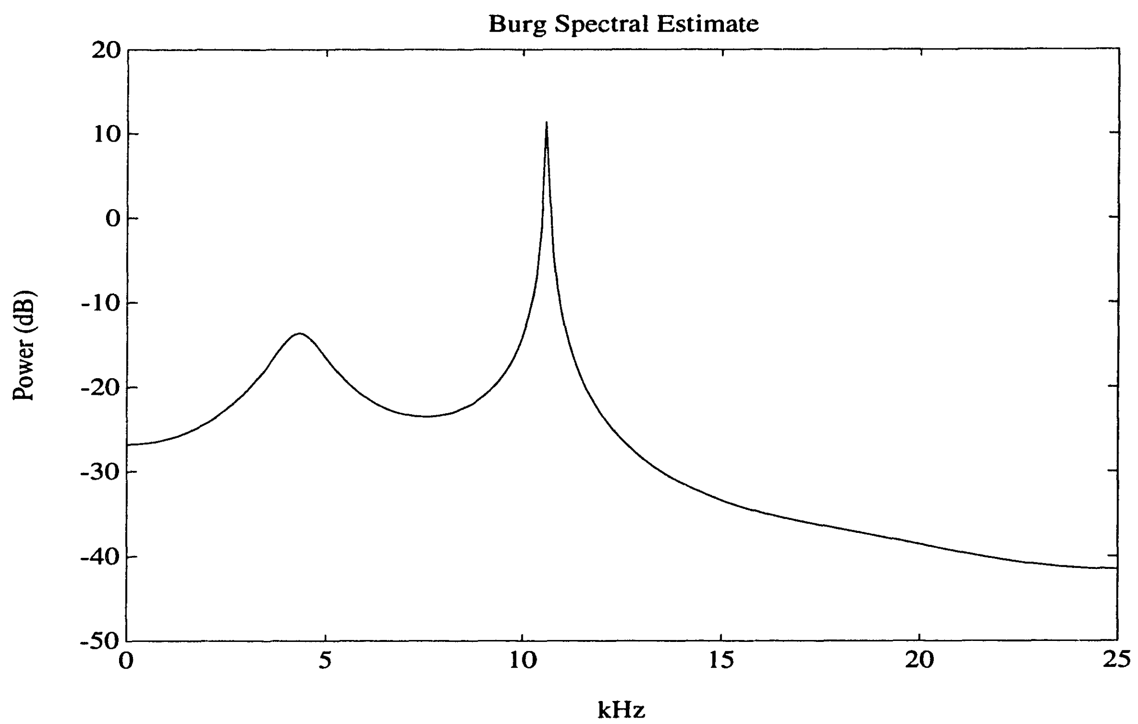


Figure 2-10: Example of Burg's power density spectrum estimate. - The model order used was 6.

ondary peaks are harder to detect. Not only that, but there seems to be disagreement between both methods as to the number and location of secondary peaks. Depending on the model order M, the secondary peaks in the Burg spectral estimate shifted in amplitude and location. This is most likely due to how weak these peaks were, more than 20 dB below the amplitude of the strongest peak.

Given the uncertainties in model order, and the lack of need for the sophistication of the Burg technique, the periodogram (with $L = 1$) was chosen as the power spectrum estimator for the thesis. The value of L was set to 1 based on the intended application of the power spectrum. The variability observed in the amplitudes of the power density spectrum estimate did not prevent identification of the fundamental frequency of the whistle. Thus, the smoothing provided by larger values of L was not necessary. Nevertheless, the code written to implement level I does give the user the option of changing the value of L.

One area in which the Burg technique does come out superior to the periodogram is in the amount of data needed to store the power spectrum. For the Burg spectrum one only needs to store $M + 1$ numbers, the a_j parameters and σ_W^2. On the other hand, the periodogram requires $1 + \frac{N}{2L}$ numbers[5] where N is the window duration and L the number of segments. Although the spectrogram was described in terms of the continuous variable ω, in practice it is computed using the FFT algorithm. An $\frac{N}{L}$-point FFT is used, resulting in $\frac{N}{L}$ equally spaced frequency samples (or bins).

In order to reduce storage requirements, the power spectrum given by the periodogram is quantized prior to storage. Since the spectra to being quantized is in dB's, a simple uniform quantizer is used. Given that the initial data stream was quantized to a maximum of 16 bits (normally just 12), the maximum amplitude possible is 32767. A sinusoid of this amplitude has power equal to 87.30 dB. Therefore, the largest peak in the power spectrum density is 84.29 dB minus $10 * \log_{10}(\Delta f)$. For a sampling rate of 50 kHz and a window size of 512 samples, the largest peak is

[5]After exploiting symmetry.

64.39 dB. However, chances of actually getting a peak that large are negligible since during the digitization process the operator tries to keep the signal away from the limits in order to avoid clipping. On the lower end the only limit is $-\infty$ dB when the input is zero. Practically, there is little to gain in keeping track of very weak signals. A dynamic range from -65 dB to 63 dB was selected for the uniform quantizer.

Having selected the dynamic range for the power spectrum quantizer the next task is to select the number of bits used to represent each quantized number. Using R bits per number, the total number of steps in the quantizer becomes 2^R. The spacing between steps, Δ, is given by

$$\Delta = \frac{\hat{S}_{\max} - \hat{S}_{\min}}{2^R} \tag{2.10}$$

where $\hat{S}_{\max}$ and $\hat{S}_{\min}$ are the upper and lower limits of the dynamic range. Assuming Δ is small enough the probability density function of the quantization error q will be uniform. Under the conditions of a small Δ and zero or negligible overload, Jayant [16] gives the following equation for the signal-to-noise ratio (SNR) of an uniform quantizer:

$$\mathrm{SNR(dB)} = 6.02R - 10\log_{10}(\frac{f_l^2}{12}) \tag{2.11}$$

where f_l is the ratio of the dynamic range to the standard deviation of the input being quantized. In our case, the input is the power spectrum values in dB.

In order to compute the SNR, we need to know the variance of the input. However, this variance is unknown and expected to vary with the specific set of data being analyzed. In addition, the SNR of Equation 2.11 does not take into account that it has been decided not to reproduce weak signals correctly. That is, it is already known that the quantizer will most likely overload at the low end but we have dismissed this as not being a problem. Nevertheless, in order to get an SNR estimate in which to base the selection of R these problems had to be dealt with.

Two files were examined to get an estimate of what the standard deviation of the input signal was like. The standard deviation values found were 13.05 dB and 11.95 dB. Using a value of 10 dB as lower bound for the standard deviation, Equation 2.11 becomes SNR(dB) $= 6.02R - 11.35$. A total of 8 bits were chosen for the quantizer, SNR[6] $= 36.81$. The resulting step size was 0.5 dB. Note that the selection of R was also influenced by implementation considerations. Having each number be stored as a byte greatly simplifies storage and handling of these numbers.

2.3 Whistle Frequency Resolution

The work described so far has been fairly generic with only a few specifics such as the analysis window size and the description of the uniform quantizer. What follows in this section is specific to the application studied in the thesis. This section describes a study of the number of frequency bins required for the dolphin whistles. The approach taken was to determine how accurately a dolphin can reproduce a specific frequency. This determines the minimum bin width required since any system should be at least as good as the animals generating the signal. Another approach would be to use how well dolphins can perceive frequencies to determine the minimum bin width but this was too complex to implement.

The first step required determining what the specific frequency should be. For this, the whistle of an animal named Spray was chosen. Spray's whistle is the first trace shown in Figure 2-11. The whistle has a clear maximum and minimum frequency which were selected for the study. In addition to examining how well Spray could reproduce these frequencies, a second animal was also used. The second animal, called Scotty, had been in contact long enough with Spray to have assimilated Spray's whistle as part of its vocabulary. So, we were able not only to investigate how well an animal reproduces its whistle, but also how well it can copy that of another. Scotty's copy

[6] SNR values will be given in dB.

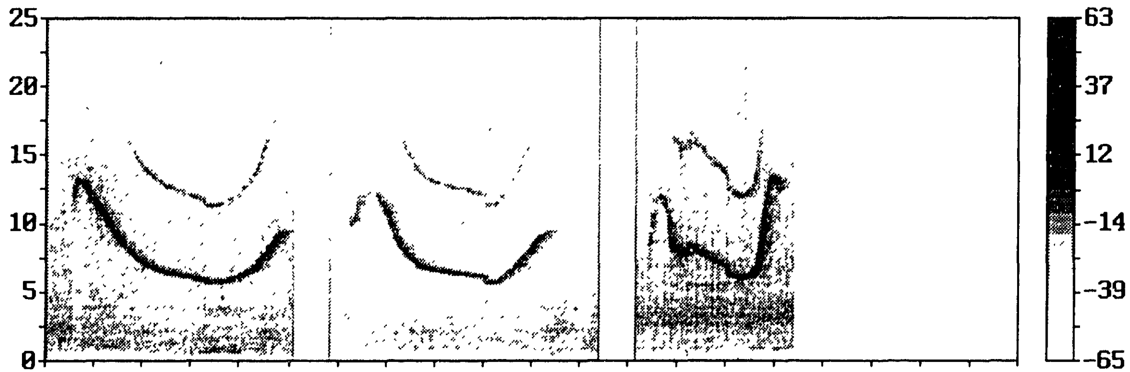


Figure 2-11: Sample spectrogram of dolphin whistles. - Three whistles shown, from left to right, signature whistle of Spray, Scotty's copy of Spray's whistle and Scotty's signature whistle. Frequency is shown on the left (in kHz). The shade of gray indicate power in dB according to the scale on the right.

of Spray's whistle is the second trace shown in Figure 2-11 while the last trace is Scotty's own whistle. Due to ambiguities in the location of the maximum frequency, frequency resolution for Scotty's own whistle (not the copy) was done using only the minimum.

A window of 2048 samples was used in this work (although Figure 2-11 was made with the standard 512-samples analysis window). Given the 50 kHz sampling frequency, the width of each frequency bin was 24.41 Hz. Two different window functions $w[n]$ were used for the analysis. However, the small variations introduced by the windowing functions did not alter the results. The maximum change in frequency estimate observed due to the window function was just about 40 Hz. Nevertheless, only cases in which both window functions agreed are reported in this thesis.

The estimates of average max and min frequency along with the standard deviation and number of samples used is given in Table 2.4. As seen from the table, when the animal is making its own whistle the standard deviation is about 250 Hz. This indicates that a 500 Hz frequency bin is adequate for the dolphin signals. Thus, the 512-samples analysis window is adequate even when the sampling rate is 40.96 kHz.

Whistle Type	Frequency	Size	Mean	Std Dev
Spray	Max	5	12.6 kHz	252.9 Hz
Spray	Min	9	5.75 kHz	230.0 Hz
Scotty's copy	Max	9	12.3 kHz	601.7 Hz
Scotty's copy	Min	9	5.97 kHz	509.6 Hz
Scotty	Min	6	5.98 kHz	255.6 Hz

Table 2.4: Variability of repeated whistle frequencies.

Other interesting results came out of this study. First, it was observed (Table 2.4) that Scotty has better control of the frequency (lower standard deviation) when reproducing its own whistle than when copying another. Other researchers have looked at how well dolphins can reproduce sounds [33], however, no attempt was made in [33] to quantify this ability. Second, the variability of the difference between max and min (relative pitch) was found to be larger than that of the absolute frequencies. So, it seems that absolute frequency is better controlled than relative frequency.

2.4 Additional Ideas

The decision to use an uniform quantizer for the power spectrum values was based on several factors. These included implementation complexity, robustness to a varying input, and compression gain. Nevertheless, several other compressors for the power spectrum signal were studied. This section briefly describes these other compressors. Experiments were done using data from 2 small files containing only whistle cuts.

The first area investigated was the SNR which could be achieved by quantizing the difference between power spectrum values instead of the (absolute) value itself. The performance gain is given by the ratio between quantization noise for the absolute value and the one for the differential value [16]. When the signal being quantized is Gaussian or logarithmic quantization is used, the ratio can be approximated by the

File	Order	Gain	Gain (dB)
dl19	1	2.5197	4.01 dB
	1	2.7970	4.47 dB
	2	2.9181	4.65 dB
dolsqk	1	3.1020	4.92 dB
	1	3.3740	5.28 dB
	2	3.5657	5.52 dB

Table 2.5: Coding gain achieved with quantizing difference signal. - Table shows the coding gain expected when the difference between current value and predicted value is quantized. Up to a second order predictor was examined. Two first order predictors were considered, a simple hold (first) and the optimal (second).

ratio between the variances of the absolute and differential values. Since the signal being quantized is in dB (logarithmic) the approximation was used.

Two ways of computing differences were examined, sequential and parallel. The sequential difference is found between frequency bin i and $i+1$. For the first bin, the last bin from the previous window is used. The parallel difference is computed for each frequency bin by subtracting an entire power spectrum from the one computed for the previous window. It turns out that higher coding gains are obtained when the difference is taken sequentially. Therefore, only the results for sequential differences will be covered.

The results obtained with the 2 files are listed below.

File	Gain	Gain (dB)
dl19	2.5223	4.0180 dB
dolsqk	3.1025	4.9171 dB

For the above results, the difference was computed between the current value and the previous value. A more general implementation of this idea takes the difference between the current value and the output of an nth-order predictor. So far, the prediction has been just the previous value. Optimal first order and second order predictors were evaluated for the two test files. Results are shown in Table 2.5.

Quantizer	SNR	Comments
Uniform - no overload	39.12† dB	$\Delta = 0.50$ dB
	38.36‡ dB	$\Delta = 0.50$ dB
Uniform - p.d.f. optimized	40.34† dB	$\Delta = 0.40$ dB
	40.34‡ dB	$\Delta = 0.37$ dB
Non-uniform	43.82 dB	
Entropy coded	45.70 dB	

Table 2.6: SNR obtained by different quantizers. - Two results shown for the uniform (no overload) quantizers, one for dolsqk file (†) and the other for dl19 file (‡).

According to the table, an average gain of approximately 5 dB can be expected if we choose to quantize the differential signal instead of the absolute signal. This is not even 1 bit. Thus, the use of differential signals was deemed not worth the effort and complexity.

After it was decided the absolute (instead of difference) power spectrum values were to be quantized, a set of quantizers were evaluated. For an 8-bit quantizer, Table 2.6 shows the expected SNR for each quantizer. When the variance of the input signal was required, the one computed for each file was used. This only affects the uniform quantizer entry. For all other quantizers, performance is dependent on the type of probability distribution of the input signal. A Gaussian probability distribution was assumed based on histograms of the input data.

As the table shows, one can gain about 5 dB by using a better quantizer other than the uniform quantizer selected before. However, once again it was decided such a performance increase was not worth it. Combining the gains of both ideas, i.e. quantize the differential signal using an entropy coded quantizer, would provide a 2 bit gain. These 2 bits in turn mean an additional compression factor of 1.33 (6 bits used instead of 8), not enough of a gain for the complexity incurred.

2.5 Reversing Level I Compression

Before ending this chapter it would be good to discuss how one could reverse level I compression and obtain a facsimile of the original signal. It should be obvious that exact reconstruction is not possible and that it is not the goal of this thesis. Instead, reconstruction is a low priority topic to be discussed in the abstract without any experiments performed to quantify the quality of the reconstructed signal.

To obtain the facsimile of the original signal, from now on simply referred to as the original signal, two processes need to be reversed. First, all the silence that was thrown out must be put back in. Second, from the power spectrum of the signal we have to go back to the time domain.

The first process is the easiest to reverse since we do not really care about listening to background noise. All we need to do is keep track of the duration of the "silence" and just play back nothing for the same amount of time. However, some listeners may find disconcerting to alternate between whistle and pure silence. In this case, we can reconstruct some background to play back either from a random source, or from the average background spectrum[7].

The second process is harder to reverse. Nawab *et. al.* [28] have looked into the problem of exact reconstruction of signals from the power density spectrum. A set of conditions and algorithms for exact reconstruction is given in [28]. Among the conditions listed is at least a 50 % overlap of the analysis windows and that each power spectrum be computed with twice the number of frequency samples than time samples. Neither condition is satisfied by the current implementation of the level I compressor. It should be noted however, that in terms of the number of samples required for storage of the signal one would be better off keeping track of magnitude and phase for non-overlapping windows than using power spectrums. That is, assuming exact reconstruction is desired.

---

[7]An average power density spectrum computed from the background windows. Discussed in more detail in the next chapter.

In order to use the algorithms of [28] the power spectrums stored during level I compression have to be interpolated in 2 directions. First, they have to be interpolated in frequency to get twice the number of frequency samples. This will allow computation of an autocorrelation function with little (1 sample) aliasing. Then, the power spectrums have to be interpolated in time, again by a factor of 2. This will simulate the 50 % overlap required by the reconstruction algorithm. Once the 2 interpolations have been carried out then reconstruction can proceed using any of the techniques given in [28].

Another way of reconstruction the original signal from the power spectrum is to simply assume a random or constant phase and compute the inverse Fourier transform of each window independently. To avoid the high frequencies introduced by the discontinuities between windows, the signal needs to be low-pass filtered. This approach takes advantage of the lack of sensitivity to phase of the human ear to produce a signal which may sound the same as the original but most likely has a completely different phase.

Chapter 3

Level II Compression

Level II compression consists of taking the quantized spectrograms produced for each whistle segment by level I and converting them (for the most part) into a single frequency-versus-time trace. Harmonics, amplitude information, and spectral content of most frequencies are removed. To obtain the desired trace, first the peaks of each power density spectrum are identified. Then a single peak is chosen for each spectrum, forming a cohesive smooth trace. Since a single frequency index (out of 257) is preserved from each power density spectrum, the amount of compression achieved in level II is an additional factor of 257 beyond that achieved in level I.

The peak detection will be covered in the first section. The algorithm for connecting the peaks is described in the second section. A discussion on how to reverse the level II compression is also included in this chapter.

3.1 Peak Detection

Any point surrounded by two points of lower amplitude qualifies as a peak. Because the initial and final samples of the power spectrum only have points on one side, a peak can never occur at those locations. Given a noisy power density spectrum, such as that shown in Figure 3-1, one can see that each spectrum is going to have a lot of

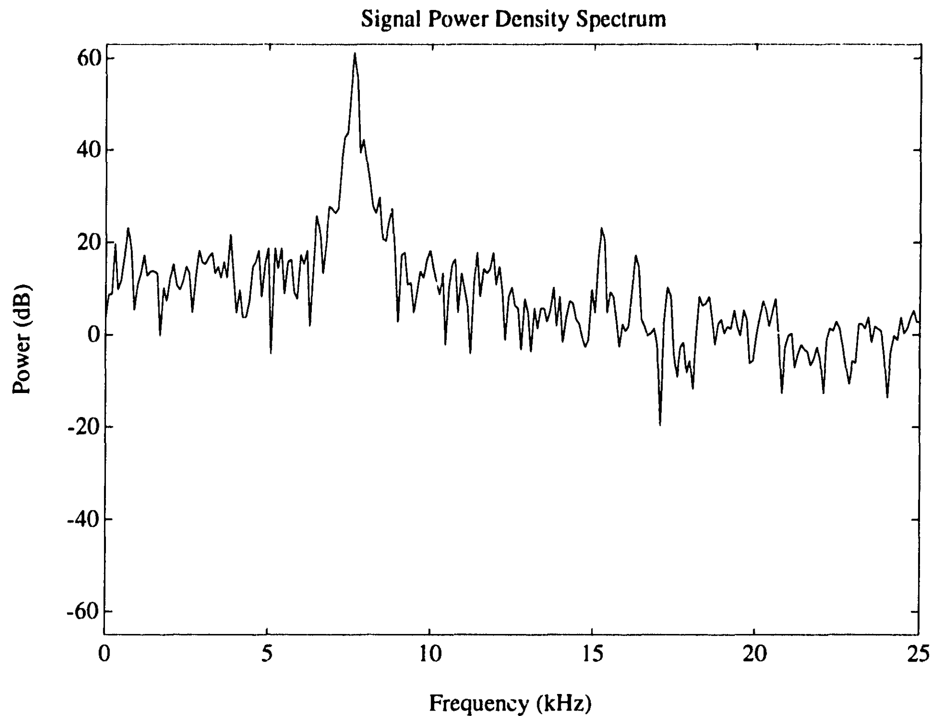


Figure 3-1: Example of power density spectrum for signal window.

peaks. Because of this, it was decided to keep a list of just the largest 5 peaks, sorted by amplitude. The amplitude of the peak is also stored in the list.

Note that a peak may also be flat at the top. That is, a set of equal-valued samples with lower-value samples at the edges. For these flat peaks, the last frequency (highest) in the set of equal-valued samples is chosen as the frequency of the peak. Chances of actually getting a wide flat peak are very small.

Before the spectrum is analyzed for peaks the background noise is subtracted. The estimate of the background noise power spectrum is an additional output of the level I compressor. During periods of "silence", the power density spectrum is computed every 20 windows and then low-pass filtered. The output of this filter is used as the estimate of the background noise power density spectrum. Every time a signal segment is stored, the current background spectrum is stored with it. Figure 3-2

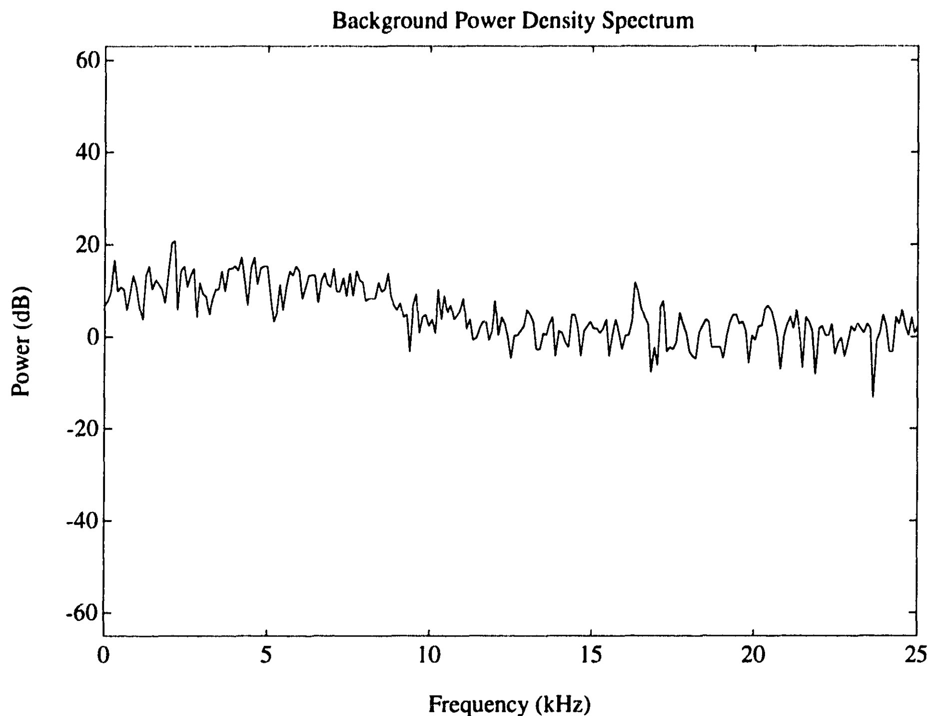


Figure 3-2: Example of background power density spectrum.

shows the background spectrum corresponding to the signal segment where Figure 3-1 was obtained. Once the average background is removed, the spectrum in Figure 3-3 is obtained. Frequencies where the background has more power than the window spectrum are given the lowest possible value, -65 dB. These points are not shown (i.e. they are missing) in Figure 3-3 and lines are drawn connecting only those points with power larger than -65 dB.

Spectra like the one shown in Figure 3-3 is then examined for peaks. In Figure 3-4 a signal segment from tape 87203 is shown. The peaks obtained from this segment are shown in Figure 3-5. All peaks are marked as circles except for the largest being marked as pluses (+).

From Figure 3-5 one can start observing some of the problems the trace estimator will have to overcome. First, although a large percentage of the time the largest-

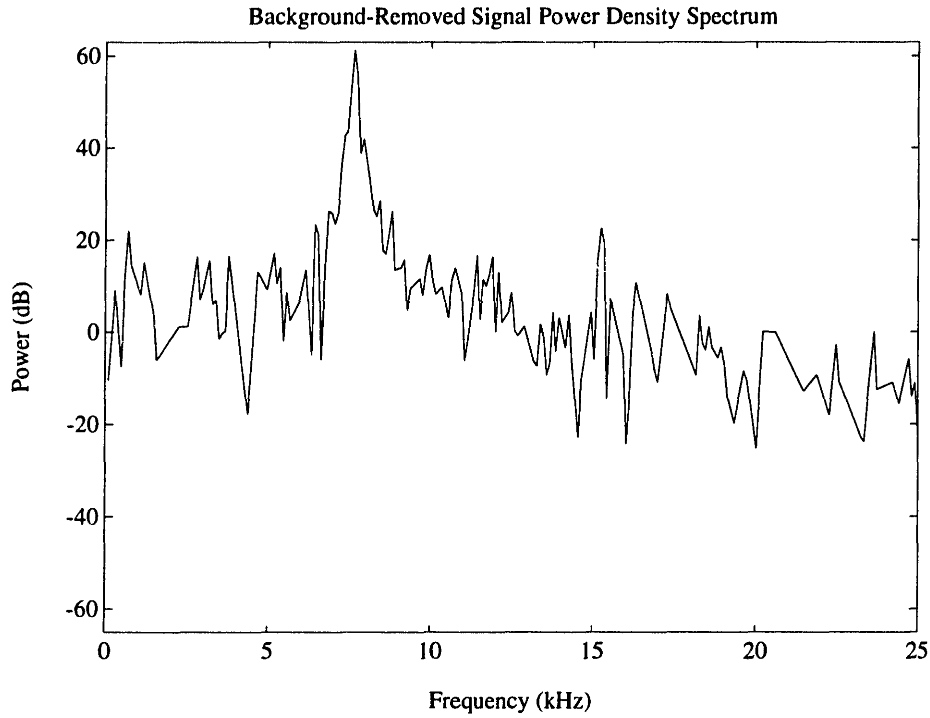


Figure 3-3: Power density spectrum for signal window after removing background.

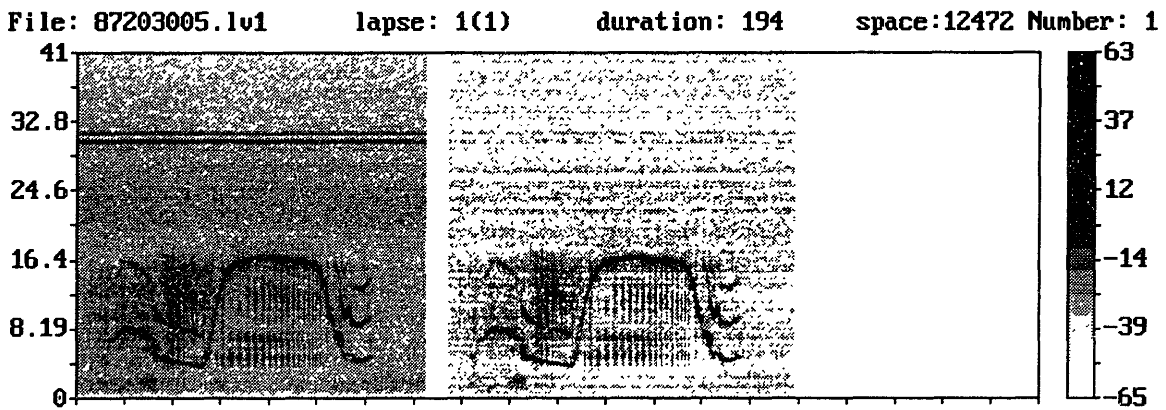


Figure 3-4: Signal segment spectrogram - Example used to illustrate peak detection algorithm. Same signal shown twice, original on the left and without background on the right.

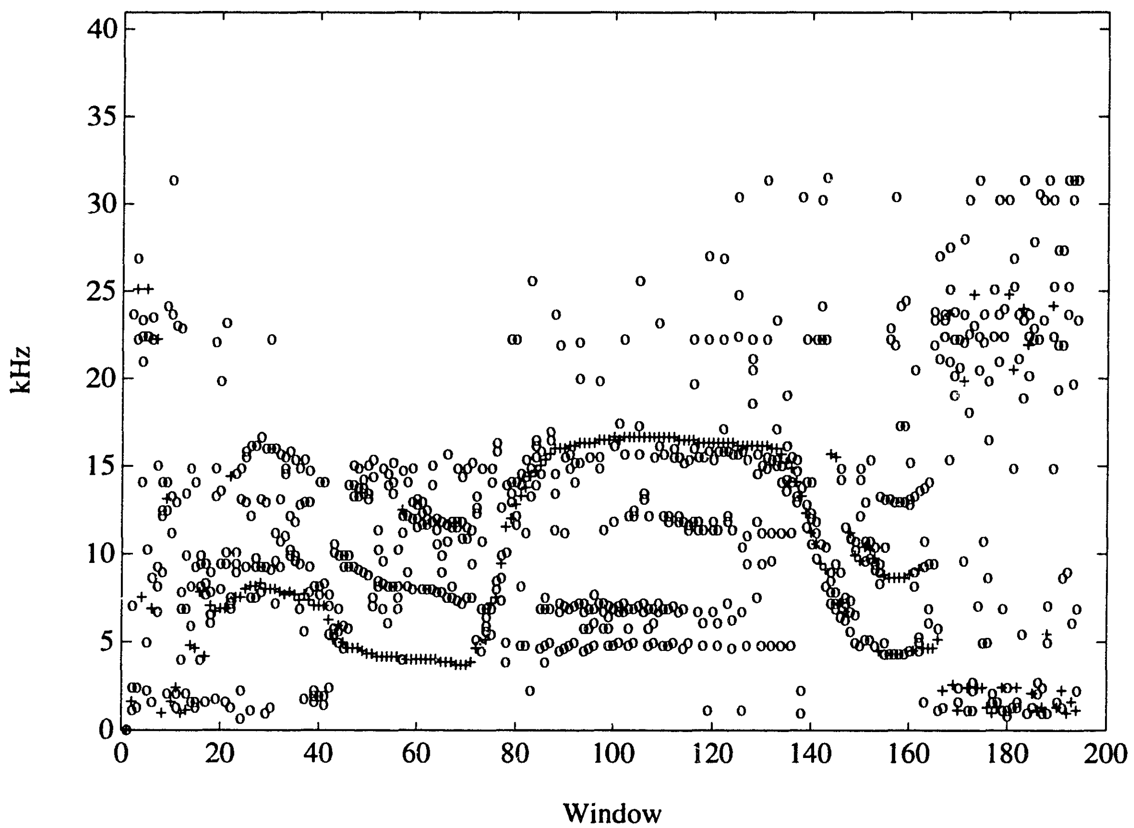


Figure 3-5: Example of peak selection algorithm output. - Five peaks are shown for each window, the largest peak is marked with a '+' and the other four with circles.

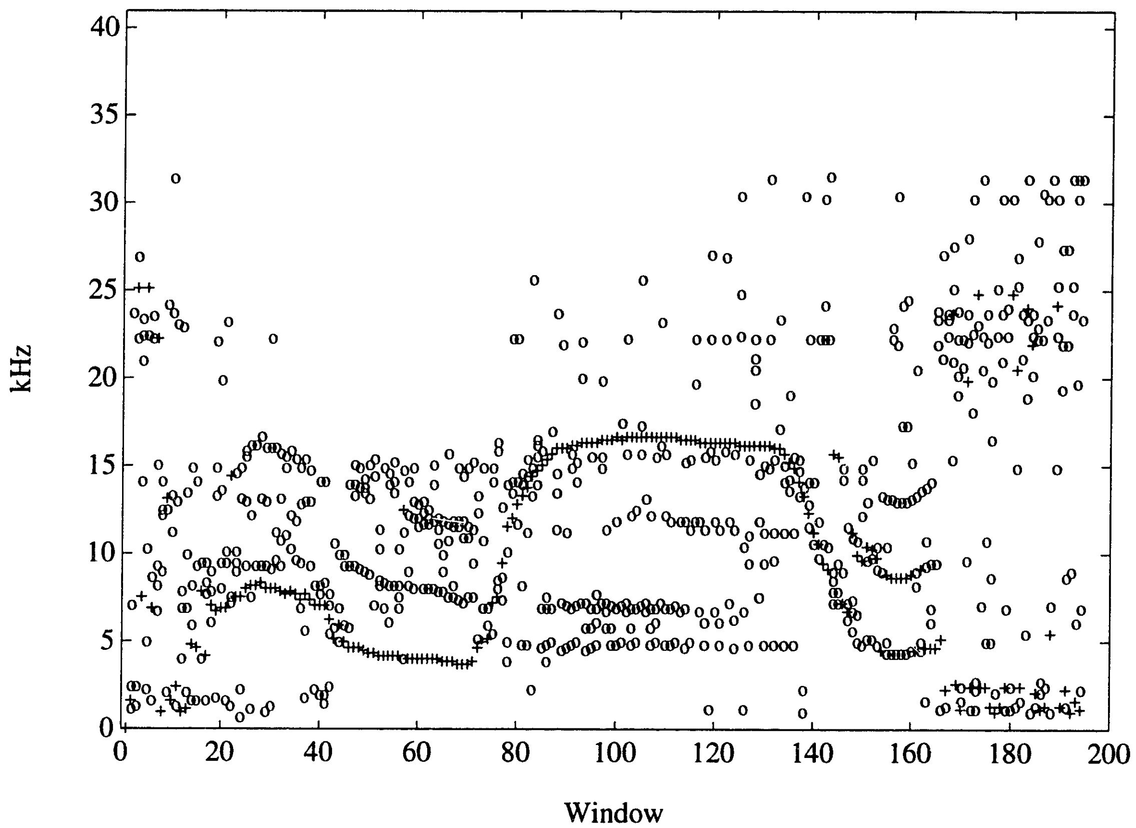


Figure 3-6: Example of peak selection algorithm output after cleanup. - Surviving peaks are shown for each window. The largest peak is marked with a '+'.

amplitude peak belongs to the desired trace a simple use-the-largest-peak strategy will not work. There are at least two instances where the simple strategy would fail. One is when a harmonic happens to be stronger than the fundamental (see the figure around window 160). Another is in windows that contain no signal but were accepted by the signal detector (at start and end of figure).

Second, some of the selected peaks are really just noise (low frequency peaks around window 120). These peaks are some times the result of clicks which boost the amplitude of ordinary noise peaks to large values. Third, it is a common occurrence for peaks to cluster together in frequency. That is, to have several peaks fairly close in frequency value. The use of the periodogram as power density spectrum estimator contributes to this clustering because it is a noisy estimator.

To reduce the clustering problem a cleanup routine was developed for the peak list. Basically, the cleanup consisted of computing the bandwidth of each peak and removing peaks that were actually part of a larger peak. The bandwidth was determined by looking for a point[1] where the spectrum stops going down in amplitude. That is, starting at the peak, one moves away from it while looking for the "bottom" of the peak. Because the periodogram estimate is noisy, the spectrum was not considered to have stopped going down until two consecutive higher-amplitude samples were found or the edge of the spectrum was reached.

The set of clean peaks corresponding to those peaks in Figure 3-5 is shown in Figure 3-6. There does not seem to a big difference between the original peak list and the cleaned one. The tracing algorithm performance was also not affected by whether the peak list was cleaned or not. Thus, the cleanup was removed from the final implementation.

[1]Two points are actually searched for, one to the left and one to the right.

3.2 Estimating the Frequency-vs.-Time Trace

Initial work on a tracing algorithm was focused on fixing up the peak list and then simply linking together the strongest peak of each window. After all, most of the time the strongest peak was the correct frequency to keep so it made sense to start there. The problem with harmonics being selected instead of the fundamental was quickly resolved by adding a harmonic check to the peak selection. The problem of deciding if a window actually had signal in it or if it was a false alarm from the previous stage was harder to solve. In fact, after several attempts it was decided that a single window did not contain enough information to make the decision.

Attempts to carry over information across windows were limited initially to amplitude information. By keeping track of a global (file or session) maximum and a local (signal segment) maximum, a threshold was established for the minimum amplitude of a signal peak. Peaks with lower amplitude were classified as silence. Setting of this threshold was ad hoc and in many instances the resulting trace failed in places where the peak barely missed being over the threshold. In addition, the local maximum was compared against the global maximum to decide if the entire signal segment should be thrown out.

Windows which are identified as false alarms have the peak set at zero frequency. That is, all those windows that do not belong in the trace should have the peak set to zero Hz. By making all the peaks in all the windows of a segment be zero the segment is thrown out. Instead of storing an empty trace (all zeroes), the time lapse between segments is increased so that whistles maintain the correct location in time. Since no real peaks can exist at 0 Hz, the use of zero as marker is unambiguous.

Going back to the tracing problem, the next step was to carry over frequency information as well as amplitude information when finding the trace. This led to the development of an algorithm based on dynamic programming [22] and similar to that used in dynamic time warping [15]. The trace will now be found by optimizing a cost

function. In addition, this cost function can also be used to represent the quality of the trace found.

Specifically, dynamic programming deals with the optimization of cost functions of the form

$$C = \sum_{k=0}^{N} c(\underline{x}(k), u(k), k)$$

where $\underline{x}(0)$ is the known initial value of the state vector $\underline{x}$, and there exists a system equation $\underline{x}(k+1) = g(\underline{x}(k), u(k), k)$ relating the past values of the state vector and the inputs u, to the next state vector. In addition, there exists constraints on the possible values $\underline{x}$ and u may assume. By optimizing the cost C one tries to find the input sequence $u(k)$ which results in a minimal cost.

There are backwards and forwards versions of the dynamic programming algorithm. The backwards version, as the name implies, begins at the allowable final states and works backwards towards a set of possible initial states. A fixed initial state is not required for optimization. When a fixed initial state is known, the forwards algorithm is better suited to the problem. In the forward algorithm, at a given $\underline{x}$ and k, all possible inputs $u(k-1)$ are used to find the optimal decision and minimum cost.

The tracing problem can be posed in the terms of the typical dynamic programming problem but first some quantities must be defined. Namely, let

- $t(\cdot)$ - trace function. The value $t(k)$ is the frequency (or frequency index) selected from window k.

- $r(\cdot)$ - rank function. Recall that the 5 peaks selected from each window are ranked according to amplitude. The value $r(k)$ refers to the rank of the peak chosen from window k. The rank is a number between 0 and 4, with 0 corresponding to largest peak.

- $p(\cdot,\cdot)$ - peak function. The value of $p(k, r(k))$ is the frequency (or frequency index) corresponding to the peak indicated by $r(k)$ for window k.

Note that, although matrix notation is used below, the terms "trace" and "rank" will never be applied to a matrix in this thesis. The only use for these terms is the one given above.

Using the terms defined above, let the state variable be a 2-element vector, $\underline{x}(k) = [x_1\ x_2]^{\mathrm{T}}$, with $x_1 = t(k-1)$ and $x_2 = t(k-2)$. That is, the state variable is simply the last 2 frequencies (or frequency indexes) in the trace. The input $u(k)$ is given by $r(k)$, the rank of the peak to be added into the trace. Both the inputs and the state variable have a restricted domain as required by the dynamic programming algorithm. The evolution of the state variable $\underline{x}$ is given by

$$\begin{aligned} \underline{x}(k+1) &= g(\underline{x}(k), r(k), k) \\ &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \underline{x}(k) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} p(k, r(k)) \end{aligned} \tag{3.1}$$

Different cost functions were tried during the development of the tracing algorithm with different performance obtained for different traces. However, the one that seemed to work best most of the time was the following,

$$\begin{aligned} c(\underline{x}(k), r(k), k) &= |p(k, r(k)) - 2x_1 + x_2| * (1 + \frac{r(k)}{5}) \\ &= |p(k, r(k)) - 2t(k-1) + t(k-2)| * (1 + \frac{r(k)}{5}) \end{aligned} \tag{3.2}$$

which is simply a first-order prediction error on the frequency (or frequency index), penalized by how far one had to go down the peak list. The largest peak incurs no penalty, $r(k) = 0$, while the smallest gets penalized by a 1.8 factor.

Having established the mathematical foundation of the tracing problem we can now discuss the algorithm used to find the trace. The implementation of the dynamic programming algorithm for obtaining the optimal trace, i.e. the trace with minimum cost, turned out to be non-trivial. A simpler, but suboptimal, algorithm was used instead to find the trace. The reasons why a suboptimal algorithm was used will

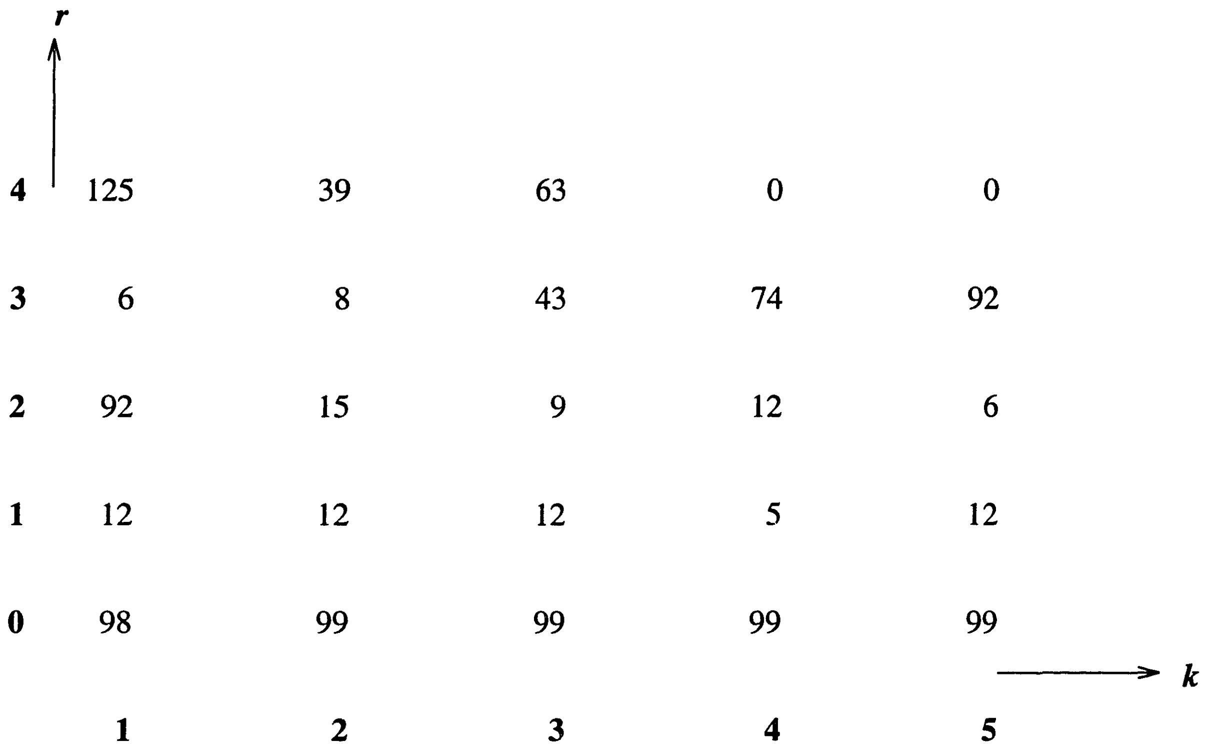


Figure 3-7: Example of grid used in finding frequency vs. time trace.

be discussed later, after some of the unique problems associated with finding the trace are covered. The suboptimal algorithm will now be described with the help of Figure 3-7.

Figure 3-7 shows a grid of points each corresponding to a peak. In the figure, the x axis represents the window number k and the y axis the peak rank r. The goal is to find a trace which connects a single dot from each column and has a low cost C. The value given with each peak is the frequency index corresponding to the peak. A frequency value can easily be obtained from this index. Just for reference, a trace connecting the largest peak of each window would show up as a horizontal line connecting the bottom row in Figure 3-7.

For the moment, assume that the optimal trace starts at the largest peak of the first window, $r(1) = 0$. From that position 5 traces are started, one to each of the

peaks in the second window. The cost of each trace is initialized using a zero-order predictor and no penalty. After initialization we then move to the next window. For each peak in window k a total of (at most) 5 traces are evaluated, one coming from each peak in window $k-1$. The trace with lower total cost is kept. This way, we always keep track of just 5 traces, one trace for each peak in the window. When we get to the last window, the best trace out of the 5 is selected.

The basic algorithm works as described in the previous paragraph. However, the window where the trace starts and ends is not always the first and last windows in the signal segment. Remember that, due to false alarms, one can expect to have windows with no signal content in the segment. Trying to find a trace through these background windows is useless and most likely would introduce errors in the trace from which the algorithm will not recover. So, the solution adopted was to keep track of the incremental cost (cost incurred in going from window $k-1$ to k) and peak amplitudes to terminate the trace and avoid trying to find a trace over these background windows.

The trace is terminated provided the following two conditions both occur. First, the amplitude of the largest peak in the window has to be less than one tenth of the amplitude of the peak where the trace started. Second, the smallest incremental cost has to be larger than twice the average total cost. This average total cost is given by the smallest total cost divided by the number of peaks in the trace being estimated. The first condition checks for low energy windows and the second for when peak location is so noisy that no good trace can be found. Random low amplitude peaks have been observed to be characteristic of background windows in the signal segment.

Once a trace is terminated a subsegment is formed with those windows for which a trace has not yet been found. To start a trace, the subsegment (or full segment when no subsegments have been formed yet) is searched for the location of the largest amplitude peak. The trace starts at this location. This decision basically states that

the strongest peak in the subsegment should be in the trace. In general this is true but not always. What happens when this assumption fails will be discussed later. Once the starting point is selected the tracing starts first to the right (forward in k) and then to the left (backwards in k) until the end of the subsegment is reached or the decision to terminate the trace prematurely is made. The algorithm is the same backwards and forwards, the x axis is manipulated so that no changes are required to the algorithm.

As mentioned above, the algorithm described does not result in a path of minimal cost. What the algorithm does, however, is to find a reasonable trace with little complexity. Let us now illustrate the algorithm with an example. At the same time it will be shown that the trace found does not minimize the cost C. Consider the peak indexes shown in Figure 3-8. As in Figure 3-7 the x axis correspond to window number and the y axis to peak rank. In addition to the peak frequency index, the figure also shows the 5 traces of each window, the incremental cost of each trace element next to the line connecting the peaks and the cumulative cost of the trace in the upper right hand corner of each index. The cost was computed based on the prediction error using frequency indexes instead of frequency values and with no penalty factor for the sake of simplicity.

As seen in Figure 3-8 the trace ending at index 95 in the fifth window found by the algorithm has cost equal to 16. This trace, shown with darker lines, is made up of the indexes {99, 99, 93, 91, and 95}. However, the trace {99, 99, 93, 94, and 95} has a cost of only 13. When finding the trace ending at index 94 in window 4 the trace coming from index 93 in window 3 has lower incremental cost than the one coming from index 101 but based on total cost the index 101 was better. The optimal path is lost at this point. To address this problem of optimality, a trace editor was coded so that any trace could be manually fixed.

Let us now briefly go over the reasons why the optimal algorithm was not implemented. The main reason was one of complexity. For the optimal algorithm it would

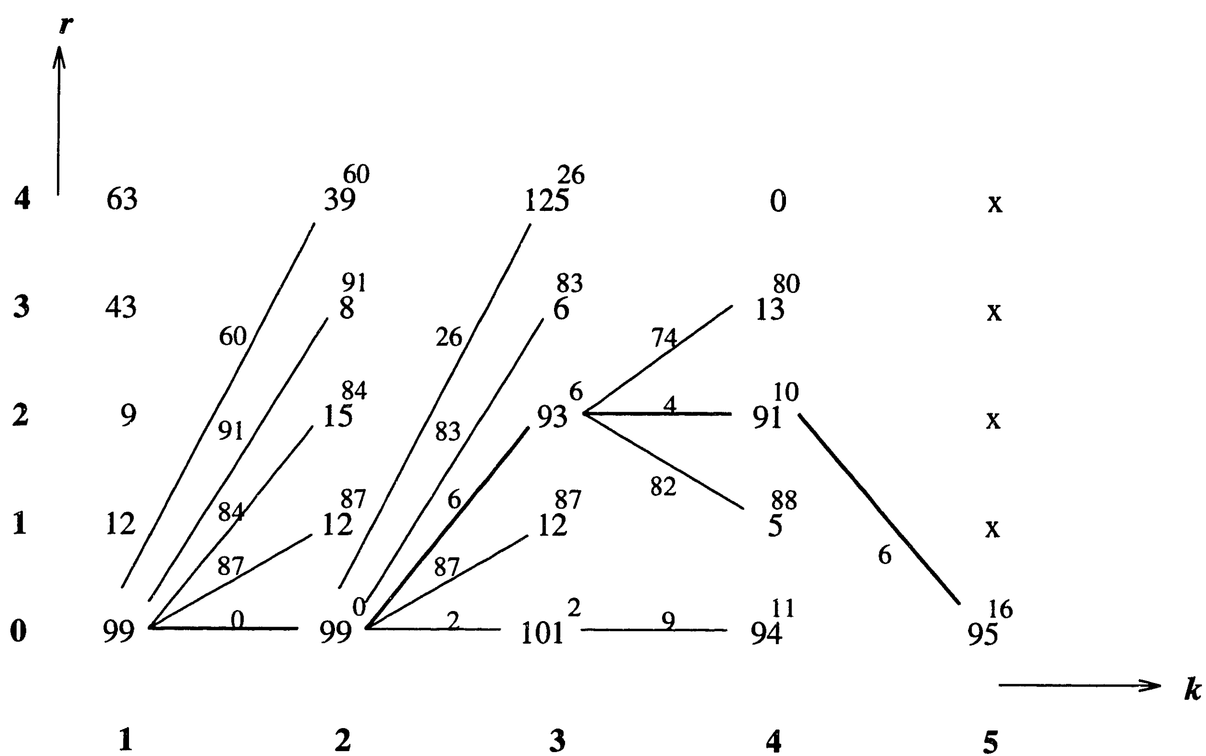


Figure 3-8: Tracing algorithm example.

have been required to keep track of 25 traces at any given time. Twenty-five is the number of possible states at the start of the algorithm and given the uncertainty on which one is correct, all must be tried. This number is dependent on the number of peaks selected from each window (5) and the predictor order used in the cost function. Keeping the cost function fixed, a reduction of the number of peaks selected from each window would reduce the number of traces that must be tracked. However, reducing the number of peaks can result in the peak belonging to the optimal trace not getting into the peak list. On the other hand, increasing the number of peaks selected from each window would make certain that the peak belonging to the optimal trace is in the peak list, but also it will increase complexity and add noise peaks to the list.

Another reason why the optimal algorithm was not implemented was the problem with background windows in the signal segment. The dynamic programming algorithm can not be easily modified to include the creation of subsegments.

It is difficult to quantize how suboptimal the simple algorithm is. In the example given the path found by the algorithm is not that much worst than the optimal. However, as work progresses and more incorrect decisions are made, the cost is expected to go up. Eventually, the trace will be terminated and a new one started in a different subsegment. Thus, severe errors are self-terminating. Figures 3-9 and 3-10 are examples of the results obtained with the simple algorithm. The figures show the trace on the left and the background-removed spectrogram on the right. The few errors seen in Figure 3-10 (e.g. the linear tail at end of trace) can be easily fixed manually using the trace editor.

3.3 Reversing Level II Compression

To some extent, it is possible to take the information preserved in a level II file and reconstruct a level I file from it. This section describes a proposed approach for this

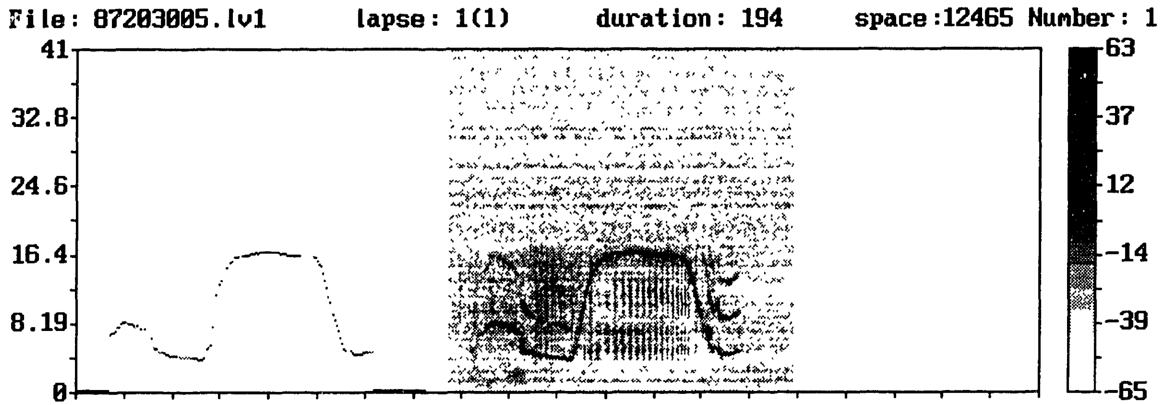


Figure 3-9: Tracing algorithm example (1) using real data.

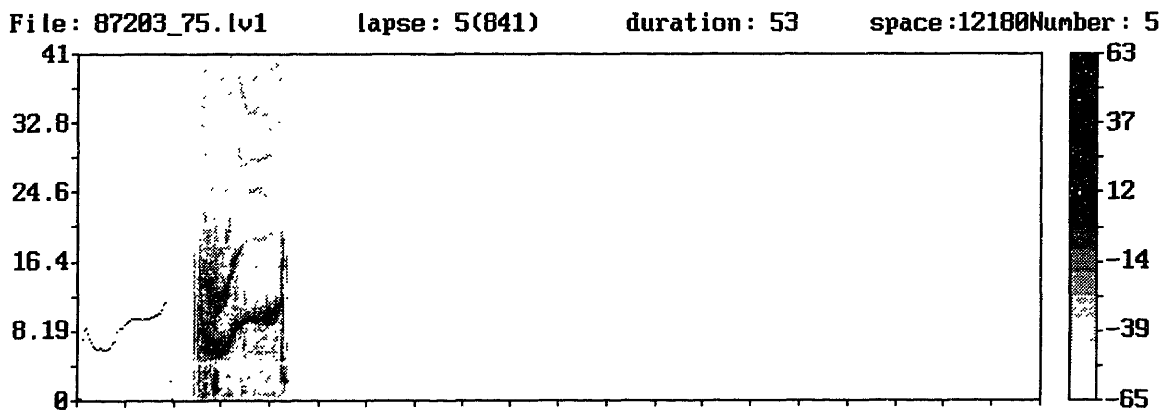


Figure 3-10: Tracing algorithm example (2) using real data.

reconstruction. As was the case when reverting level I compression, the reconstruction will not be exact.

Reversing level II compression requires the creation of a power density spectrum for each window in the signal segment, a background noise spectrum, and some additional bookkeeping such as sampling rate and spacing between signal segments. The noise spectrum and bookkeeping information is saved unchanged into the level II file as it was in the level I file. Thus, "reconstructing" them is not a problem.

Reconstruction of each power density spectrum is a lot harder since the only information maintained (so far) from a signal window in the level II file is the main peak and its amplitude. One can use the background noise spectrum for information on frequencies other than where the peak occurs. However, it was decided instead to preserve some of the non-peak information in the power density spectrum by using a KL (Karhunen-Loeve) transform. The use of the KL transform should provide a better reconstruction than that which could be obtained by just repeating the same background noise spectrum.

Along with the peak location and amplitude, the values of 3 KL coefficients are stored for each window in the level II file. Experiments with the KL transform show that 3 coefficients is enough to capture more than 50 % of the variability in the spectrum. When reversing the compression, the KL coefficients are use to reconstruct a baseline spectrum to which the peak information gets added. The combination is then saved as the power density spectrum for the window in the level I file. Assuming that the 3 coefficients are stored using floating-point precision (4 bytes per coefficient) then the compression factor of level II is reduced from 257 to about 20. Quantization of the KL coefficients can help improve the compression factor.

3.4 Improved Endpoint Detection

As discussed before, the animal detector used in level I is intentionally biased towards false alarms. This better-safe-than-sorry approach was determined as the best way to deal with uncertainties in the recording environment of our signal. Unfortunately, these false alarms now have to be addressed by the tracing algorithm. Failure to do so would result in invalid traces.

The tracing algorithm described above attempts to deal with the problem by keeping track of the cumulative cost. When the cost gets too large the trace is terminated. In theory, the truncation of the trace results in isolation of false alarm segments in the whistle. Once the false alarm segments have been isolated, then a simple power or duration check would serve to reject the segment.

In practice, the identification and rejection of false alarms by the tracing algorithm does not perform as well. If the entire whistle is a false alarm, power thresholds are useless.[2] Also, on many occasions the cost does not jump immediately at the end of the whistle. It may be possible for a few "silence" (i.e. false alarms) peaks to get appended to the whistle simply because they happened to occur at a convenient place.

For the above reasons, methods of refining the selection made by the level I animal detector were explored. In contrast to the level I detector, the detectors explored here are all frequency-domain based. Also, a lower emphasis on computational complexity and speed is given because we are dealing with a lower volume of data.

The section explores two main ideas based on slightly different approaches to the problem. First, by considering the animal detection problem the same as identifying voiced vs. unvoiced segments of speech, the application of cepstral analysis to the dolphin whistles was made. Second, by thinking of the problem as a statistical classification problem based on a set of observations, the application of discriminant

[2] A threshold on the total *cost* may be useful here.

analysis and clustering was studied. Another approach, that of considering the problem a combination of parameter estimation and hypothesis testing (like sonar) was considered briefly. However, the uncertainty in characterizing noise sources as well as the computational burden of this last approach argued against this method.

3.4.1 Discriminant Analysis and Clustering

Given that the signal segment contains a small number of windows and that the power spectra of these windows have already been computed, we could afford to consider new measures for endpoint detection. The following is a list of the measures considered,

1. power - Adding the power at each frequency after removing the background.

2. max power band - After removing background, spectrum is divided into 4 equal size bands. Power is then computed for each band and the number of the band with maximum power is recorded.

3. amplitude of largest peak - Again, measure taken after background is removed.

4. spectral flatness measure - The spectral flatness measure is defined as the ratio of the arithmetic average to the geometric average of a signal. Assuming that the signal for which the spectral flatness measure is going to be computed is $x[n]$ for $n = 1, 2, ..., N$, then the spectral flatness measure is given by,

$$\text{sfm} = \frac{\frac{1}{N}\sum_{n=1}^{N} x[n]}{\sqrt[N]{\prod_{n=1}^{N} x[n]}}$$

   The background is not removed prior to computation of the measure. Two different ways of computing this measure were used, one with the original power spectrum and the second with the power spectrum in dB units.

The discriminant analysis, called Fisher's method and discussed in Chapter 2, was tried first. The idea is to find a linear combination using any of the above measures that can serve as discriminant between the signal and background populations. Results with this technique were marginal. Acceptable false alarm and miss probabilities were possible with the data set used in the analysis but results did not hold when new data was considered.

The clustering work will now be considered. With clustering, all the values of each measure are divided into 2 clusters. Hopefully, each cluster will correspond to each of the populations expected. Since an a priori threshold is not required for making a decision, this technique should continue to perform well when new data is introduced. Unfortunately, a better signal detector than that of Chapter 2 could not be produced. The clustering algorithm resulted in a detector with a large probability of miss. Attempts with multidimensional clustering and boolean combinations of the outputs of different detectors did not improve detection enough to justify the expense of the clustering algorithm.

3.4.2 Cepstral Analysis

Cepstral or homomorphic analysis provides a way in speech processing for the classification of speech segments as either voiced or unvoiced [32]. This section applies cepstral analysis to the dolphin signals to see if a similar discrimination is possible.

The complex cepstrum $\hat{x}[n]$ of a discrete time signal $x[n]$ is defined by the following set of equations

$$X(e^{j\omega}) = \sum_{n=0}^{N-1} x[n]e^{-j\omega n} \tag{3.3}$$

$$\hat{X}(e^{j\omega}) = \log[X(e^{j\omega})] \tag{3.4}$$

$$\hat{x}[n] = \frac{1}{2\pi}\int_{-\pi}^{\pi} \hat{X}(e^{j\omega})e^{j\omega n}\, d\omega \tag{3.5}$$

Because of the problems associated with dealing with the logarithm of a complex

signal the cepstrum, $c[n]$, was introduced. It is defined by the equation,

$$c[n] = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log |X(e^{j\omega})| e^{j\omega n} \, d\omega \tag{3.6}$$

By using only the magnitude of the power spectrum, use of the complex logarithm is avoided. This makes the cepstrum easier to compute than the complex cepstrum. Also, note that the cepstrum is not computed here as given by Equation 3.6. Instead, the cepstrum is approximated using the discrete Fourier and inverse Fourier transforms.

As mentioned before, level II only has access to the magnitude of the power spectrum. This means that one can only compute the cepstrum. This fact does not limit us in any way that we are interested in, since the cepstrum shares many of the properties of the complex cepstrum. In particular, the cepstrum can be used for voiced/unvoiced decisions.

For this study a small sample consisting of just 3 animals was used. The animals were identified as FB 35, FB 62 and FB 153. Only 3 manually-selected windows were used from each animal. One window represented silence (i.e. animal not making noise), another whistle and a third window represented whistle plus click. A window affected by a click was selected because clicks are among the most common secondary sources of sound in the tapes and any improved detection scheme should know how to deal with them.

Figures 3-11 through 3-13 show the power spectrum and computed cepstra for each animal. The left column of each figure shows the magnitude of the power spectrum used to compute the cepstrum on the right column. Windows are ordered silence, whistle, and whistle plus click starting at the top of each figure. An encouraging sign are the peaks which appear in the cepstrum when only the whistle is present as shown in the middle plots of the figures.

Ignoring peaks at cepstral indices 0 and 1, the list in Table 3.1 gives the magnitude of each cepstral peak for each window and animal. It should be clear from the table

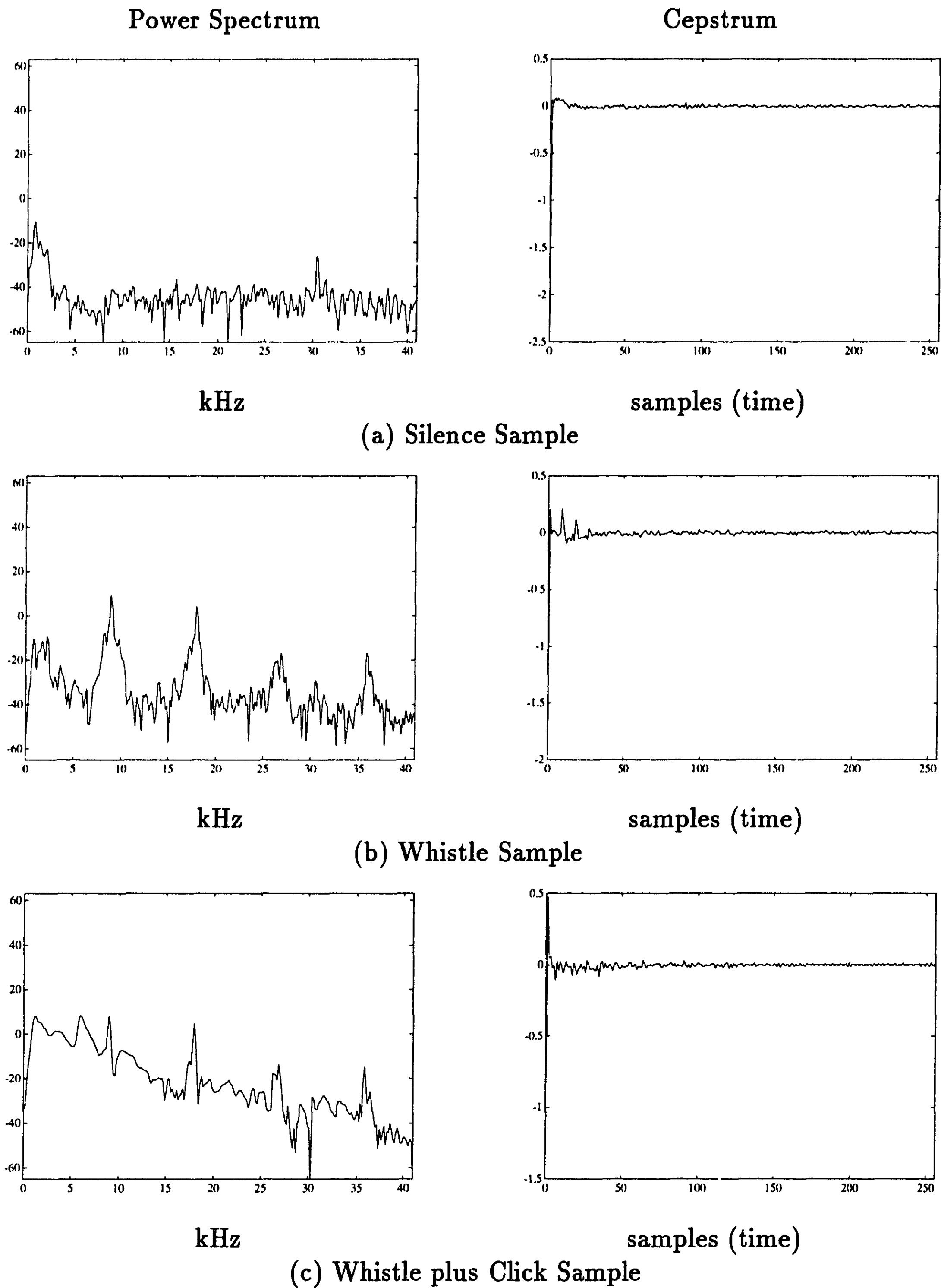


Figure 3-11: Sample spectra and cepstra for FB 35. Samples are 12.2μs apart. Only the (x) positive side shown in each plot.

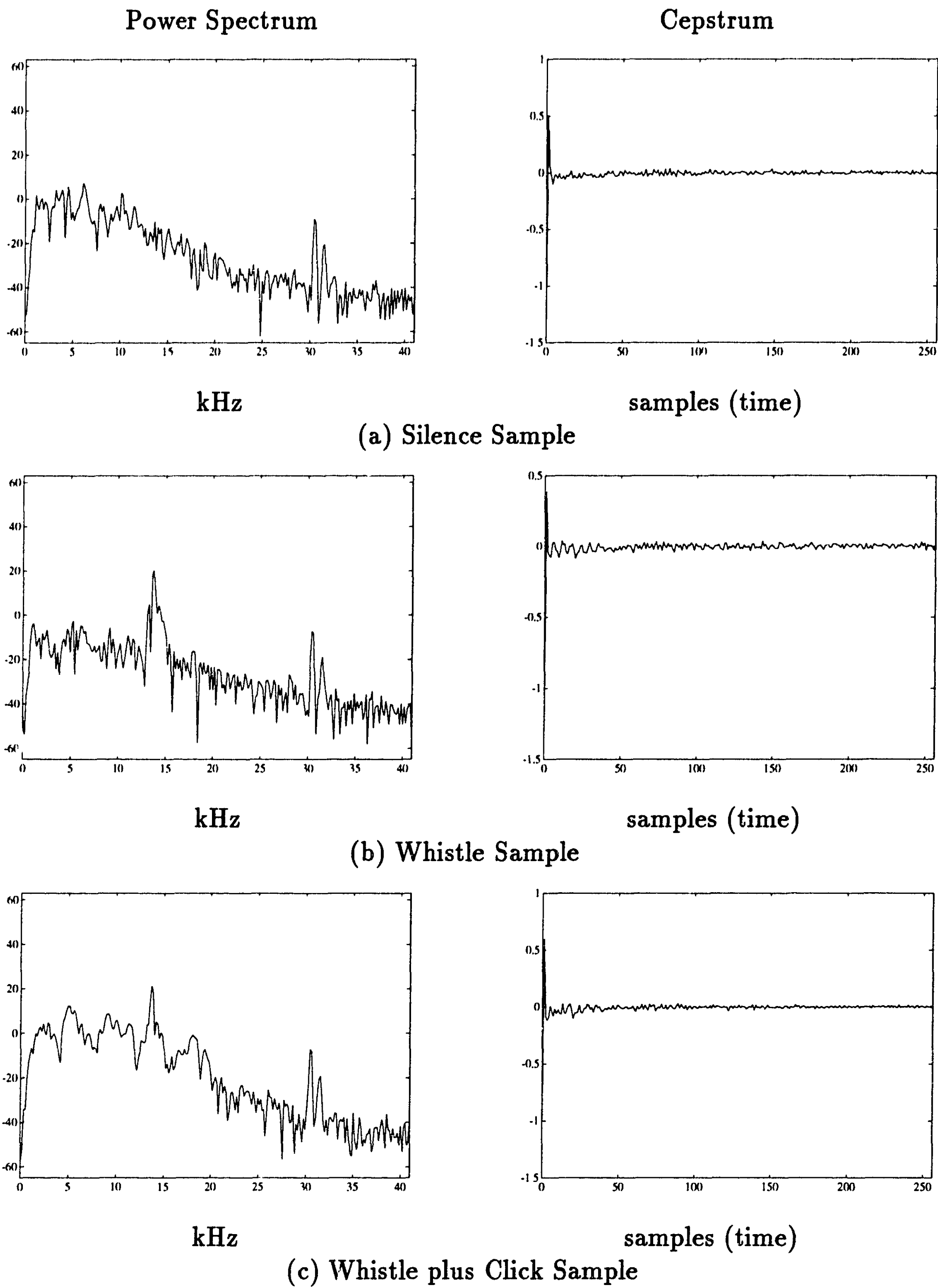


Figure 3-12: Sample spectra and cepstra for FB 62. Samples are 12.2μs apart. Only the (x) positive side shown in each plot.

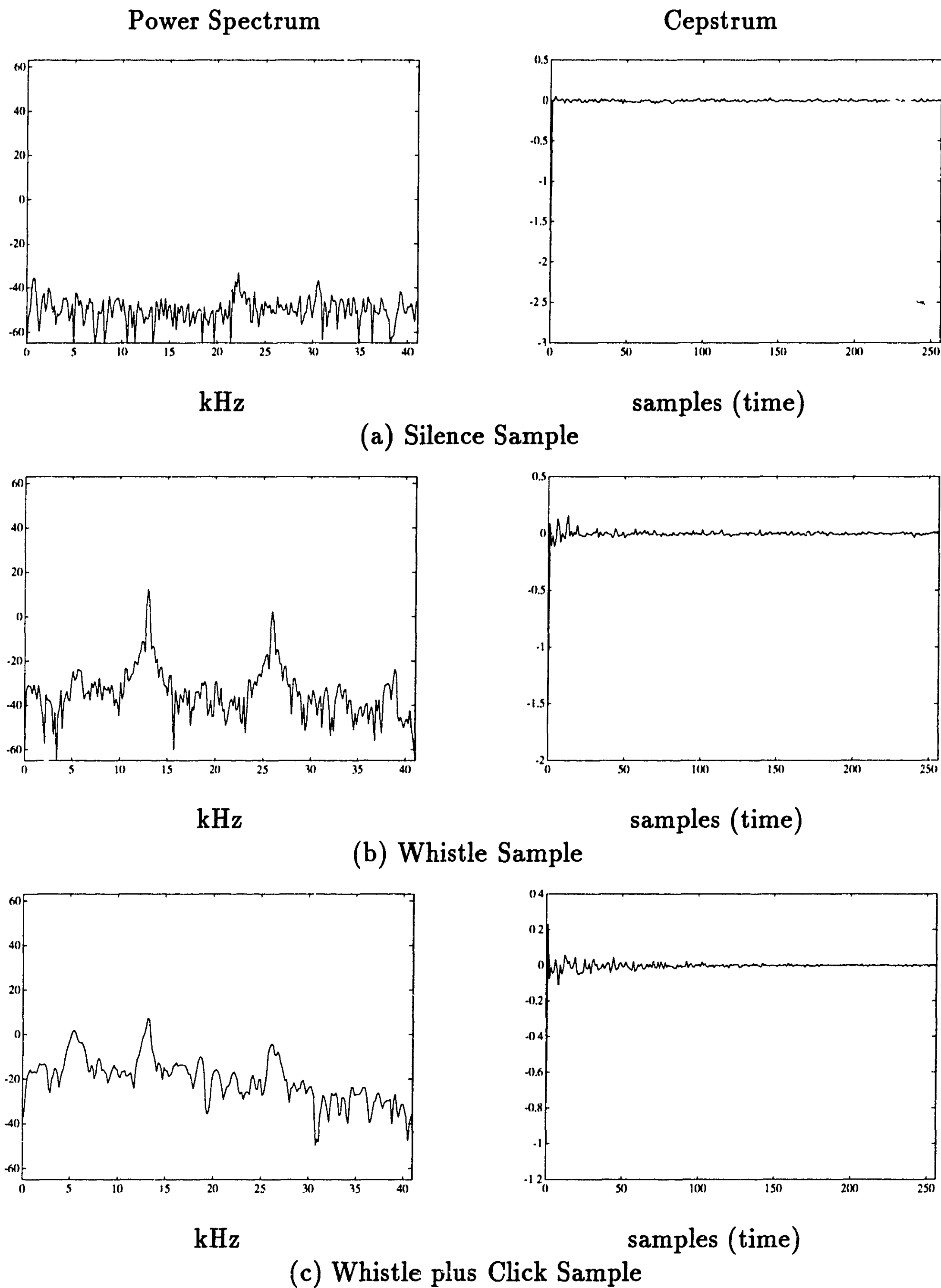


Figure 3-13: Sample spectra and cepstra for FB 153. Samples are 12.2μs apart. Only the (x) positive side shown in each plot.

Animal	Silence	Whistle	Whistle+Click
FB 35	0.097	0.216	0.029
FB 62	0.021	0.040	0.040
FB 153	0.052	0.167	0.064

Table 3.1: Cepstral peak amplitudes.

and the figures that a cepstral-based animal detector has little chance of dealing properly with clicks. If it were not for clicks, by comparing the amplitude of the largest peak[3] to the average amplitude of peaks above a a certain index, say 20, discrimination between dolphin and silence would be possible. However, from the cepstral-analysis perspective, the clicks have the effect of making a whistle look like silence. Thus, the cepstral-based detector would have many misses.

There are a few things unrelated to animal detection worth mentioning from this study. First, the peaks sometimes observed at low index (0 and 1) are the result of the tilt in the power spectrum. Compare the silence windows of FB 35 and FB 62 in Figures 3-11 and 3-12 for example. Second, in contrast to human speech and assuming an excitation plus vocal tract model can be used with dolphins, the excitation used for generating the whistles is of a frequency comparable to that of the resulting whistle. Notice that a high-index cepstral peak, characteristic of voicing in humans, is absent in the dolphin cepstra. Research on the excitation signal is currently under way at the Woods Hole Oceanographic Institution. Cepstral analysis may be used by scientists at Woods Hole as part of their investigation.

Finally, by using the cepstrum it is possible to obtain a smooth version of the power spectrum. Basically, one may filter the cepstral signal and Fourier-transform it back into the frequency domain. Two examples of this operation[4] are shown in Figure 3-14. It was attempted to use this smoothed spectra for finding the frequency-vs.-time trace. The hope was that by removing some of the noise in the power spectrum, the

[3]Ignoring peaks at 0 and 1.

[4]Only 41 cepstral components were used, 20 on each side plus the value at zero.

tracing algorithm would have an easier time processing the signals. Unfortunately, the smoothed spectra were not consistent from window to window. As can be seen at the bottom of Figure 3-14, the peak in the smoothed spectrum does not align perfectly with that obtained by the periodogram. This shift in frequency resulted in jagged traces.

To summarize, cepstral analysis of the dolphin signal did not result in a viable animal detector. The main problem with such a detector would be dealing with clicks in the signal. In addition, attempts to use power spectra obtained after filtering the cepstrum produced jagged frequency-vs.-time trace. Such traces were unacceptable. On the positive side, cepstral analysis proved itself as a possible tool in the research on signal excitation. This research, however, is beyond the scope of this thesis.

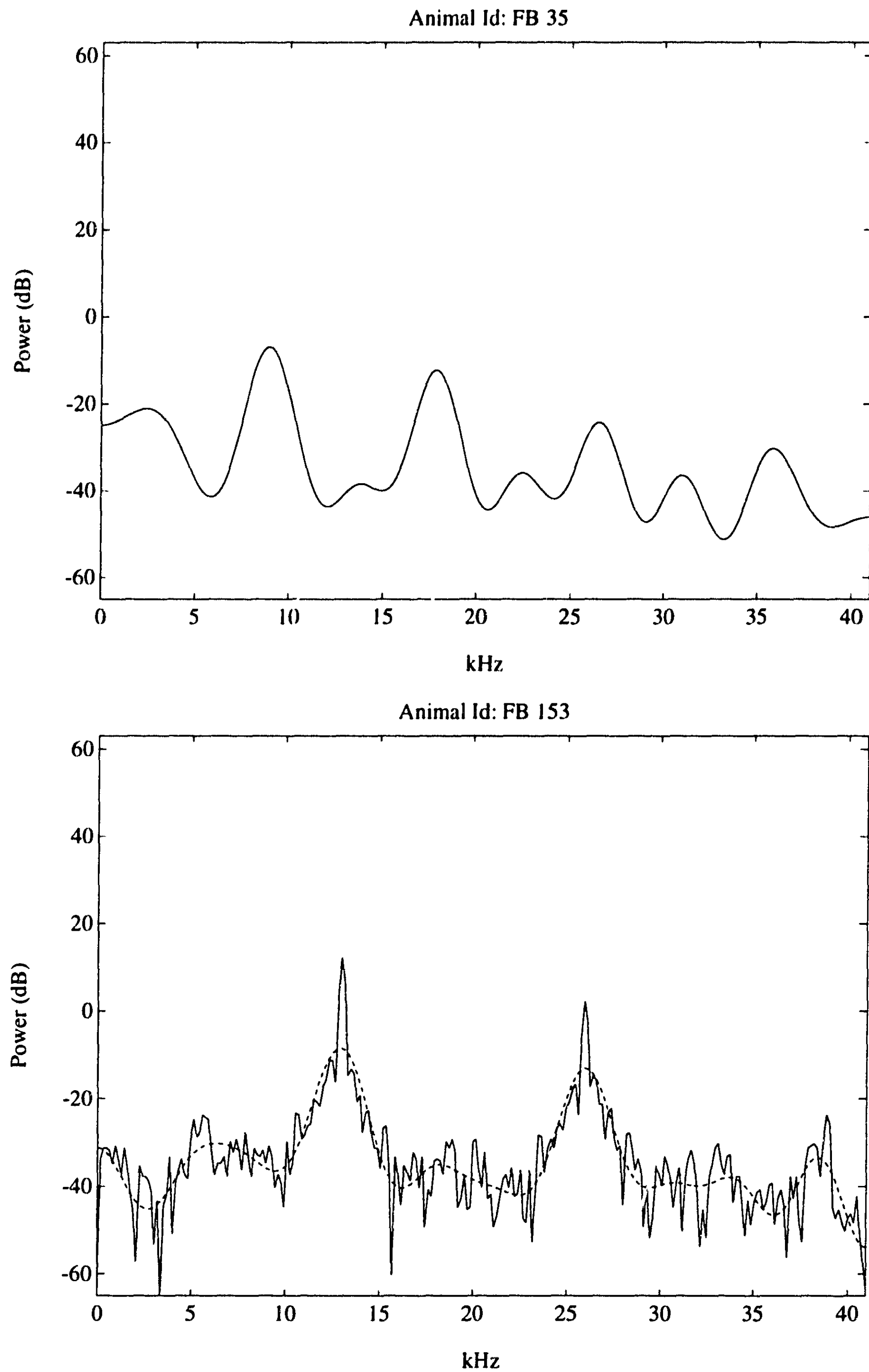


Figure 3-14: Power density spectrum from filtered cepstrum. Two examples, one from FB 35 and one from FB 135, of spectra obtained by filtering the cepstrum. The FB 135 plot shows the corresponding periodogram spectrum. Slight mismatch between the location of peaks in the smooth spectrum and the periodogram is visible.

Chapter 4

Level III Compression - Coding Space

The output of the level II compressor is mostly a single frequency-vs.-time trace. Since an FFT (Fast Fourier Transform) is used, the frequency values are quantized according to the duration of the analysis window. The DC level, frequency equal to 0, is used to indicate gaps in the trace. It is assumed that the trace contains all the necessary information for the detection of repetitions. That is, all information needed to identify the nearest trace to a given target trace. There is other output available from the level II compressor but it is used for other tasks such as trace editing and reversing the level II stage and will not be considered here.

This chapter deals with the selection of the coding space used to represent the traces. Each trace corresponds to a point in the coding space. A good coding space is one where similar signals are clustered together tightly and there is a very large distance between clusters. Each dimension of the coding space corresponds to a measure that can be computed for the trace.

The first section will describe the dimensions proposed for the coding space. The selection process is described in the second section along with the statistics used in evaluating each space. Once the space is set, the third section considers the size of

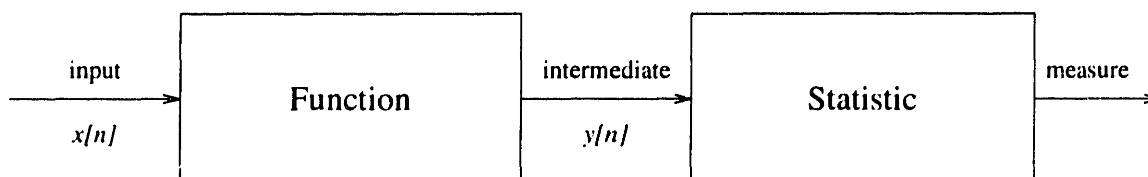


Figure 4-1: Block diagram for generating a large number of measures.

the potential vocabulary such a space may hold. Finally, the last section studies how well humans can classify these traces. The work with humans served to illustrate what it is that lets us separate one trace from another. The measures considered in the first section were based in part on what was learned with humans.

4.1 Candidate Dimensions

Two different approaches were taken to the creation of candidate dimensions. The first was to generate a very large set of candidates (above 200) and let a complicated evaluation process select a small set from this larger set. The second was to produce a smaller, more polished set, and have a less complicated selection process. The second approach was the one that worked out best.

Figure 4-1 shows the approach taken to produce a large number of measures for each signal. Basically, the signal first goes through a functional transform which produces an intermediate output of roughly the same duration as the signal. The intermediate output is then mapped to a single number by some statistical function. A total of 23 functions and 12 statistics were used to yield 276 measures.

The following list describes the 23 functions. When applying the function, gaps in the signal (marked with zero values) are skipped. When equations are provided, let $x[n]$ be the input and $y[n]$ the intermediate output.

1. raw - All-pass function. $y[n] = x[n]$ with gaps removed.

2-4. pow2, pow3, pow4 - Power functions. $y[n] = (x[n])^p$ where $p = 2, 3, 4$ and gaps

are removed.

5-6. der1, der2 - Simple derivatives. The function der1 is the first-order derivative, $y[n] = x[n] - x[n-1]$. The second-order derivative, der2, is given by $y[n] = x[n+1] - 2x[n] + x[n-1]$. No derivatives are computed near gaps.

7-8. nder1, nder2 - Normalized derivatives. Similar to the derivatives computed by der1 and der2 but with each point scaled by the frequency at the point. These function combine information from both the absolute pitch and the relative pitch of the signal. For nder1, $y[n] = (x[n] - x[n-1])/x[n-1]$. For nder2, $y[n] = (x[n+1] - 2x[n] + x[n-1])/x[n]$.

9-10. sder1, sder2 - Smooth derivatives. Derivatives computed with non-successive points. The function sder1 is given by $y[n] = (x[n+1] - x[n-1])/2$ and sder2 given by $y[n] = (x[n+2] - 2x[n] + x[n-2])/4$.

11-16. absder1, absder2, nabsder1, nabsder2, sabsder1, sabsder2 - Absolute derivatives. Contains information about how fast a signal is changing without paying attention to which direction the signal is changing. The definitions are the same used in 5 through 10 above but the absolute value of $y[n]$ is taken.

17-19. pred_err0, pred_err1, pred_err2 - Prediction errors. The zeroth-order, $y[n] = x[n] - x[n-1]$, first-order, $y[n] = x[n] - (2x[n-1] - x[n-2])$, and third-order, $y[n] = x[n] - (3x[n-1] - 3x[n-2] + x[n-3])$, prediction errors. No prediction errors computed near gaps.

20-22. moment1, moment2, moment3 - Moment-like functions. For these functions, $y[n] = x[n] * n^p$ where $p = 1, 2, 3$ depending on the function. The functions are considered to be moment-like because the sum of $y[n]$ is the moment of x. To increase the chance of making a correct match when gaps are present, the gaps were included when applying these functions.

23. logarithm - Natural log. $y[n] = \ln(x[n])$. Gaps are removed before taking the log.

As for the statistics, the following lists covers the 12 used.

1. sum - Measure is the sum of all intermediate values.
2. mean - Measure is the sum of all intermediate values, divided by the number of values added.
3. std - Measure is the standard deviation of intermediate values.
4. median - After sorting the intermediate values, the median is selected as measure.
5. avedev - This statistic is similar to the standard deviation, but instead of estimating $\sqrt{(\mathrm{E}\{(y-m)^2\})}$ it estimates $\mathrm{E}\{|y-m|\}$, where m is the mean of y.
6. avedev_median - Same as last statistic, but use median instead of mean.
7. percentile20 - Similar to median, but instead of selecting the 50 % point, the 20 % point is used.
8. percentile80 - Same as above but using 80 % point.
9. skewness - Given by the 3rd moment of the trace about its mean, divided by the 3rd power of the standard deviation. The mean and standard deviation are computed using standard probability formulas after normalizing the trace so that it integrates to unity.
10. kurtosis - This measure is similar to skewness but the 4th moment about the mean and power of the standard deviation is used. Also, 3 is subtracted from the ratio in order to make Gaussian-like (bell-shaped) traces have zero kurtosis.

11. avesgn - This statistic is given by the number of positive (or zero) values minus the number of negative values in the intermediate signal $y[n]$. Result is scaled by the number of points in $y[n]$.

12. aveabs - Same as mean but the absolute value of the intermediate signal is used.

Note that some combinations of functions and statistics are redundant. For example, if $y[n]$ is positive for all n, then the mean and aveabs statistics will generate the same output. Duplicated measures were removed from the set before selection of measures was done. The complex process used initially to select the coding space involved genetic algorithms [12]. In genetic algorithms, a string of ones and zeros of length equal to the number of dimensions available is used. A one in any location of the string means that the dimension corresponding to that location is used in the coding space. Also, a fitness function is defined which computes the relative worth of any particular string. The fitness function used in this work was a clumping measure which estimated how clumped the data was in the coding space specified by the binary string.

Starting with an initial, random population of such strings, the genetic algorithm starts by computing the fitness of every string in the population. Based on fitness, 2 strings are selected for reproduction. These 2 strings generate 2 new strings (offsprings). The population is either expanded to accommodate the offsprings or 2 elements of the population are removed (based on fitness) to accommodate the offsprings. The process continues until a specific performance is achieved or the desired number of generations have passed.

Unfortunately, the genetic algorithm was not able to deal with highly correlated candidate dimensions. Not even when a penalty for correlation was incorporated into the cost function did the genetic algorithm work. It was common for the genetic algorithm to select a coding space with, for example, both the average and the median of a function as dimensions. Nevertheless, the use of genetic algorithms helped identify some of the most promising measures in the set.

Measure name
1. Eigenvector 1 projection
2. Eigenvector 2 projection
3. Eigenvector 3 projection
4. Eigenvector 4 projection
5. Eigenvector 5 projection
6. Eigenvector 6 projection
7. Eigenvector 7 projection
8. Eigenvector 8 projection
9. Eigenvector 9 projection
10. Eigenvector 10 projection
11. Mean trace frequency
12. Median trace frequency
13. Mean + Median trace frequency
14. Mean - Median trace frequency
15. Mean-frequency crossing rate of trace
16. Frequency range (using average of extremes)
17. Frequency range (btwn fifth-most extremes)
18. Mean of absolute first-order derivative
19. Median of first-order derivative
20. Zero-crossing rate of derivative
21. "Duty cycle"
22. Duration in seconds

Table 4.1: List of candidate dimensions (measures) for the coding space.

Given the inability of genetic algorithms to deal adequately with the correlations between dimensions, a smaller set of candidates dimensions was produced. With the smaller set, it was easier to select uncorrelated dimensions for the coding space. Only 22 measures were considered and are listed in Table 4.1 and described below. The proposal of measures was mostly driven by knowledge of which characteristics one wants the recognition system to distinguish. This knowledge was obtained from experience with the signal as well as a study in which humans were asked to classify different whistles into clusters (see Section 4.4). Knowing what should be distinguished helps in creating a measure that will serve for that purpose. Also, the measures sought were continuous to avoid making decisions that could affect the adjacency of whistles.

1-10. Eigenvector 1-10 projections - A set of 92 traces from 7 animals was used to find a KL expansion. To make all traces be the same length, the traces were normalized to a length of 128 samples by interpolation. After finding the KL expansion the eigenvectors with the 10 largest eigenvalues were selected for computing "linear" measures. When computing the measure, the trace is first normalized to a length of 128 points, a stored mean vector is subtracted from it and the result is projected onto the eigenvector. The resulting number is used as measure.

11-14. Mean/Median operations on trace - Four measures are computed out of the mean and median frequency in the trace. These four measures are the two values themselves, plus the sum and difference of the two values. High correlation index is expected between mean and median, by taking the two linear combinations the hope is to generate less correlated measures.

15. Mean-frequency crossing rate - Number of times the trace crosses its mean frequency value divided by the number of samples in the trace. For a crossing to be counted, the signal must remain on the other side for a duration at least 5 % of the total trace duration.

16-17. Signal range - The frequency range of the signal was computed in two different ways. One evaluates the range as the difference between the averages of the 5 largest frequency values and the 5 smallest frequency values. The other just defines the range to be the difference between the fifth largest frequency value and the fifth smallest.

18-19. Mean/Median operation on first-order trace derivative - After computing the derivative (Hz/msec) it was then used to find a mean and a median derivative. When computing the mean, the absolute value of each point in the derivative was used. Otherwise, the average mean would just be the last value minus the first divided by duration.

20. Zero-crossing rate of first-order derivative - The derivative was also used to compute a zero-crossing rate with the resulting rate used as measure. The same definition of crossing given in 15 (above) applies here.

21. "Duty cycle" - This measure is defined as the difference between the duration of upward (positive slope) portions of the trace and the duration of downward (negative slope) portions of the trace. After the difference is computed, the sign is modified such that a positive measure indicates that the longest upward portion was found before the longest downward portion. The result is also normalized by dividing by the total duration. All durations for this measure are computed in samples.

22. Duration - How long the trace is in seconds.

From this list of 22 measures, 16 measures were selected to form the coding space. The selection procedure is covered in the next section. As can be seen from the list, it contains some of the measures explored initially (the over-200-measures set) plus some original ones. The intent was to preserve many of the simple and fundamental measures of the first set, specially those which the genetic algorithm picked often.

4.2 Evaluation and Selection of Coding Spaces

Initial work in the selection of a coding space concentrated on being able to find a coding space using no other information than the data itself. That is, there was no knowledge of which cluster the trace belonged to or how the trace should be classified. Techniques such as vector quantization (VQ) [25] and genetic algorithms were examined under this framework.

Genetic algorithms have already been discussed in the previous section. The role of vector quantization was the following. If one quantizes the coding space using VQ, with one code vector corresponding to each cluster, then the average distance between

code vectors is a possible quality measure of the space. However, this method is fairly close to simply identifying the clusters manually and computing the average distance between centroids. With manually identified clusters, VQ is not really necessary. Therefore, it did not make sense to use VQ as a quality measure of coding spaces.

Another use considered for vector quantization was that of database compression. For this purpose, each signal segment is represented by a code vector. By having 1024 code vectors, for example, each signal can be quantized using just 10 bits. However, the code vectors will be tailored to the specific database from which they were computed. As new signals come into the database, performance will suffer. The VQ method has no way of knowing that a previously empty region of space is going to hold a cluster later on. If the space is to be partitioned and a code vector be assigned to each partition, a more general way of partitioning should be used. As with genetic algorithms before, vector quantization was recognized as the incorrect approach to use in this work.

After the experiments with the "blind" approach (i.e. without any knowledge of to which cluster a trace belongs) to coding space selection failed, it became obvious some additional information was required. This additional information came in the form of cluster identification tags for each trace. Table 4.2 shows from where the 92 traces used in evaluating dimensions were obtained. As seen from the table, a total of 8 manually-identified clusters (based on visual inspection of the traces) from 7 animals were used. Using the above clusters several selection statistics were developed and tested. The definition of the statistics will be done first, followed by the experimental results in final coding space.

It has already been mentioned that a good coding space is one in which clusters are small and spread apart from each other. The first selection statistic, called the signal-to-noise ratio or SNR[1], attempts to quantify our sense of what a good space is directly. The SNR is defined as the ratio of the average distance between cluster

[1] Not to be confused with the SNR defined in Chapter 2.

Clstr	Level II file	No.	Trace list (number of trace in file)
A	905780K1.lv2	16	2,4,10,11,12,13,14,16,17,18,24,30,31,35,38,40
B	90579BK1.lv2	10	8,13,16,18,24,28,31,41,44,47
C	90580AK1.lv2	7	19,20,21,24,26,31,44
D	90580BK1.lv2	8	9,13,15,17,31,36,53,54
E	90588K01.lv2	17	3,4,5,6,16,20,21,23,24,32,33,35,46,47,48,50,54
F	91537K02.lv2	19	2,5,8,11,12,15,16,20,21,25,26,30,32,38,42,45,50,52,54
G	91537K02.lv2	8	1,3,4,6,7,9,18,24
H	91546K01.lv2	7	2,7,11,15,19,22,34

Table 4.2: Medium database contents. - List of 92 traces used in the evaluation of candidate dimensions.

centroids over the average standard deviation of the clusters. Thus, the notion of being "spread apart" is captured by the average distance between centroids and the cluster "size" is represented by the standard deviation of the cluster.

Let $m(i,j)$ be the mean and $\sigma(i,j)$ be the standard deviation of cluster i along dimension j, then the SNR(j), the value of SNR along dimension j, is given by:

$$\mathrm{SNR}(j) = \frac{2\sum_{k=1}^{N}\sum_{l=k+1}^{N} |m(k,j) - m(l,j)|}{(N-1)\sum_{k=1}^{N}\sigma(k,j)} \tag{4.1}$$

where N is the total number of clusters. The SNR defined in Equation 4.1 only works for a single dimension. To deal with the entire space one can define a multi-dimensional SNR, MSNR, as follows:

$$\mathrm{MSNR} = \frac{2\sum_{k=1}^{N}\sum_{l=k+1}^{N} |\underline{m}(k) - \underline{m}(l)|}{(N-1)\sum_{k=1}^{N} \sqrt[2d]{|\Lambda(k)|}} \tag{4.2}$$

where $\underline{m}(k)$ is now a vector whose elements are the means along each dimension of cluster k, $\Lambda(k)$ is the covariance matrix of cluster k, and d is the dimensionality of the

coding space. When $d = 1$, the two SNR measures are the same. The notation $|\underline{m}|$ represents the L_2-norm of the vector $\underline{m}$ and $|\Lambda|$ represents the determinant of matrix Λ.

When going from a single dimension to a multi-dimensional space, it is necessary to normalize dimensions. Normalization consisted of subtracting the mean and dividing by the standard deviation of each dimension. This normalization makes the dimensions unit-less and comparable in value. That way one does not get a single dimension dominating any distance computation. Note that the normalization does not make each cluster have the same "size" (standard deviation).

One problem of the MSNR measure is that it is not appropriate for comparing coding spaces of different dimensionality. When moving from a 3-D space to a 4-D space, for example, all the distances between clusters can only remain the same or increase and the covariance-based definition of size captures less of the total volume occupied by the cluster. To fix the problem with the covariance-based size, the cluster was assumed to have a Gaussian distribution along each dimension and the size was computed as the volume necessary to capture 70 % of the total "probability" [2]. A fix for the distance could be to divide the average distance by the number of dimensions in the space. However, the distance fix resulted in a monotonic decrease in MSNR as the dimensionality increased, in complete disagreement with other selection statistics. Results were more reasonable without the distance fix but questions remained about the validity of the MSNR selection statistic. There is little doubt that the MSNR works adequately when comparing spaces of same dimensionality, however, caution is recommended when comparing spaces of different dimensionality.

Another selection statistic used was the overlap count, OVLPC. This statistic refers to the number of points that overlap a cluster in which they do not belong. Clearly, the lower the value of OVLPC, the better the coding space. For this statistic a cluster is defined as the smallest hypersphere that encloses all the points in the cluster. If a point not belonging to the cluster lies inside the hypersphere (i.e. is

closer to the centroid of the hypersphere than the farthest point in the hypersphere) then it is counted in the OVLPC. When the overlap count is zero (non-overlapping hyperspheres) the coding space is considered to be "consistent" because there are no ambiguities regarding which point belongs to which cluster.

One advantage of the OVLPC statistic is that there is no problem when comparing spaces of different dimensionality. One disadvantage is the fact that all sets of measures (spaces) with OVLPC equal to zero are considered equally good by this statistic while it makes sense for a space to get better as the distance between the hypersphere boundaries increases. Unfortunately, this saturation-like behavior is also shared by the selection statistics to be discussed next.

A third selection statistic was based on the average distance of points to the centroid of its cluster. The average was computed in 2 different ways. The AVED1 statistic was computed by taking the average of the distance of *each point* to the centroid of its cluster. For the AVED2 statistic, however, one first computes an average for *each cluster* and then computes the average of cluster averages. In the first method each point contributes equally to the statistic while in the second it is each cluster that contributes equally to the statistic. Given the fairly uniform size of the clusters (see Table 4.2) used in computing the measure, both statistics took similar values.

For the AVED1 and AVED2 statistics, the lower its value the better. A value of zero corresponds to a space in which all the points in a cluster have the same coordinates. These statistics emphasize cluster "size" over separation. Also, the 2 statistics suffer from the same limitation encountered with MSNR when comparing coding spaces of different dimensionality. The average distance will go up as the dimensionality increases without necessarily meaning the higher dimensional space is any worst.

Finally, the last statistic considered was based also on distance to cluster centroids. For each point, the distances to the centroids of all clusters were computed and sorted.

The average rank of the distance to the centroid of the cluster where the point belongs was then computed. Again, this average was computed in 2 ways. Statistic AVER1 is the average rank when all points contribute equally and AVER2 is computed with all clusters contributing equally. The best possible value of AVER1 and AVER2 is an average rank of 1. For this case, each point is closer to the centroid of the cluster they belong in than to the centroid of any other cluster. Having an average rank equal to 1 does not mean there is zero overlap. On the other hand, zero overlap does mean the average rank is 1. As it was the case with OVLPC, the average rank measures can be used to compare spaces of different dimensionality.

The four "families" of selection measures are shown again in Table 4.3 along with short descriptions. In general, the overlap count and average rank are the only good statistics for comparing spaces of different dimensionality. Although both cluster size and separation affect the value of the statistics, only the SNR statistic considers both directly. The overlap count seems more sensitive to cluster distance while the average distance statistics are more sensitive to cluster size. For the average rank statistic it is hard to tell which aspect dominates.

Experiments done with coding spaces of different dimensionalities by both this author and Carlos Cabrera [2] were unable to determine the optimal coding space. Even with a small set of candidate dimensions, direct evaluation of every possible coding space is impossible. Simple techniques such as forward and backwards algorithms produced inconclusive results. In the forward algorithm one starts with a 1-D space, the one with best performance, and try all possible 2-D spaces that contain the best single dimension. After finding the best 2-D space, one looks for the best 3-D space by trying all other dimensions together with those from the best 2-D space. The process continues until all dimensions are used or degraded performance makes continuing unnecessary. The backwards algorithm is similar but instead one starts with a high dimensionality space and removes dimensions one at a time while minimizing performance loss.

Stat.	Name	Description
MSNR	signal-to-noise ratio	Ratio of average cluster separation to average cluster size. Not appropriate for comparing different dimensionality spaces. Only statistic without saturation.
OVLPC	overlap count	Number of points in coding space that overlap a cluster to which they do not belong. Can be used to compare different dimensionality spaces. The statistic emphasizes cluster separation.
AVED1, AVED2	average centroid distance	The statistic computes the average distance from point to centroid of their cluster in 2 ways. AVED1 computes ave. by point and AVED2 computes ave. by cluster. Not appropriate for comparing spaces with different dimensionality. Emphasis on cluster size.
AVER1, AVER2	average rank	After computing and sorting the distance from each point to every centroid, the statistic computes the average rank of the distance corresponding to the centroid in which the point belongs. Average computed in 2 was like the average centroid distance. Can be used for comparing spaces with different dimensionality.

Table 4.3: List of selection statistics for evaluation of coding spaces.

In order to compare spaces of different dimensionality, the overlap count and average rank selection statistics were used first. However, these statistics saturated too quickly. That is, the statistics reach their best possible value too quickly. After the statistic saturates, it is not possible to compare coding spaces. The performance of the average distance statistics simply increased with increasing dimensionality. The MSNR, on the other hand, initially exhibited increasing performance with increasing dimensionality until it ran into numerical problems at the 6th dimensional space. The 6th dimensional coding space for the MSNR was composed of the dimensions: eigenvector 1 and 2 projections, mean trace frequency, median trace frequency and the 2 linear combinations of the mean and median. Because of the high correlation between dimensions, the determinant of the covariance matrix should be nearly zero. However, the algorithm used to compute the determinant was returning a negative value for some clusters.

Given the problems with correlation, it was decided to compute the correlation coefficient for each pair of measures and remove redundant measures. Also, we ceased comparing spaces of different dimensionality since the best statistics saturated too quickly. The MSNR seemed to work with spaces of different dimensionality, but it could be rendered ineffective by correlations between dimensions. Instead of trying to select the best multidimensional coding space, it was decided to select the best dimensions independently and form a multidimensional space with them. Since correlation between dimensions is being taken care of already, selecting dimensions independently should work reasonably well.

Table 4.4 shows the selection statistic values obtained for each dimension. For statistics such as average rank and distance, only average by cluster is shown. When sorting the dimensions by each statistic, there is not an exact match in the sorted lists. For example, the eigenvector 2 projection is the best single dimension according to MSNR and AVED2, 5th according to OVLPC, and 7th according to AVER2. Nevertheless, the correlation between the statistics is strong (see Table 4.5) which

Measure name	MSNR	OVLPC	AVED2	AVER2
1. Eigenvector 1 projection	11.35	69	0.084	1.250
2. Eigenvector 2 projection	12.52	79	0.068	1.399
3. Eigenvector 3 projection	8.23	45	0.115	1.256
4. Eigenvector 4 projection	5.76	123	0.128	1.430
5. Eigenvector 5 projection	5.74	146	0.148	1.368
6. Eigenvector 6 projection	2.10	344	0.351	2.443
7. Eigenvector 7 projection	3.08	167	0.220	1.637
8. Eigenvector 8 projection	1.90	263	0.308	1.976
9. Eigenvector 9 projection	1.29	262	0.360	2.318
10. Eigenvector 10 projection	1.41	385	0.517	2.700
11. Mean trace frequency	11.81	87	0.084	1.366
12. Median trace frequency	9.99	133	0.092	1.384
13. Mean + Median trace frequency	11.70	68	0.085	1.236
14. Mean - Median trace frequency	4.99	134	0.161	1.503
15. Mean-frequency crossing rate of trace	4.54	136	0.199	1.514
16. Frequency range (averaging extremes)	8.87	119	0.084	1.395
17. Frequency range (fifth-most extremes)	7.13	86	0.098	1.513
18. Mean of absolute 1st-order derivative	2.47	242	0.284	2.251
19. Median of trace 1st derivative	5.10	126	0.190	1.455
20. Zero-crossing rate of derivative	1.50	345	0.507	2.742
21. "Duty cycle"	3.34	171	0.263	1.665
22. Duration in seconds	8.50	50	0.111	1.312

Table 4.4: List of candidate dimensions (measures) for the coding space.

indicates there is general agreement as to the relative fitness between dimensions. Also included are scatter plots of MSNR vs. OVLPC (Figure 4-2) and AVED2 vs. AVER2 (Figure 4-3).

As mentioned above, the correlation coefficient between dimensions was used to remove redundant measures. Some of these correlation coefficients are shown in Table 4.6. In removing measures, a correlation coefficient of 0.8 (absolute value) or larger was considered to be enough to justify removing one of the two dimensions. Which dimension to remove was based mostly on their MSNR values. Whenever the MSNR values seemed too close to each other, then OVLPC was used. It has already been established that general agreement exists between selection statistics and there-

Stat	Selection Statistic			
	MSNR	OVLPC	AVED2	AVER2
MSNR	1.0000	-0.8160	-0.8521	-0.7872
OVLPC	-0.8160	1.0000	0.9488	0.9642
AVED2	-0.8521	0.9488	1.0000	0.9565
AVER2	-0.7872	0.9642	0.9565	1.0000

Table 4.5: Correlation coefficients for selection statistics.

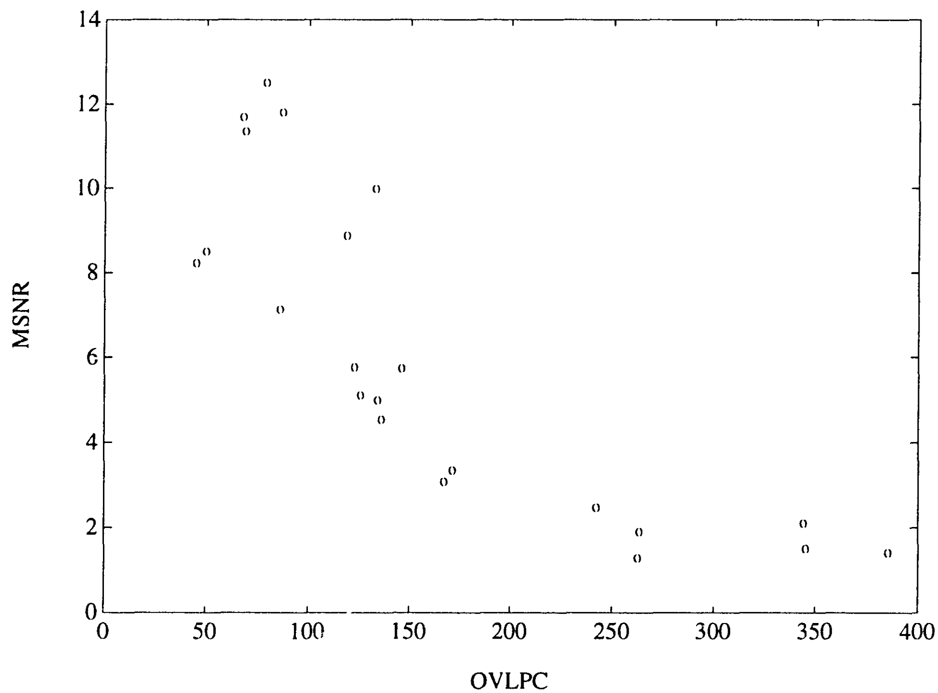


Figure 4-2: MSNR vs. OVLPC scatter plot.

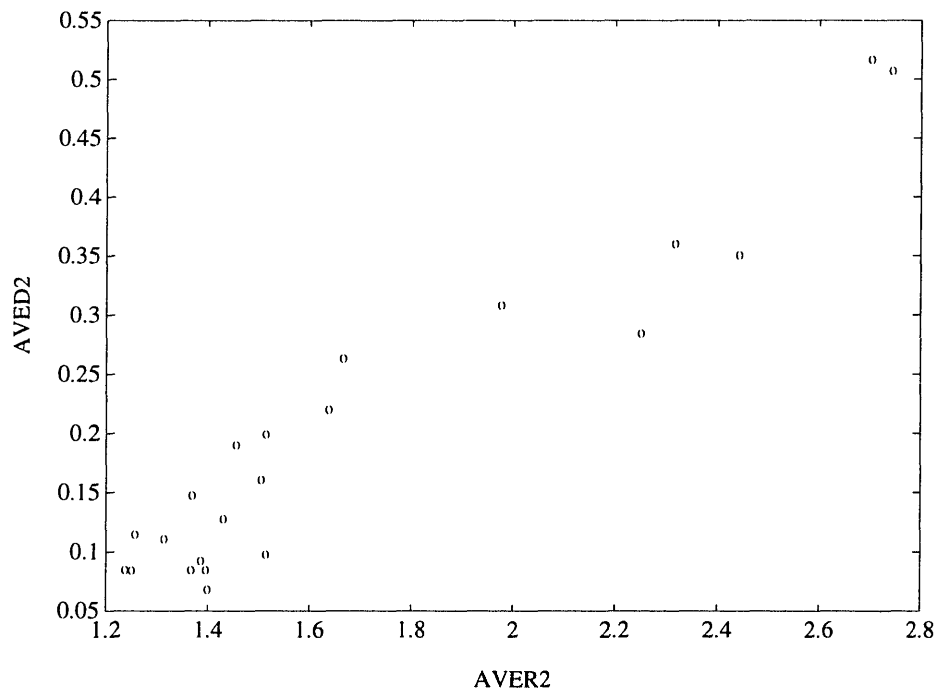


Figure 4-3: AVED2 vs. AVER2 scatter plot.

	1	**2**	**3**	**4**	**5**	**11**	**12**	**13**	**14**	**16**	**17**	**18**	**19**	**20**
1	1.00	0.15	0.07	0.11	0.06	0.85	0.77	0.83	0.14	0.42	0.29	0.24	0.55	0.12
2	0.15	1.00	0.37	0.18	0.34	0.60	0.48	0.55	0.05	0.79	0.87	0.05	0.21	0.03
3	0.07	0.37	1.00	0.34	0.23	0.08	0.07	0.07	0.01	0.72	0.69	0.54	0.50	0.03
4	0.11	0.18	0.34	1.00	0.06	0.00	0.23	0.13	0.50	0.20	0.26	0.42	0.18	0.34
5	0.06	0.34	0.23	0.06	1.00	0.16	0.18	0.03	0.66	0.37	0.33	0.45	0.14	0.60
6	0.17	0.30	0.16	0.30	0.08	0.26	0.37	0.33	0.33	0.16	0.23	0.05	0.27	0.16
7	0.04	0.44	0.32	0.42	0.33	0.23	0.05	0.08	0.53	0.43	0.43	0.17	0.24	0.29
8	0.05	0.08	0.22	0.22	0.06	0.03	0.07	0.05	0.10	0.22	0.22	0.14	0.14	0.06
9	0.06	0.20	0.27	0.18	0.13	0.03	0.02	0.02	0.01	0.24	0.25	0.10	0.08	0.11
10	0.02	0.08	0.26	0.02	0.41	0.11	0.22	0.17	0.28	0.09	0.04	0.22	0.20	0.43
11	0.85	0.60	0.08	0.00	0.16	1.00	0.89	0.97	0.11	0.63	0.58	0.09	0.20	0.10
12	0.77	0.48	0.07	0.23	0.18	0.89	1.00	0.98	0.55	0.55	0.52	0.39	0.08	0.18
13	0.83	0.55	0.07	0.13	0.03	0.97	0.98	1.00	0.36	0.60	0.56	0.26	0.14	0.05
14	0.14	0.05	0.01	0.50	0.66	0.11	0.55	0.36	1.00	0.04	0.07	0.66	0.19	0.55
15	0.13	0.46	0.10	0.12	0.79	0.27	0.01	0.13	0.47	0.45	0.42	0.49	0.06	0.61
16	0.42	0.79	0.72	0.20	0.37	0.63	0.55	0.60	0.04	1.00	0.98	0.34	0.28	0.13
17	0.29	0.87	0.69	0.26	0.33	0.58	0.52	0.56	0.07	0.98	1.00	0.29	0.13	0.05
18	0.24	0.05	0.54	0.42	0.45	0.09	0.39	0.26	0.66	0.34	0.29	1.00	0.40	0.47
19	0.55	0.21	0.50	0.18	0.14	0.20	0.08	0.14	0.19	0.28	0.13	0.40	1.00	0.13
20	0.12	0.03	0.03	0.34	0.60	0.10	0.18	0.05	0.55	0.13	0.05	0.47	0.13	1.00
21	0.55	0.60	0.06	0.19	0.16	0.11	0.12	0.12	0.07	0.20	0.36	0.27	0.58	0.05
22	0.17	0.47	0.09	0.63	0.39	0.14	0.44	0.31	0.68	0.22	0.36	0.32	0.54	0.56

Table 4.6: Absolute value of correlation coefficients for some dimensions - The dimension is identified by the numbers given in Table 4.1 (shown in bold).

fore either one of them would be appropriate. However, since MSNR was the only one that does not exhibit saturation, is was chosen as primary selection statistic.

There were 2 sets of variables that each had strong correlations between its members. One set contained the projection of the trace onto the second eigenvector and 2 frequency ranges, the other set contained the projection onto the first eigenvector and the mean frequency of the trace, median frequency, and their sum. Of these 7 dimensions, only the eigenvector projections, the 5-pt average frequency range and the sum of mean and median trace frequency remained. Normally, only one measure from each set should be preserved, however, it was decided to keep the two eigenvector projections because there seemed to be no real explanation, other than coincidence, for the correlation between the projections and the other variables in the set.

Measure name	MSNR	OVLPC	AVED2	AVER2
2. Eigenvector 2 projection	12.52	79	0.068	1.399
13. Mean + Median trace frequency	11.70	68	0.085	1.236
1. Eigenvector 1 projection	11.35	69	0.084	1.250
16. Frequency range (averaging extremes)	8.87	119	0.084	1.395
22. Duration in seconds	8.50	50	0.111	1.312
3. Eigenvector 3 projection	8.23	45	0.115	1.256
4. Eigenvector 4 projection	5.76	123	0.128	1.430
5. Eigenvector 5 projection	5.74	146	0.148	1.368
19. Median of trace 1st derivative	5.10	126	0.190	1.455
14. Mean - Median trace frequency	4.99	134	0.161	1.503
15. Mean-frequency crossing rate of trace	4.54	136	0.199	1.514
21. Duty cycle	3.34	171	0.263	1.665
7. Eigenvector 7 projection	3.08	167	0.220	1.637
18. Mean of absolute 1st-order derivative	2.47	242	0.284	2.251
6. Eigenvector 6 projection	2.10	344	0.351	2.443
8. Eigenvector 8 projection	1.90	263	0.308	1.976

Table 4.7: Coding space dimensions (measures).

Also, the reason why the sum of the mean and median frequency (MSNR=11.81) was kept instead of just the mean frequency (MSNR=11.70) was that the sum had the second best OVLPC value while MSNR values were fairly close. This leaves 19 measures from which to select a coding space. Since experiments with different coding space sizes were unsuccessful, the final dimensionality of the space was set to 16 somewhat arbitrarily[2].

Based on MSNR, a coding space using the 16 dimensions listed in Table 4.7 was created. The 16 dimensions are sorted by MSNR. For the overall space, the MSNR could not be computed because 3 of the 8 clusters had determinants with values less than zero. If a size of zero is used for the 3 problem clusters, an MSNR of 238.5 results.

[2]The number 16 was chosen because that is the size of short binary integers. If one ever decides to quantize each dimension to 1 bit, the quantized representation will pack well into a short int variable.

4.3 Animal Potential Vocabulary

This section considers the question of how many distinct signals can the coding space hold. The number of signals will be the potential vocabulary for the animals. Computing this number requires knowledge of the cluster size of each signal and the size of the entire space. It was decided to use the average cluster size taken over the 8 clusters in the set as the size of the typical cluster. The potential vocabulary is on the order of the number of these average clusters that one can fit in the total space. Only order-of-magnitude estimates can be given since a precise measure of the total and cluster volumes is unavailable.

The size of the full space will be estimated as follows. Since each dimension has been normalized to zero mean and unit variance, it is reasonable to assume that each dimension extends from -3 to $+3$ units. So, a quick estimate of the size for the entire space is 6^d, where d is the dimensionality of the space. In a 16 dimensional space, the total size is $6^{16} = 2,821,109,907,456$ units^{16}.

To measure the size of a cluster, one could use a definition similar to that used in MSNR. The size of a cluster would be given by

$$6^d \sqrt{|\Lambda(k)|} \tag{4.3}$$

where $\Lambda(k)$ is the covariance matrix of the cluster and d the dimensionality of the space. Note that there is no compensation for the decrease in volume captured by the expression as the dimensionality increases. If such a term is included, it should also be included into the expression for total space volume and the two corrections end up cancelling each other. Also, a $2d$-root is no longer needed since one is trying to compute a volume instead of a radius.

The 6^d factor in Equation 4.3 is used to make the volume (size) definitions compatible. Consider the case when the covariance matrix is a diagonal matrix. Then, without the 6^d factor, the cluster size would be given by the product of the standard

deviations along each dimension. However, the total space was computed using the product of 6 times the standard deviation (a $\pm 3\sigma$ interval) along each dimension. Thus, a 6^d factor is needed to make the size definitions equivalent. Because of this factor, the resulting potential vocabulary size is simply given by the reciprocal of $\sqrt{|\Lambda(k)|}$.

When computing the cluster size using the above definition we ran into trouble. The same numerical problems that prevented the computation of MSNR for spaces of dimensionality higher than 6 in the forward experiment, prevented the calculation of cluster sizes in the 16 dimensional space. The condition number of the covariance matrix for each cluster was examined, the smallest was 1.7245×10^7 while the largest was 1.2924×10^{33}. Three clusters had negative sizes. The problems were made slightly better when the projection onto the first 2 eigenvectors was removed (a 14 dimensional space) but the analysis remained impossible.

Given the problem with the current definition of cluster size, two possible alternatives were considered. First, one can keep the definition as is and, starting from a 1-D space, try to compute the number of signals that can fit in the space as a function of dimensionality. The number of signals is computed for as many subspaces as one can before running into the limitations of the cluster size definition. Then, results are extrapolated to 16 dimensions. If one uses the value of the determinant to determine the maximum dimensionality for which Equation 4.3 can be used, one must stop at a space of dimensionality equal to 6. If instead one uses the condition number of the covariance matrix, one must stop much sooner. This means that the extrapolation to the 16 dimensional space is going to be based on very few points.

Nevertheless, the extrapolation approach was tried. The results are listed in Table 4.8. As shown in the table, as dimensionality increases, the total volume goes up while the mean cluster size goes down. Roughly speaking, the potential vocabulary goes up an order of magnitude for each dimension in the space. The log of the potential vocabulary size was found to be linearly dependent on the dimensionality.

Dimensionality	Cluster Volume		Potential Vocabulary
	Mean	Max	
1	0.0852	0.2114	11.7306
2	0.0074	0.0175	135.8853
3	5.73×10^{-4}	0.0018	1745.3
4	5.20×10^{-5}	2.14×10^{-4}	1.92×10^{4}
5	5.06×10^{-6}	2.51×10^{-5}	1.97×10^{5}

Table 4.8: Cluster volume and potential vocabulary size as function of dimensionality. - The cluster volume shown does not include the 6^d factor of Equation 4.3.

Figure 4-4 shows the actual values for the log of the potential vocabulary (circles) and the linear fit given by $\log_{10}(v) = 1.0604d + 0.0236$ (solid line), where v is the potential vocabulary size and d is the dimensionality. From the equation, the estimated size of the 16-dimensional-space potential vocabulary is 9.8×10^{16}. A huge potential vocabulary can be supported by the coding space.

As to the second alternative, the cluster size definition can be modified to one without the above problems. Since the size of the total space is computed based on the products of $\pm 3\sigma$ intervals along each dimension, one can define cluster size as the product of the $\pm 3\sigma$ interval for the cluster along each dimension. This definition is equivalent to replacing the determinant in the previous definition (Equation 4.3) by the product of the elements along the main diagonal of the covariance matrix. Results are quite different when this definition is used. The potential vocabulary estimated for each dimensionality is the one given by the "×"s in Figure 4-4. For a 16 dimensional space, the potential vocabulary is 1.6×10^9 and the mean cluster volume is 6.2×10^{-10}.

The discrepancy between the two results (about 10^{17} and 10^9) can be attributed to a number of factors. First, the linear approximation may not hold as dimensionality increases. It is possible for the rate of increase of the potential vocabulary to get smaller as the dimensionality grows. Some of this is evident already, if one fits a line to the log of the potential vocabulary size for dimensionality 1 through 4, the slope

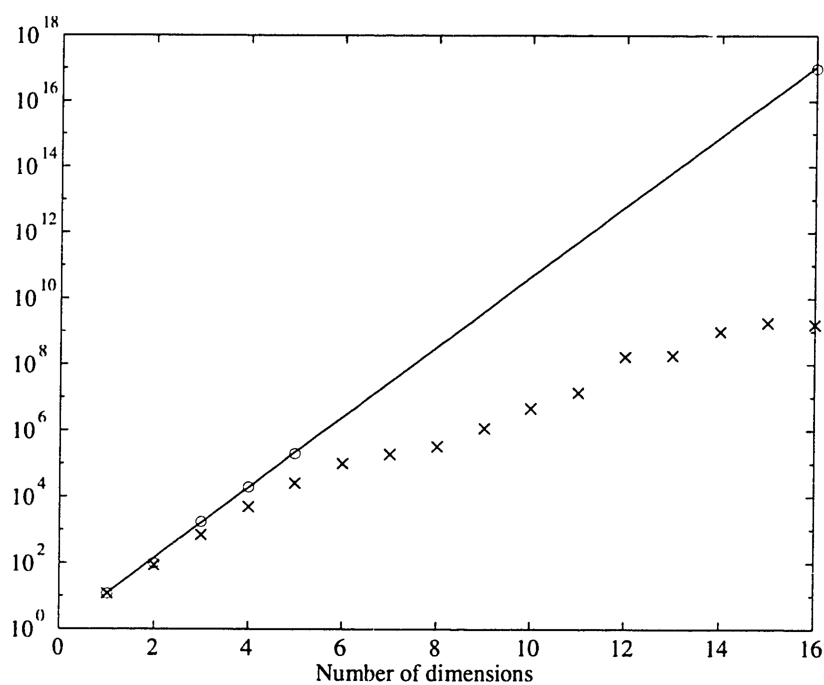


Figure 4-4: Log(potential vocabulary) versus dimensionality plot. - The plot shows the actual values (circles for method 1 and "×"s for method 2) and the linear fit (solid line) for method 1 given by the expression $\log_{10}(v) = 1.0604d + 0.0236$.

of the curve is 1.0751. Thus, since the slope of the linear fit seems to be decreasing as dimensionality increases, one can expect the 16 dimensional space to hold a potential vocabulary smaller than $O(10^{17})$.

Another possibility is that the second method is over-estimating the cluster volume. When the second method is used for a space of dimensionality 5, results are off by 1 order of magnitude (see Figure 4-4). Since this space should still be in the "linear" region of the size-vs.-dimensionality curve, the discrepancy should be caused by something other than the linear fit. It is possible that correlations among dimensions are making clusters smaller, and the second method, by ignoring off-diagonal elements in the covariance matrix, is ignoring this effect. Thus, a potential vocabulary bigger than $O(10^9)$ is possible.

It should be noted, however, that the above potential vocabulary estimates are based on whistles which are the signature whistle of the animal making the sound. No copies of whistles were used in estimating cluster volume. If one includes mimicry (copying) in the potential vocabulary estimates, the clusters should be bigger and the estimates smaller due to the observed reduction in frequency control when the animal copies a whistle. This reduction should increase the standard deviation of every frequency-based dimension by about a factor of 3. Since there are 14 frequency-based dimensions, the correction to apply to the potential vocabulary estimates is $3^{14} = 4782969$ (assuming uncorrelated dimensions). The range of the potential vocabulary then is from slightly over 300 whistles to 2×10^{10} whistles.

Figure 4-5 shows the same data as for Figure 4-4 but compensated for mimicry. As can be seen in the graph, the second method of calculating the potential vocabulary provides an stable estimate of "shared" potential vocabulary of a few hundred whistles. The estimate based on the first method continues to grow linearly with dimensionality.

Given the discrepancies between the 2 estimates of potential vocabulary size and the reasons for those discrepancies, all one can say at this time is that the coding

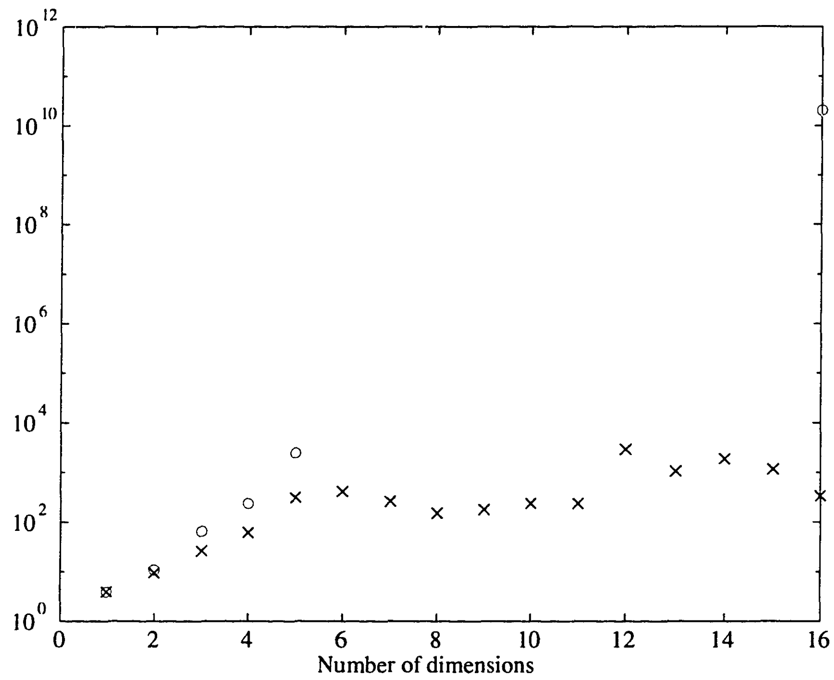


Figure 4-5: Log("shared" potential vocabulary) versus dimensionality plot. - The plot shows the actual values, circles for method 1 and "×"s for method 2.

space should be able to hold at least $O(10^9)$ different signals when mimicry is ignored and at least 300 when it is not. Note that this is the number of distinct signals the coding space may hold and is completely independent of the number of records (whistles) in the database.

4.4 Comparing Cluster Sets

To illustrate the difficulties in attempting to classify whistles into clusters and to learn more about what criteria is of importance for classifying whistles, a set of 180 whistles was given to several people who were asked to manually classify them. The group of people consisted of 4 subjects (labelled KF, PF, PT, and LS) and included both experts and non-experts in the area of dolphin whistle analysis.

Different subjects cluster whistles differently; as a result, one must find a way of comparing their cluster sets. This comparison can be used as a basis for evaluation of the classification task. However, such a comparison is complicated by the fact that no a-priori cluster assignments exist for these whistles. Also, there is really no correct way of classifying the whistles.

The measure selected for the comparison of cluster sets was the following.

Definition 1 *Cluster Similarity (CS)*

Minimum number of elements that must be removed from both sets in order to get a 1-to-1 mapping between clusters.

This measure can also be presented as a percentage (CS%) of the total number of whistles. Clearly, the value of CS can range between 0 and 179 for the data set specified.

Let us now describe in more detail the problem of finding the CS value. In general, we want to compare two sets of clusters, $A = \{A_1, A_2, A_3, ..., A_M\}$ and $B = \{B_1, B_2, B_3, ..., B_N\}$, which have been defined over a set of n elements. It is further

required that,

$$A_i \cap A_j = \emptyset \text{ for } i \neq j, B_i \cap B_j = \emptyset \text{ for } i \neq j, \text{ and } \aleph(A) = \aleph(B) = n$$

That is, that the clusters in each set be mutually exclusive and include all n elements. The notation $\aleph(A)$, the cardinality of A, is defined as the number of *elements* in the set A. Note that each cluster is also a set and therefore set operations (cardinality, intersection, etc.) can be performed on them. However, to avoid ambiguity, the term "set" will refer to a set of clusters only.

One can also define a cluster distance metric, $d_c(A_i, A_j)$, as the number of elements *not* in common between the two clusters. Mathematically,

Definition 2 *Cluster Distance*

$$d_c(A_i, A_j) = \aleph(A_i) + \aleph(A_j) - 2 * \aleph(A_i \cap A_j)$$

Note that the distance between two clusters is precisely the number of elements that need to be removed to make both clusters look the same. When the clusters have no elements in common, all elements are thrown out to get two identical empty clusters. It should be intuitively obvious that clusters assignments should minimize the distance between the clusters involved.

To obtain the 1-to-1 mapping required in the definition of CS, it will be necessary to use empty clusters. These empty clusters will be labeled NULL and any number of them can be added to a cluster set without affecting the set requirements given above. Strictly speaking, NULL clusters are only required when the number of clusters in each set is not the same. The smallest set should be augmented with empty clusters until both sets have the same number of clusters thus making a 1-to-1 mapping possible.

The distance of a cluster A_i to an empty cluster can be easily shown to be $\aleph(A_i)$. Thus, the contribution to CS of assigning $A_i \leftrightarrow B_j$ when the clusters do not intersect

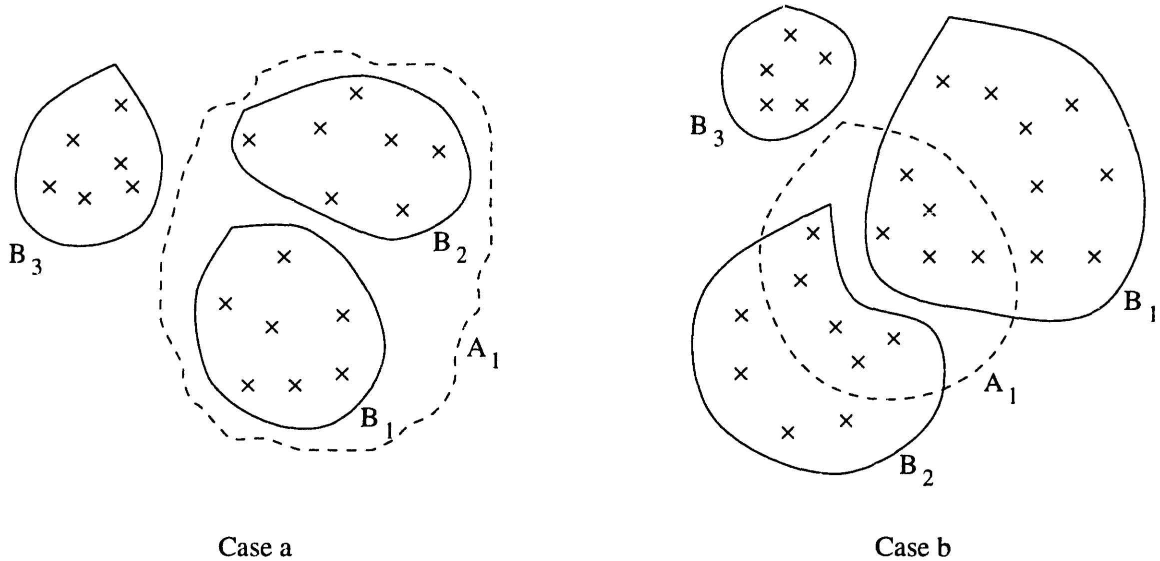


Figure 4-6: Cases when matching cluster A_1. - The resulting set $S = \{B_1, B_2\}$ for both cases. Data used is completely artificial.

is the same as assigning $A_i \rightarrow$ NULL and $B_j \rightarrow$ NULL. I consider it more pleasing to assign a cluster to NULL than to another cluster with nothing in common. Therefore, extra NULL clusters, beyond those required to match the number of clusters in the sets, will be added to avoid assigning disjoint (non-empty) clusters to each other.

Without loss of generality, let us consider matching cluster A_1 to a cluster in the set B. Start by forming a set $S = \{B_i : A_1 \cap B_i \neq \emptyset \text{ and } i = 1, ..., N\}$. Then one of the following must be true:

Case a: $\aleph(S) = \aleph(A_1)$

Case b: $\aleph(S) > \aleph(A_1)$

Figure 4-6 illustrates each possible case.

Theorem 1 *Having $\aleph(S) < \aleph(A_1)$ is impossible.*

Proof. (by contradiction) If $\aleph(S) < \aleph(A_1)$ then $\exists$ an element $q \in A_1$ such that $q \notin S$. Having $q \notin S$ implies that $q \notin B_i$ for some $i = 1, ..., N$ because if q were in any B_i it would have to be in S by construction. The existence of such an element q violates the requirement that $\aleph(B) = n$. Thus, no such element can exist and therefore

the theorem is true. ∎

Let us now discuss how to make cluster assignments for each of the possible two cases. Start with the simplest case to handle, case a ($\aleph(S) = \aleph(A_1)$). In this case the cluster A_1 in A has been subdivided into one or more clusters in the set B. The optimal assignment to A_1 is that which attains the minimum $\min_{B_i \in S} d_c(A_1, B_i)$. This result comes from the fact that only one of the clusters in S can be assigned to A_1 and it should obviously be the one closest to it. Note that the simplest possible case occurs when S has a single cluster resulting in a perfect match for A_1.

Case b ($\aleph(S) > \aleph(A_1)$) is harder to handle. The main reason for the difficulties with this case is that the set S and cluster A_1 do not contain enough information to allow the appropriate assignment of cluster A_1. Since all the elements in S are not contained in A_1 the possibility exist that another cluster in A can provide a better match to one of the clusters in S than A_1 can.

In order to correctly assign the cluster A_1 one must consider all the clusters affected by the assignment. This involves recursively finding all interconnected clusters. That is, after finding all the clusters in B which overlap A_1 forming the S set, one then finds all other clusters in A which overlap with any of the B clusters in S[3]. The process is repeated for each of the new A clusters. Let I be the set of all interconnected clusters. Only after the set I has been formed, is it possible to make an optimal assignment. Note that I contains clusters from both A and B.

There are two "greedy" algorithms which immediately come to mind for solving this problem. Using only the clusters in I one can:

[3]Other overlapping A clusters must exist since $\aleph(A_1) < \aleph(S)$. That is, there are whistles in S which do not belong to A_1.

Algorithm 1. Find the 2 clusters that overlap the most, assign them to each other. Remove the assigned clusters from I and repeat. This process continues until one runs out of either A or B clusters in I. The clusters left after no further assignments are possible are then assigned to NULL.

Algorithm 2. Find the 2 clusters that are closest (minimum $d_c(\cdot)$), assign them to each other. Remove the assigned clusters from I, update all affected distances, and repeat. This process continues until one runs out of either A or B clusters in I. The clusters left after no further assignments are possible are then assigned to NULL.

Let us now formalize the two algorithms by introducing the matrices R and M defined as follows:

$$\begin{aligned} R &= \{r_{ij}\} \text{ where } r_{ij} = \aleph(A_i \cap B_j) \\ M &= \{m_{ij}\} \text{ where } m_{ij} = d_c(A_i, B_j). \end{aligned}$$

The matrix M can be easily obtained from R by noting that m_{ij} is the sum of all the elements in the ith row and j^{th} columns of R except r_{ij} itself. After forming the above matrices for the A and B sets one can then form a submatrix using only those clusters in the I set. The discussion that follows applies equally to both a submatrix as well as the full matrix and making any distinctions will be avoided. However, as a practical issue, one should work with the submatrices. It is assumed, without loss of generality, that there are at least as many columns as there are rows in each matrix.[4]

[4]Some re-labeling may be required.

Algorithm 1 | Algorithm 2

$$R = \begin{bmatrix} 14 & 1 & 1 \\ 0 & 12 & 8 \end{bmatrix} \qquad M = \begin{bmatrix} 2 & 27 & 23 \\ 34 & 9 & 13 \end{bmatrix}$$

Both methods select $(1,1)$. CS$'$=2. Remove assigned rows and columns to get,

$$R = \begin{bmatrix} 12 & 8 \end{bmatrix} \qquad M = \begin{bmatrix} 8 & 12 \end{bmatrix}$$

Both methods select $(1,1)$, which corresponds to $(2,2)$ in original matrix. CS$'$=10.

Figure 4-7: Matching clusters - example 1: Both algorithms produce correct result. Assuming rows correspond to the A set and columns to the B set, the final assignments are: $A_1 \leftrightarrow B_1$, $A_2 \leftrightarrow B_2$, and $B_3 \rightarrow$ NULL.

Having defined the matrices, the two algorithms suggested are:

1. **Algorithm 1**
   (a) Find largest element in R, call it $r_{i_o j_o}$.
   (b) Match $A_{i_o} \leftrightarrow B_{j_o}$.
   (c) Remove row i_o and column j_o. Remap cluster to row/column assignments as needed.
   (d) If there are rows left, go to a. Otherwise, match clusters corresponding to the remaining columns to NULL.

2. **Algorithm 2**
   (a) Find smallest element in M, call it $m_{i_o j_o}$.
   (b) Match $A_{i_o} \leftrightarrow B_{j_o}$.
   (c) Remove row i_o and column j_o. Remap cluster to row/column assignments as needed.
   (d) Recompute distances affected by the previous step.
   (e) If there are rows left, go to a. Otherwise, match clusters corresponding to the remaining columns to NULL.

As Figure 4-7 illustrates, both algorithms are reasonable and may indeed produce the correct result. Note that after removal of the 1st row and column of the M matrix the remaining distances are reduced. This is because the first match already

Algorithm 1

$$R = \begin{bmatrix} 14 & 2 & 1 \\ 22 & 12 & 8 \end{bmatrix}$$

Select (2,1). CS′ = 34. After removing assigned rows and columns we get,

$$R = \begin{bmatrix} 2 & 1 \end{bmatrix}$$

Select (1,1) which corresponds to (1,2) in the original matrix. CS′ = 35.

Algorithm 2

$$M = \begin{bmatrix} 25 & 27 & 24 \\ 34 & 32 & 35 \end{bmatrix}$$

Select (1,3). CS′ = 24. After removing assigned rows and columns we get,

$$M = \begin{bmatrix} 12 & 22 \end{bmatrix}$$

Select (1,1) which corresponds to (2,1) in the original matrix. CS′ = 36.

Figure 4-8: Matching clusters - example 2: Both algorithms produce incorrect result. Assuming rows correspond to the A set and columns to the B set, the optimal assignments are: $A_1 \leftrightarrow B_1$, $A_2 \leftrightarrow B_2$, and $B_3 \rightarrow$ NULL. These assignments result in CS = 33.

declared that some whistles are to be removed. These removed whistles should not be counted as part of the distance between remaining clusters. The figure also includes a cumulative CS value labelled CS′. The final CS′ value, 10, is obviously the correct CS.

One reasonable question to ask about these algorithms is if they are the same. Figure 4-8 clearly shows that is not the case. Not only do both algorithms result in a different CS value, but both results are wrong. The correct CS for example 2 is 33.

It has been demonstrated that both "greedy" algorithms can fail to compute an accurate CS value. The underlying problem is that there are too many degrees of freedom in the problem. The optimal cluster assignments depend on two things, the amount of overlap between the clusters and the size of the clusters involved. Algorithm 1 only cares about overlap and can be made to fail by adjusting cluster size accordingly. On the other hand, algorithm 2 cares about cluster distance, a combination of size and overlap. However, it still can be made to fail. These failures leave us with no choice but to do take a "full search" approach. That is, try all cluster assignments and select the best.

Failure of the "greedy" algorithms was disappointing. However, the introduction of the R matrix allows us to look at the CS in a different way. In terms of R, the minimum number of whistles are rejected when the sum of the r_{ij} corresponding to each cluster assignment is maximized. The optimization problem is then, given an arbitrary matrix R, select p elements as to maximize the sum of those elements subject to the constraint that each row and column can be used only once. The value of p is either the number of rows or columns, whichever smaller. Unused rows or columns are assigned to NULL.

From this perspective, it is easy to see a case for which "greedy" algorithms do provide the correct CS. Since one can only pick one element from each row[5] to enter the sum, one can do no better than picking the maximum of each row. When such a selection is possible, the "greedy" algorithms do work. This observation ties in nicely with case a. Recall that in case a, a single cluster gets divided into multiple clusters. The R matrix for such case will have a single row. This makes selection of the maximum always possible. However, even for case b it will be possible some times to use a "greedy" algorithm. Therefore, case b will be divided into case b1 and case b2 depending on whether the "greedy" algorithm will work (b1) or not (b2).

Before summarizing the algorithm used in computing CS and discussing the results there is one more observation to be made. Consider the R matrix shown below.

$$R = \begin{bmatrix} 14 & 18 & 1 \\ 0 & 12 & 8 \end{bmatrix}$$

For this example the assignment $A_1 \leftrightarrow B_1$, $A_2 \leftrightarrow B_2$ results in the same CS as if the assignment $A_1 \leftrightarrow B_2$, $A_2 \leftrightarrow B_3$ is made. Given the possibility for such ambiguity, reference to which specific whistles are rejected is generally useless. In order for specific whistle information to make sense great care should be exercised in assigning clusters as to make sure one rejects the same whistles whenever possible. Such care

[5] Remember that there are at least as many columns as rows.

was not incorporated in the final algorithm since it is assumed that only the final CS value matters and not which whistles get removed to achieve it.

To summarize then, the algorithm used in computing CS was the following.

1. Initialize algorithm by setting CS = 0, assigning all clusters to NONE, and defining A as the set with the least clusters.

2. Compute intersection matrix $R = \{r_{ij}\}$ where $r_{ij} = \aleph(A_i \cap B_j)$.

3. Find the 1st unassigned cluster in set A. Form the interconnected set I described in page 109. Create the submatrix R_s using only those rows and columns in R which correspond to clusters in I. Transpose the submatrix if needed to have more columns than rows if possible.

4. Identify the case as one of the following:

   (a) Case a: A single row.

   (b) Case b1: Multiple rows but the maxima in each row occurs in different columns.

   (c) Case b2: Multiple rows but one or more of the maxima for each row occur in the same column.

5. Depending on case, find optimal cluster assignments. For cases a and b1 simply assign each row to the column where the maximum occurs. For case b2, try all possible cluster assignments and select the best one. Find CS′, the incremental CS value. Unassigned columns get assigned to NULL.

6. Let CS = CS + CS′. If there are unassigned clusters in A, go to step 3. Otherwise, we are done.

Now let us cover the results obtained in the human matching experiments. In the experiments, each subject was asked to manually classify the same 180 whistles. Whistles were given in completely random order. No additional information, such as

Animal	File	Whistle Number (in-file, in-experiment)
FB 163	90579AK1	(21, 97),(24,156),(27,23),(30, 46),(33,151),(36, 12),(39,21) (22,144),(25, 2),(28, 1),(31,166),(34,177),(37, 78),(40,48) (23, 84),(26, 47),(29,41),(32,170),(35,109),(38,128)
FB 92	905810K1	(21,13),(24,168),(27, 58),(30,145),(33,105),(36,161),(39,96) (22,32),(25, 38),(28,112),(31, 6),(34, 83),(37,140),(40,82) (23,95),(26, 80),(29,130),(32, 86),(35,113),(38, 35)
FB 50	90580AK1	(21,111),(24,110),(27, 65),(30, 89),(33, 53),(36, 20),(39,131) (22, 90),(25, 36),(28,135),(31, 94),(34, 42),(37, 8),(40, 93) (23, 85),(26, 56),(29, 76),(32,137),(35,114),(38,102)
FB 158	90588K01	(1,22),(4,120),(7,153),(10,148),(13,176),(16, 25),(19,125) (2, 9),(5,132),(8,178),(11,171),(14, 77),(17,149),(20,138) (3,79),(6,115),(9,180),(12,116),(15, 43),(18,179)
FB 161	905780K1	(1, 3),(4,40),(7, 33),(10,133),(13,134),(16, 75),(19,122) (2,172),(5,57),(8,108),(11, 4),(14,157),(17,143),(20,146) (3, 62),(6,49),(9,117),(12, 28),(15,107),(18,118)
FB 90	90580BK1	(1,29),(4,121),(7, 64),(10, 7),(13,104),(16, 14),(19, 99) (2,37),(5,101),(8, 87),(11,11),(14, 15),(17, 67),(20,167) (3,10),(6,164),(9,147),(12,88),(15, 60),(18,123)
FB 19	91537K02	(1,154),(4,106),(7,158),(10, 17),(13,103),(16, 61),(19,127) (2, 91),(5, 66),(8,139),(11,129),(14,136),(17, 44),(20,126) (3, 16),(6,124),(9, 45),(12, 55),(15,174),(18,173)
FB 23	91546K01	(1,100),(4,159),(7,141),(10,34),(13, 73),(16, 24),(19,81) (2, 26),(5, 50),(8, 72),(11,59),(14,160),(17,162),(20,54) (3, 98),(6, 69),(9, 39),(12,27),(15,155),(18, 74)
FB 183	905890K1	(1,175),(4, 31),(7,70),(10,163),(13,169),(16,150),(19, 5) (2, 68),(5,165),(8,63),(11, 92),(14, 19),(17,152),(20,119) (3, 51),(6, 30),(9,52),(12, 71),(15,142),(18, 18)

Table 4.9: List of whistles used in manual clustering experiments.

Subject	Number of Clusters	Maximum Cluster Size	Minimum Cluster Size	Number of Single Clusters
KF	20	21	1	9
PF	20	20	1	2
PT	32	20	1	8
LS	54	20	1	32

Table 4.10: Cluster set statistics per individual subject.

how many animals[6] were involved or how many clusters to make, was provided. Each subject worked on his/her own. The whistles used in the experiments are listed in Table 4.9.

In Table 4.10 some relevant statistics regarding the clusters assignments created by each subject are shown. Observe that no agreement exists with regard to the number of clusters present. On one hand, subject LS found a lot of differences between the whistles, resulting in many different clusters and a lot of single whistle clusters. While on the other hand, subject PF was perhaps more forgiving of whistle differences and ended up with fewer clusters and only two single-whistle clusters.

The computed CS and CS% values are given in Table 4.11. The CS values appear above the diagonal while the CS% values are below the diagonal. The best agreement was obtained between subjects PT and PF. Note that although the second best match is between subjects PT and LS, that does not make subject PF a good match to subject LS. No consensus can be formed among the subjects as to what are the "correct" cluster assignments for the given whistles.

Also in Table 4.11 information is given regarding the number of different cases found and the contribution of each to the final CS. By far the most common case was case a, that is, to have a cluster in one set subdivided into one or more clusters in the other set. The contribution to the CS for each case a was on the average only about

[6]Recall that, as mentioned in Chapter 1, the identity of the animals is known with certainty. However, none of this information was given to the subjects.

CS and CS% values

	KF	PF	PT	LS
KF	–	53	66	73
PF	29.4	–	36	52
PT	36.7	20.0	–	38
LS	40.6	28.9	21.1	–

Contribution to CS by case

match	Case a	Case b1	Case b2
KF vs PF	30	10	13
KF vs PT	23	28	15
KF vs LS	73	0	0
PF vs PT	17	3	16
PF vs LS	41	11	0
PT vs LS	20	0	18

Number of individual cases found

match	Case a	Case b1	Case b2
KF vs PF	9	1	1
KF vs PT	9	2	1
KF vs LS	20	0	0
PF vs PT	14	1	1
PF vs LS	18	1	0
PT vs LS	27	0	2

Table 4.11: Comparison of manual cluster assignments. - For each pair of subjects these tables show the CS value as well as the contribution of each case to CS. The last table shows how many cases were of each type.

2.3 whistles. On the other hand, cases b1 and b2 while much less frequent, had a CS contribution per case of around 11 whistles. Given the fact that the most common case was a, we can conclude that subjects mostly disagreed about whether a cluster was a single, large cluster or if it should be subdivided into smaller, non-overlapping clusters.

An interesting problem to consider is how well each subject distinguished one animal from another, that is, how many times a cluster contained more than one animal. For this problem, unlike the cluster classification problem, there indeed exist a correct solution. Information on which whistle corresponds to which animal was given in Table 4.9.

Table 4.12 shows how well each subject did for each particular animal. The table has 3 pieces of information for each animal and subject. First, column "C" tells how many larger-than-one-element clusters the animal's whistles were divided into.

Second, column "S" lists the in-file number of whistles assigned to single-element clusters. Third, column "W" lists the in-file number of whistles assigned to the wrong animal as well as which wrong animal (in parenthesis) the whistle was assigned. A blank entry in the second or third column of a subject indicates that no whistles fit the category.

In general, the subjects did a good job in recognizing the individual animals. The least amount of errors were made by subject LS, but that was at the expense of lots of single-whistle clusters. Subject KF did a better job, only 3 errors with a reasonable amount of single-whistle clusters. Perhaps this subject tried to assign clusters based mostly on separating the different animals. This strategy could be responsible for the large CS values between KF and other subjects.

Considering the individual animals several observations can be made from the table. First, animal FB 158 was so distinct that no whistles were taken or added to the cluster. Second, other very distinct animals were FB 161 and FB 19 but not as perfect as FB 158. Third, whistle 20 of FB 90 and 32 of FB 50 were misclassified by all subjects. This is an strong indication that the whistles may indeed be different from the normal animal's whistle. Fourth, when an animal's whistles were divided into many clusters, the number of clusters made was kept small. However, this tendency may be driven by having only 20 whistles per animal and the subjects' idea of what a reasonable cluster size was.

Animal	KF			PF		
	C	S	W	C	S	W
FB 163	1	26,31,32,33, 35		3		26,32,35 (FB 23)
FB 92	1	30		2		30 (FB 183)
FB 50	1		32 (FB 92)	3		32 (FB 92)
FB 158	1			1		
FB 161	1			1		
FB 90	2	5	20 (FB 23)	2	4	20 (FB 23)
FB 19	1			2		
FB 23	1	3		2		
FB 183	2	7	8 (FB 23)	2	7	15,20 (FB 19)

Animal	PT			LS		
	C	S	W	C	S	W
FB 163	6		26,32,35 (FB 23)	3	24,26,28,29, 30,31,32,33, 34,35,39,40	
FB 92	2	30		4	30,37	
FB 50	3	30	32 (FB 90)	3	29,30,32,36	
FB 158	1			1		
FB 161	1	1		1	1	
FB 90	2	4,5,7	20 (FB 23)	2	2,4,5,6,7	20 (FB 23)
FB 19	2			2		
FB 23	4		20 (FB 90)	3	3	
FB 183	4	7,20	8 (FB 23) 15 (FB 19)	3	2,7,8,9, 14,15,20	

Table 4.12: Manual clustering for animal identification. - Evaluation of how well the different subjects did in separating one animal from another. In the table, C = number of larger-than-one-element clusters whistles were divided into; S = in-file number of whistles assigned to single-element clusters; W = in-file number of whistles assigned to the wrong animal. Which wrong animal a whistle was assigned to appears in parenthesis.

Chapter 5

Level III Compression - Detecting Repetitions

Once a coding space has been found, all records (i.e. signals or, for the specific application studied, whistles) in a database are represented by points in this coding space. The compression factor achieved by level III comes from this conversion from trace to a point in multi-dimensional space. The typical (incremental) compression factor achieved in level III is about 1.

To detect a repetition for a particular record, one takes that record (called the target) and searches the space to find points which lie close to it. If the distance between the target and its nearest record is smaller than a given distance threshold, then it is said that a match has been found. This distance threshold is set by the user and it should be set somewhere near the typical cluster radius.

The core of the level III system is the technique used to search the coding space. The straightforward approach of computing the distance from each point in the database to the target, quickly becomes impractical as the size of the database grows. A better, more efficient technique than an exhaustive search of the space record by record is necessary. This chapter concentrates on the woık done in searching for

efficient techniques to find the m-nearest neighbors to the target. The value of m is specified by the user and can be changed with every search.

One thing to note is the fact that instead of restricting the search space using the distance threshold and looking for points in it, one searches the entire space for the nearest neighbors. Searching only the area of the space where a repetition could be, can be considered a simpler search problem. However, if a match is not found, one can not say anything else. More important, how can one tell which records belong in this restricted subspace without having to compute the distance of every record to the target? Evidently, this simpler approach is nothing more than another exhaustive search.

On the other hand, a more efficient search of the entire space allows one to report on which signal in the space is closer to the target signal even when a match (repetition) is not found. The system developed for handling the database incorporates this feature.

In the next section some of the alternatives for efficient searching techniques are covered. The two most promising techniques, however, are only touched upon briefly in this first section. Full details of each one are given in a separate section. Once all the necessary descriptions have been given, then a discussion regarding the experimental results obtained is represented in the last section.

5.1 General Background

The problem of nearest-neighbour searching has been studied in other disciplines such as information storage and retrieval, database management, and computer science. The idea is to retrieve the best m matches to a target as efficiently as possible. Efficiency is most often measured by the number of records that need to be examined before finding the m nearest. When m is equal to 1, the algorithm returns the record in the database which is closest to the target.

The different methods found in the literature will be discussed in this section. It was found that these methods seem to fit several broad categories. Lacking any other logical way of arranging this material it was decided to explain methods by category. As expected, many methods do not match perfectly the application at hand. As part of the discussion below there are also comments on such discrepancies.

Let us now establish some notation. Each record in the database is characterized by a set of numbers. How many numbers are used depends on the chosen dimensionality, d, of the space. Thus, as said previously, one can think of the record as a point $\underline{x}$ in d-dimensional space. In vector form, $\underline{x}^T = [x_1\ x_2\ \cdots\ x_d]$. The target record will be $\underline{w}^T = [w_1\ w_2\ \cdots\ w_d]$. The best-match problem is finding the best m matches to $\underline{w}$ as efficiently as possible.

In order to define a match, some kind of distance function must be defined. It is commonly required that the distance function be also a metric for the space. The standard metric distance used is

$$D(\underline{x},\underline{w}) = \left[\sum_{i=1}^{d} |x_i - w_i|^p\right]^{1/p} \tag{5.1}$$

The most common values for p are 1 (city block distance), 2 (Euclidean or L_2-norm distance) and ∞ (maximum coordinate distance). Of these three, $p = 2$ was selected for our coding space and has already been used in the previous chapter for computing the distance between cluster centroids. Also from the previous chapter, recall that dimensions are normalized to zero mean and unit variance in order to avoid allowing a single dimension to dominate the distance calculation.

After spending some time studying the retrieval problem I have become convinced that, in absolute terms, only a full search of every record in the database can reveal which ones are the m one wants. One then wonders what is the difference between all these methods and a simple full search of the database. The answer is that better algorithms "hide" information about the elements of the database in their structure.

This "hidden" information allows us to remove entire sections of the database from consideration quickly. This will become clearer as the several methods are discussed.

Let us begin with a one-line description of the categories used for classifying the different algorithms.

1. full search - Exhaustively check target against every available alternative.

2. adaptive buckets - Methods which directly partition the space into buckets that change size as the search progresses.

3. hash functions - Use a hash function to partition the coding space. The idea is for the hash function to place close records into either the same bucket or adjacent buckets.

4. metric based - Use the triangle inequality to quickly reject large portions of the database from consideration.

5. trees - Use a tree for storage of database records. There are several alternatives in this category. Also, some of the methods in other categories can be posed in a "tree-like" fashion.

6. clustering - Partition the space based on the creation of clusters. Find best match by finding which cluster is closest. Since there are fewer clusters than records (in general), the method requires fewer comparisons. However, this method requires a manual identification of every record in the database or an expensive clustering procedure in order to define the clusters. It was decided not to pursue this approach. Also, given the simplicity of this idea, a separate explanation section was deemed unnecessary.

5.1.1 Full Search Methods

This is the most straightforward and computationally expensive solution to the nearest-neighbour retrieval problem. Assuming that there are N records in the database, this

method requires N distance computations followed by a sort and then selection of the m nearest records. Time requirements for this algorithm are $O(N)$.

By far the most computationally expensive part would be the computation of all distances. Assuming the use of Euclidean distances, each distance will require about d multiply-adds. The higher the dimensionality, the higher the computational costs. Computational requirements are therefore $O(dN)$. Because costs grow linearly with database size, this approach is not recommended except for very small databases.

One approach that may be used to still do a full search but which requires less time and computation is to partition the space into buckets. For the search, one takes the target record and first finds the bucket in which the target belongs. Then, compute distances only to the other records in the same bucket. Next, to examine the possibility of finding a nearest neighbor in an adjacent bucket, one finds the closest bucket boundary to the target record and searches the records in the next bucket. Of course, the target record may be close to more than one boundary, in which case additional buckets will have to be searched. There is even the possibility that no records will be found in the bucket for the target record or any of the adjacent buckets. In such case one can extend the search using the boundaries of the adjacent buckets.

It is certainly possible to use buckets of any shape in the scheme described above. However, to simplify implementation let us assume that buckets are created by independently partitioning each dimension. Each bucket is then a rectangular section[1] of the space. The independent use of each dimension simplifies not only finding the bucket for the target record but also computing the distance to the bucket boundaries. The specifics regarding what constitutes being close to a boundary have been purposely left undetermined at this point[2].

[1]Other rectangular partitions of the space are possible but they do not allow for dimensions to be treated independently.

[2]One idea considered is: if the nearest distance from the center of the bucket to the boundary is x, one can decide to search across the boundary if the target record is less than a distance cx, $0 < c < 1$, from it.

The computational costs involved in this approach are highly dependent on the number of records in the buckets and the number of buckets searched. The number of buckets to be searched range from 1, when the target is not close to any boundary, to $2^d - 1$, when the target is near a corner of the bucket. The number of records per bucket depends on the (unknown) record distribution over the space and the shape and position of the buckets. It is reasonable to desire buckets of approximately equal numbers of records but this may not be possible given the scheme chosen for creating the buckets.

Let us say we use B buckets and managed to get N/B records per bucket (allow fractions of a record for now). Then, assuming only adjacent buckets need to be searched, in the best case one will only need to compute N/B distances. In the worst case, $(2^d - 1) * N/B$ distance are computed. As long as $B > 2^d - 1$ and the cost of finding the distance of the target record to the boundaries is negligible, the bucket idea will result in less computation than the straight full search. In order to make $B > 2^d - 1$, each dimension needs to be partitioned more than once. Also, if the m nearest records are not found after searching the adjacent buckets to the target record, costs will be larger.

Let us now consider an extension to the bucket idea. Namely, let all records in a bucket collapse into a single point at the center of the bucket. This is equivalent to discretizing the record positions so that they only lie at specific values. Having done this, one can no longer talk about the m closest records since real distance information has been "eliminated".

What one gains by such a step is a reduction of the number of bits necessary to identify the record. If there are B buckets, then only $\log(B)$ bits[3] are needed per record (as opposed to $32d$ bits needed for a floating-point representation of each point). Also, there is no need now for distance computations. Since only B different positions are allowed in the space one can just compute the $B * (B - 1)/2$ distances

[3] $\log(\cdot)$ stands for the base 2 log.

beforehand and store them in a lookup table. Of course, if the number of buckets is too large, the lookup table idea may be impractical.

However, these gains do not come without a price. First, as mentioned above, the real distance between records is lost. For records lying close to a boundary it is even possible that the closest record will lie outside the bucket. Granted, it is unlikely that such a situation will occur but a misidentification of the target record is possible.

Second, the target record will be equidistant to all the boundaries. This will result in an increase of the number of buckets to be searched. That is, if the bucket that contains the target record is empty or one wants more matches than the bucket offers, then one has to search across all boundaries. However, when no distances need to be computed, this may not be a big problem.

Third, the distinct potential vocabulary the coding space can hold is reduced to the number of buckets in the space. To have a vocabulary on the order of 10^9 signals (whistles), one needs at least 32 bits to represent each bucket. Although such a representation is not impractical, having such a large number of buckets is. The large number of buckets would prevent the use of a lookup table to compute distance. Besides, most buckets will be empty, which indicates a wasteful representation.

The search process in this single-point bucket strategy is to first find the bucket the target record belongs in, and if unsatisfied with the number of records there, then search across all boundaries into nearest buckets. In the unlikely event that m records have not been found after searching the nearest buckets, the search has to be expanded. Note that in this strategy, once buckets are defined, a number is assigned to each and records are wholly identified by bucket number.

One important thing to mention here is that this strategy does not really require rectangular buckets. Thus, buckets may be defined in a completely different way and all that would change is the algorithm for finding out which bucket a record belongs to. This algorithm is trivial when dimensions are treated independently and

therefore its cost has not been considered so far. If the buckets used required an expensive record-to-bucket algorithm, this searching approach would be less appealing.

Nevertheless, having a search technique that sacrifices distance information for speed was not found appealing because distance is the fundamental quantity used for identifying the signals. Although there are ways in which one could preserve distance, better methods were found. Thus, none of these full search methods were implemented.

Before closing this section, note that it began with full search of *records* and ended with a full search of *buckets*. One has to keep in mind that searching buckets is beneficial only when the number of buckets to be searched is less than the number of records in the file. If because of the dimensionality of the space, the number of adjacent buckets to be searched is much bigger than the total number of records, it may be better to search the records instead. This is unaffected by whether the records maintain their "true" distance information or not.

5.1.2 Adaptive Buckets

In an adaptive bucket strategy the idea is to use a bucket of variable size to determine the m-nearest neighbors to the target. One algorithm which fits under this category is the one developed by Yunck [41]. In his algorithm, Yunck finds the m nearest neighbors by counting the number of elements inside a variable-size hypercube (bucket) for which the center is the target record.

The size of each side of the hypercube is set initially to l, where l is set by the user. If not enough records ($< m$) are inside the bucket, the value of l is increased appropriately. If the number of records inside the hypercube is too large, the size of l is reduced. Exact details on how new values of l are computed are not provided in [41]. After changing l one counts the number of records inside the bucket again and repeats the process until only m records are found.

When $p = \infty$ in the distance definition, this search strategy does not require any distance computations to find the m-nearest neighbors. All the searching is done with comparisons (logical operations). For other values of p, some distances need to be computed. The expected number of records that need to be examined is proportional to m, and the proportionality constant depends on the distance metric, p, and the dimensionality of the space d.

A second method which falls under this category is the expanding bucket approach. This approach works similarly to that described by [41] but the bucket always grows. Expansion is controlled by the data in the database. Also, the method is better suited to the $p = 2$ metric (see Equation 5.1). Since the expanding bucket is one of the methods implemented for this thesis, a full description of it is given in Section 5.3.

5.1.3 Hash Functions

In hashing one also divides the coding space into buckets. However, the buckets do not cover continuous regions of the space. Instead, the hashing function is used for record-to-bucket transformation. As expected, this offers a large degree of freedom with respect to how buckets are defined.

Given the goal of minimizing the search time for finding the closest records to the target record, one can speculate on what the ideal hash function should be. It seems logical to require a hash function that places each record and its m nearest neighbors in the same bucket. However, such a function is of no use since it may end up placing every record in the same bucket.

Along a similar line of reasoning, H. Du and R. Lee [7] develop a multi-key hash function based on Gray code. Their approach guarantees that "the record stored at location k and the record stored at location $k+1$ will be nearest neighbors". However, since they only store one record (whistle) per location (bucket) their approach only provides one near neighbor of many. Because of the use of a single record per location,

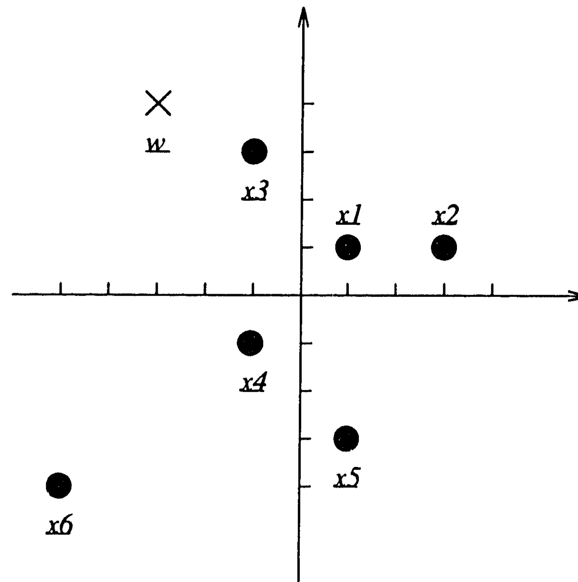


Figure 5-1: Example used to illustrate metric-based method.

this approach basically amounts to a linear sorting of the records in the space in such a way that each record is "close" to the ones adjacent to it in the list.

Considering that Grey code hashing seems to be nothing more than sorting and that the "ideal" hash function is useless, there is little hope for the use of hashing as an effective solution to the problem addressed by this thesis.

5.1.4 Metric Based Methods

The work presented in this section is based on the ideas discussed by D. Shasha and T-L. Wang in their paper [36]. Details on the algorithm's implementation can be obtained from the reference. How the method works is illustrated by the example below in which the nearest ($m = 1$) neighbour to a target is found.

Consider the six-record database shown in Figure 5-1. The target record for which the nearest neighbors need to be found is labelled $\underline{w}$. The Euclidean distance metric is used in this example. The position of each record is given; $\underline{x1} = (1,1)$, $\underline{x2} = (3,1)$, $\underline{x3} = (-1,3)$, $\underline{x4} = (-1,-1)$, $\underline{x5} = (1,-3)$, $\underline{x6} = (-5,-4)$, and $\underline{w} = (-3,4)$.

The method requires the existence of at least one reference record in the database to which distances from all other records have been computed. Choosing $\underline{x1}$ as the reference leads to the distances given in the second column of Table 5.1. Note that all these distances are computed beforehand and only once.

With the above distances in place one is ready to begin finding the best match to record $\underline{w}$. The process is as follows. Let ξ be the value of the current minimum distance to $\underline{w}$. First, choose a record in the database and compute its distance to $\underline{w}$. How the record is chosen is based on a heuristic described below. If this distance is smaller than ξ, replace the ξ value; otherwise, keep the old ξ value. Second, update all bounds for the distance between $\underline{w}$ and each record in the database using the new information. Third, based on the bounds and ξ determine if there are any records which may be removed from consideration. If only m records remain after removal, stop. Otherwise, go back to the first step and repeat.

When the algorithm is started, the heuristic described in [36] says to simply pick a record at random from the database. Assume that the selected record is $\underline{x4}$. The value of $D(\underline{x}4, \underline{w}) = \sqrt{29} = 5.39$ where $D(\cdot, \cdot)$ is defined in Equation 5.1 with $p = 2$. Since no ξ value exists yet just let $\xi = 5.39$.

For each record one must now compute the distance bound between the record and $\underline{w}$. This bound is computed using the triangle inequality on each possible path between the two records. A path is made out of connections between records whose distance is known exactly. Since several paths are possible, several candidate values exist for the desired bound. The largest candidate is the one that must chosen as bound because the actual distance from the record to $\underline{w}$ is larger than all of these candidates. The fifth column of Table 5.1 shows the value of this bound for each record, as well as the path used in each iteration of the algorithm. Note that the path is indicated as a,b,c where each letter is replaced by the id of the record.

The negative bounds are of little help by themselves. However, they contain information about the relative potential of the records to be the closest one to $\underline{w}$.

whistle	$D(\underline{x}i,\underline{x}1)$	iter	path	bound	ξ
x1	0	1	x1, x4, w	2.56	5.39
		2	x1, x5, w	3.14	5.39
		3	x1, x5, w	3.14	5.39
		4	x1, x5, w	3.14	2.24
		5	Eliminated		2.24
		6	Eliminated		2.24
x2	2	1	x2, x1, x4, w	0.56	5.39
		2	x2, x1, x5, w	1.14	5.39
		3	x2, x1, x5, w	1.14	5.39
		4	x2, x1, x5, w	1.14	2.24
		5	x2, w	6.71	2.24
		6	Eliminated		2.24
x3	$\sqrt{8}$	1	x3, x1, x4, w	−0.27	5.39
		2	x3, x1, x5, w	0.31	5.39
		3	x3, x1, x5, w	0.31	5.39
		4	x3, w	2.24	2.24
		5	x3, w	2.24	2.24
		6	x3, w	2.24	2.24
x4	$\sqrt{8}$	1	x4, w	5.39	5.39
		2	x4, w	5.39	5.39
		3	x4, w	5.39	5.39
		4	x4, w	5.39	2.24
		5	Eliminated		2.24
		6	Eliminated		2.24
x5	4	1	x5, x1, x4, w	−1.44	5.39
		2	x5, w	7.14	5.39
		3	Eliminated		5.39
		4	Eliminated		2.24
		5	Eliminated		2.24
		6	Eliminated		2.24
x6	$\sqrt{61}$	1	x6, x1, x4, w	−0.40	5.39
		2	x6, x1, x4, w	−0.40	5.39
		3	x6, w	8.24	5.39
		4	Eliminated		2.24
		5	Eliminated		2.24
		6	Eliminated		2.24

Table 5.1: Steps in metric based method example.

Using the bounds obtained one then looks to see if any record may be removed from consideration. In the example, no bound is larger than the current ξ and therefore no record can be removed.

Repeating the process one has to again select another record from the database. According to the heuristic in the paper, the record to select should be the one with the smallest bound. The reason for picking the record with the lowest bound is that it has the best chance of being the closest one to $\underline{w}$. For the example, one should select $\underline{x5}$ which has a lower bound of -1.44. The distance from $\underline{x5}$ to $\underline{w}$ is $\sqrt{51} = 7.14$ and is larger than ξ. Therefore, ξ remains unchanged. The distance bounds are updated to the ones shown in iteration 2 of Table 5.1. The only record that can be removed is $\underline{x5}$.

The record with the smallest bound at this stage is $\underline{x6}$. Once the distance from $\underline{x6}$ to $\underline{w}$ is computed, $\sqrt{68} = 8.25$, the updated distance bounds are the ones shown in Table 5.1, iteration 3. The entry for $\underline{x5}$ indicates the record is no longer under consideration. The value of ξ stays the same once again. Unfortunately, only one record ($\underline{x6}$) may be removed. The record with the smallest bound now is $\underline{x3}$.

Distance from $\underline{w}$ to $\underline{x3}$ is $\sqrt{5} = 2.37$. This distance is smaller than ξ therefore we let $\xi = 2.37$. After the bounds are updated one can see that $\underline{x4}$ and $\underline{x1}$ may be eliminated. Since the bound from $\underline{x2}$ to $\underline{w}$ is still below ξ one has to go ahead and compute the real distance. The result is that $\underline{x2}$ is further away from $\underline{w}$ than $\underline{x3}$.

The entire process has correctly determined that $\underline{x3}$ is the closest record to $\underline{w}$ at a distance of 2.37. A total of 5 distances had to be computed. For this specific example one only saved 1 distance computation over a full search. In general, the number of distances required to find the nearest neighbors is highly dependent on the initial record chosen by the algorithm. Choosing $\underline{x1}$ instead of $\underline{x4}$ at the start would have resulted in only 3 distances needed for finding closest match. The authors of [36] do not offer a better heuristic for choosing records than the one used here. They do note the dependency of performance on how lucky one is with one's choices.

Having no guaranteed reduction in the number of distances to be computed is a problem but not an extreme one. After all, the worst case is when the distance to all records need to be computed, which means this method is always better than a full record search. As new distances are computed, more paths are available to $\underline{w}$ and better bounds are obtained. Therefore one would expect the number of distances to be computed to go down as records are added to the database.

However, keeping track of all computed distances has the potential of developing into an storage problem. A database of 2048 elements would require 8 Mbytes of storage (for the distance between all pairs) assuming distances are kept as single-precision floating point numbers. One possible solution to the growth in storage is to simply not store any new record-to-record distances. Instead, just store the distance from the new record to the reference. This makes storage $O(N)$ instead of $O(N^2)$, allowing use of larger databases but making the search slower.

Another potential problem is a direct result of using the triangle inequality for bounding distances. Basically, reflections about any point in the path connecting 2 records would result in the same distance bound. For example, $(-1,-1) \rightarrow (0,0) \rightarrow (1,1)$ results in the same bound as $(-1,1) \rightarrow (0,0) \rightarrow (1,1)$, however one end is closer to (1,1) than the other. This problem only gets bigger as the dimensionality of the space increases because the number of possible reflections increases.

Before ending this section and dismissing the metric approach, let us discuss some ideas on how to work around the memory and reflection problems. For the memory problem the only reasonable solution seems to be to cut back on the amount of data stored and, as discussed above, not store any new record-to-record distances. Implementation of this idea will force some modifications to the algorithm presented in [36]. However, since less distances are available, the distance bounds will be weaker and search time will probably be longer.

For the reflection problem one approach may be to modify the heuristic which selects records so that it uses information about which "quadrant"[4] with respect to the reference the record lies in. This information can be computed every time it is needed. That is, let the distance bound (ξ) be used to rank records but instead of going ahead and computing the distance to the record with the lower bound immediately, first use all records that lie in the same quadrant as the target. Once all records in the same quadrant have been used, the rest of the records can be used in the order dictated by the distance bound for each.

The computational cost of finding which quadrant a record is in is assumed to be small compared to a distance computation. However, if one has to go through the entire record list only to discover that no records lie in the same quadrant, it would have been less expensive to not consider quadrant information at all. It is certainly possible to specify beforehand how many records in the list it is worthwhile to examine before giving up on the search for records in the same quadrant. Thus, one may establish a heuristic, based on 1) the relative computational expense of distance computation to quadrant determination and 2) the distance bound value (ξ), to determine when to stop searching the list.

It should be noted that the quadrant information is just an aid for obtaining better distance bounds. It is entirely possible for the closest record to not even be in the same quadrant as the target record. Thus, attempts to search *exclusively* the quadrant in which the target record lies should be discouraged. Eventually, for every record not removed from consideration by just its distance bound an exact distance will be computed regardless of quadrant.

Several variations on the "quadrant information" idea are possible. A limit has already been proposed introducing on how many candidate records to check before going back to the heuristic of using the record with smallest distance bound. Another

---

[4]A quadrant is defined as one of the multi-dimensional subspaces created by partitioning each dimension at the location of the reference.

variation is to precompute quadrant information instead of incurring the computational expense of finding in which quadrant a record lies. Certainly, memory will remain $O(N)$ even after storing "quadrant from" along with "distance to" reference for each record. But once quadrant information is available for all records one can sort records by quadrant and look for where the minimum bound occurs in the quadrant of interest. By-quadrant sorting is done only once, compared to having to sort the record list by distance bound at each iteration. The case where no records are in the same quadrant as the target record can be quickly identified and the alternative strategy of finding the overall lowest bound used.

Unfortunately, running the example again using quadrant information resulted in no gain in terms of the number of distances computed. This seems to result from the quadrant information not affecting the number of records eliminated but rather the order in which they are eliminated. It seems that the distance to a record that looks close because of the reflection problem will have to be computed in order to eliminate the record.

There is another way in which quadrant information may be used. One can obtain a, perhaps better, distance bound using the distance to the boundaries of a quadrant. Obviously, any record that is not in the same quadrant as $\underline{w}$ must be farther away than the distance to the common boundary between the quadrant the record is in and the quadrant $\underline{w}$ is in. Nevertheless, computing the distance to each of the $2^d - 1$ other quadrants is not the way to go. Such an approach will provide no savings in the number of computations for high-dimensionality spaces.

Instead of computing an exact distance to each quadrant one can just compute a lower bound. That is, find a lower bound to the distance to $\underline{w}$ from all records in a particular quadrant. This lower bound must be very inexpensive to compute in order to avoid contributing to the bulk of the computations. A very simple bound is given by the distance to the boundary when $p = \infty$ (see Equation 5.1). In terms of computations all that is required is the 1-D distance to the reference from $\underline{w}$, a total

of d "adds". Then, for each quadrant one just has to determine which dimensions are in "disagreement" with the quadrant $\underline{w}$ is in. The largest 1-D distance among those dimensions in disagreement is the minimum bound for all records in the quadrant.

Using the improved bounds in the example resulted in only 2 distance calculations needed to select $\underline{x3}$ as closest. Despite the improvements obtained with the improved distance bounds, this method was not selected for implementation. The main problem with keeping track of quadrant information is how the number of quadrants grows with the dimensionality of the space. For the 16-D coding space proposed in this thesis, there are $2^{16} = 65536$ quadrants.

There is also the possibility of having more than 1 reference record. However, although more reference records may help speed up the search, they also aggravate the problems mentioned above.

To summarize, storing quadrant position along with distance to reference has been shown to have the potential to reduce the number of total distances necessary in order to find the closest record. To realize this potential, the heuristic for selection of records must be biased towards the records in the same quadrant as the target record. Also, the quadrant information is used to improve on the lower bounds provided by the triangle inequality without significantly increasing computational costs. It has been observed that storage costs may be quadratic if every distance between records is preserved. Keeping only the distance to the reference helps reduce this problem. However, ultimately, the storage costs coupled with the algorithmic complexity (keeping track of all possible paths) prevented the implementation of this algorithm.

5.1.5 Trees

In general terms, a tree is a collection of nodes and edges such that there is one and only one path on these edges connecting any pair of nodes [37]. When a distinguished node, called the root, is added one obtains what is called an oriented tree. This allows

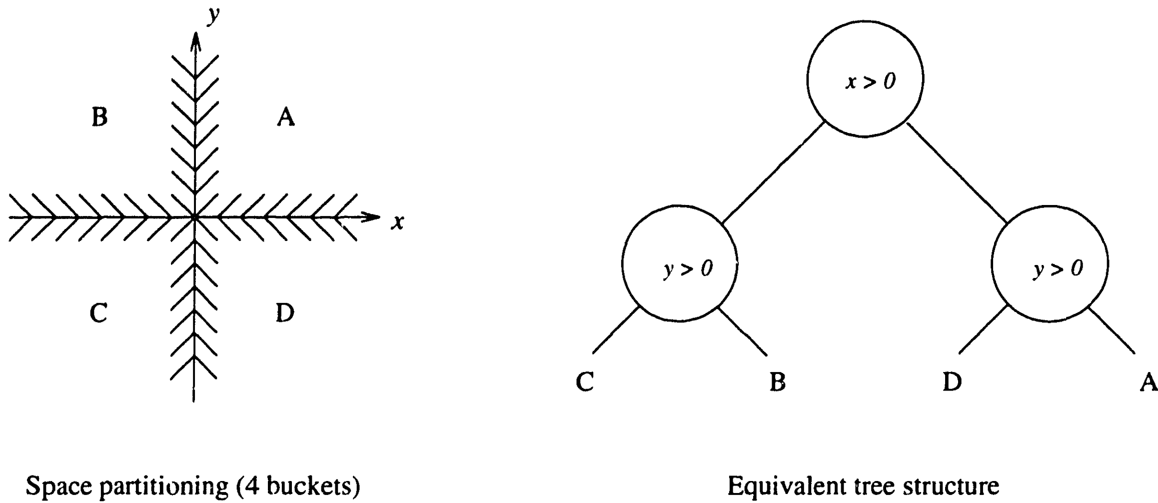


Figure 5-2: Tree representation of a coding space partitioning.

to specify a direction on the edge connecting the nodes as either inward (toward the root) or outwards (away from root). A tree where each node has only two edges coming out of it is called a binary tree.

Several kinds of binary tree have been developed. An extension to the binary tree, the B-tree [5] has been used extensively in database applications for its advantages in reducing search time from $O(N)$ to $O(\log N)$. Of the several variations on binary trees the one best suited to our problem is closely related to B^+-trees. In this kind of binary tree, all records are stored at the leaves. The inner nodes of the tree are used for navigation. That is, to quickly go from the root to the desired leaf. In a sense, a B^+-tree describes a different way of partitioning the d-dimensional space into buckets where records reside. Figure 5-2 shows an example.

Binary trees are used by Weiss [39] as part of both a deterministic and a probabilistic search algorithm. All records are stored at the leaves of the tree. When doing a search, the leaf where the target belongs is checked first. Getting to that leaf involves taking one of two branches for the nodes visited as one travels from root to the leaf. In the internal visited nodes of the tree one stores information regarding the branch not taken. After examining all records in the leaf where the target belongs,

the node information is used to guide the search into other leaves. Entire branches of the tree may be removed from consideration based on the information stored in the node.

The information stored in the node depends on the specific algorithm. For the deterministic algorithm, the node stores a "priority value" which is simply the minimum similarity between a record in the branch not taken and the target. The algorithm uses a similarity function in much the same way as the distance function of Equation 5.1. However, the similarity measure in [39] is best suited to non-numerical data, that is, records where each dimension represents a property the record may have, and the values the dimension may take are just simple binary terms like "yes" or "no". In order to apply the deterministic algorithm to my work, the similarity function and priority values would have to be adapted to fit the definition of distance used in this thesis. However, as mentioned in [39], the deterministic algorithm is not very efficient. Only a gain of about 10 % over a full search was reported using a database of 1400 records.

In the probabilistic approach of Weiss, instead of computing a single priority value for a branch, a set of priority values, along with the probability of each is computed. The jth value in the priority set corresponds to the priority (similarity) of a record with exactly j dimensions in common. For the probability set, the jth value corresponds to the probability of having j or more dimensions in common. Also, the user can define a stopping criterion based on the expected value of the number of records to be explored that can probably be in the set of m nearest neighbors.

When the stopping criterion is zero, that is, when no records will be left unexamined if there is a chance it may be a nearest neighbour, the probabilistic algorithm does not seem to perform any better than the deterministic one. As more errors are tolerated, then fraction of the database that need to be explored is reduced accordingly. For example, for an expected error of 1 record, only 45 % of the database needs

to be searched. Unfortunately, the probabilistic algorithm can not be modified to accommodate a continuous similarity function such as distance between records.

Another tree-based search technique is the $k-d$ tree method. This method has been implemented as part of the database handling package for level III. A complete discussion is given in Section 5.2.

5.2 $K-D$ Tree Implementation

The $k-d$ tree is a data structure developed by Friedman *et. al.* [9] for finding the m nearest neighbors to a record in a database. The search algorithm used in conjunction with this structure requires only O(log N) time, where N is the database size. Unlike the tree method described previously, the $k-d$ tree method can use the distance between records as a (dis)similarity measure. In the remainder of this thesis, the term $k-d$ tree method will be used to refer to the combination of the $k-d$ tree data structure and the algorithm used to conduct the search based on the information in the data structure.

This section will describe the $k-d$ method as described in [9] first. The description includes how the $k-d$ tree is constructed and the algorithm used to search it. Then the modifications made to the method so that it could be applied to the specific application considered in this thesis are covered. Along with the modifications, an assessment of their impact on performance will be given whenever possible.

The $k-d$ tree is another generalization of the binary tree used for sorting and searching. Each node in the tree represents a partitioning of the coding space. The tree "begins" at the root node which represents the entire space. For each node, there are 2 *sons*. The partition at each node creates two subspaces, one for each son. A node without any sons is known as a terminal node or leaf. The information in each node consists of the dimension along which the subspace is being partitioned, along with the value (threshold) used for partitioning.

The work in [9] concentrated on how to optimize the parameters of the $k-d$ tree to make the search the most efficient possible. The parameters of the tree are; the number of levels in the tree, the dimension to partition at each node, and the value of the threshold (position of the partition). The optimal value of these parameters is given below,

- number of levels - As many levels as necessary in order to make the subspaces associated with the terminal nodes have 1 record each.
- dimension to partition - For the subspace to be partitioned, find the dimension with the largest spread. This dimension is selected since it minimizes the chance of a record lying close to a boundary. In the reference, the spread of a dimension is defined as the range of the values for the dimension. Other definitions are possible.
- value of threshold - The partition is divided at the median of the data along the dimension being split. This decision is based on the fact that information from a binary decision is maximized when both alternatives are equally likely.

It should be mentioned that although, theoretically, the terminal nodes should create subspaces of 1 record each, in practice having more than one record leads to better performance (Figure 4 in [9]). Thus, the number of records per terminal node is seldom restricted to one.

The procedure for creating the optimal $k-d$ tree is the following. Starting with the entire space (the root node), find the dimension with the largest spread and determine the median value of the records along that dimension. Using the dimension and threshold found, divide the space into 2 subspaces. For each subspace (son node), repeat the operation. The process continues, adding more levels to the tree, until the desired number of records per terminal node is reached. As can be expected, the construction requires a static database. Otherwise, a new tree must be created with every change in the database.

The search procedure is a recursive procedure which starts at the root node. If the node being considered by the procedure is a terminal node, then all the records in the specific subspace[5] are compared to the target record (i.e. the distance to the target is computed for each). Assuming one is looking for the m nearest records, then the m smallest distances and the records they correspond to are stored. The distance to the m record, the largest of them, is stored in a separate variable also. If not enough records are available (less than m), the stored distance is ∞. If the node is not terminal, the recursive procedure is called with the son node corresponding to the subspace containing the target.

There are two tests used to determine when to stop the search. If the search is not stopped, it continues until all records have been examined. One test is called "ball-within-bounds" and it is performed every time a terminal node is reached. The ball the test refers to is one centered at the target and of radius equal to the distance between target and the mth nearest neighbour encountered so far. The test returns true if the ball is completely enclosed in the subspace corresponding to the terminal node and false otherwise. A positive (true) response ends the search.

The second test is called "bounds-overlap-ball" and it uses the same ball defined above. This test is done every time the recursive procedure returns and it is used to decide if a son node that was not considered earlier should be examined. Recall that in the search procedure, when the node is not terminal, the procedure is called again with the son node corresponding to the subspace where the target is[6]. After returning to the parent node, one must consider if the other son may contain a record close enough to be a nearest neighbour. What the test does is examine if the ball overlaps the bounds (i.e. edges) of the other son-node subspace. That is, the test examines if the branch ignored earlier should be considered at all. If the test returns

[5]Recall from page 137 and Figure 5-2 that the binary tree can be viewed as a specific partitioning of the multi-dimensional space. Each node can then be viewed as identifying a different subspace.

[6]Of the 2 branches available, the most likely one is considered first.

true, the search procedure is called with the other son node. If false, the search continues with other nodes if available.

Having covered the details of the algorithm as described in [9], now the discussion will focus on what was actually implemented for level III. During implementation, the search algorithm was kept exactly the same as described in the reference. All changes were made only to the $k-d$ tree construction algorithm. The changes were necessary because the SNR of Equation 4.1 was used as the definition of spread. The reason for using SNR was that it is a better indicator of the relative worth of each dimension than the range of values the dimension may take.

Even after deciding to use SNR as the measure of spread, the construction algorithm in [9] could have been used if the SNR were as easy to compute as other spread measures. However, to compute SNR one needs a database of manually clustered data. This manually clustered data is infeasible for large databases. The same 92-whistle (record) database used in the previous chapter (see Table 4.2) was therefore used to find the SNR values to create the $k-d$ tree. As a result, the tree is static, it does not change as the records in the database change. Note that recomputing the tree every time the database changes is computationally very expensive. Thus, a static tree, albeit extreme, is not such a bad idea.

One consequence of having a static tree is that the threshold for each node (the value used to split the dimension) will no longer correspond to the true median. As the records in the database change, the threshold will be just a random number near the true median according to the degree to which the database used in computing the threshold matches the statistics of the current database. In addition, the threshold does not depend on the specific subspace being partitioned. That is, every node that partitions dimension x will partition it along the same point, regardless of the subspace actually being partitioned. The only exception to this is when a dimension

is being partitioned a second time. In such cases, the threshold depends on which side of the old partition the new partition will be made. The exception was introduced to avoid creating subspaces where there can be no records.

The use of incorrect thresholds results in terminal nodes with unequal numbers of records. (In the $k-d$ tree of the reference, the number of records in terminal nodes is very similar across nodes). When doing the search, it is possible to end in a node with no records or 60 records. The problem is that when the terminal node has a number of records larger than average, the number of distances computed goes up. However, it is not at all clear that terminal nodes with equal numbers of records will reduce the number of distances that must be examined. It may just be the case that even though there is a lower number of comparisons per terminal node, the number of terminal nodes examined in the search will go up and offset any gains.

Another problem created by the use of SNR to measure spread is that splitting dimensions more than once is more complicated. To start with, a new SNR has to be computed for each partition. In addition, the SNR of each partition most likely will be different, which means that one partition, the one with the better SNR, will be reused before the other. Consider what happens when a dimension is reused. At the level where a dimension is reused, only half the nodes are in the subspace being split. That is, if one is reusing dimension x, previously split along the value 0 and now going to be split along the value 50, only half the nodes contain records with values of x that will be affected by the new split. The other half of the nodes will be affected by the split of the lower SNR side.

There are several ways of dealing with the problem of reusing dimensions (splitting dimensions more than once). First, one can ignore the fact that a partition may have better SNR than a full dimension and never reuse dimensions. Second, one can waste nodes and allow a split to occur even when it does nothing except create a subspace that will always be empty. Third, for the half of the nodes unaffected by a second split, one can use the next best SNR dimension instead. Finally, fourth, one can use

the good and poor SNR sides of the same dimension (sides created by the first time the dimension was used) and split each one at the same time.

Of these approaches, the first 2 were quickly dismissed. Regarding the third approach, it has the advantage of being the most faithful to the original algorithm for creating the $k-d$ tree. However, comparing the third and fourth alternatives, the third alternative shifts nodes of larger SNR[7] towards the root while inserting low SNR nodes at the leaves. The fourth alternative introduces low SNR nodes closer to the root node but it also allows for a simpler implementation of the $k-d$ tree since all the nodes at the same level split the same dimension.

In the end, the 4th alternative was selected for implementation. Only 2 dimensions were used more than once. In both of them, the partitions were of high enough SNR that the side with the low SNR has been moved towards the root only 1 level closer than where the 3rd alternative would have placed it. Therefore, not a big performance penalty is expected from this decision.

In creating the tree, it was arbitrarily decided to have just 10 levels. This allows 1024 terminal nodes (or buckets). The dimensions used in the $k-d$ tree are given in Table 5.2. The table lists which dimension is being split, the threshold, and the SNR of the dimension. When the dimension is being split for a second time, two SNR and threshold values are given. All this information is listed by level, where the root node is level 1 and the terminal nodes are at level 10. Note that the last dimension used, duty cycle, was selected in spite of other dimensions having better SNR values. The reason for this was that the subspaces created by splitting this dimension had very good SNR.

[7]By the SNR of a node, one is referring to the SNR of the dimension being split at the node.

Level	Dimension	Threshold	SNR
1	Eigenvector 2 projection	2320.4	12.52
2	Mean + Median trace frequency	19444.0	11.70
3	Eigenvector 1 projection	9491.4	11.35
4	Eigenvector 2 projection	−2478.2,5155.2	16.73,9.15
5	Mean + Median trace frequency	16510.0,21018.8	8.81,9.59
6	Frequency range (averaging extremes)	7656.2	8.87
7	Duration in seconds	1.029	8.50
8	Eigenvector 3 projection	−1192.0	8.23
9	Eigenvector 4 projection	−1005.1	5.76
10	"Duty cycle"	0.1419	3.34

Table 5.2: List of dimensions used in $k-d$ tree.

5.3 Expanding Bucket Implementation

The basic idea behind the expanding bucket approach is to create a hypercube (bucket), centered at the location of the target record, that slowly grows until the m-nearest neighbors are inside of it. As the hypercube grows, the distance from target to each record inside the hypercube is maintained in a queue. The search ends when all records have been marked as "used"[8]. Once the search is finished, the m-nearest neighbors are read off the queue. The specific details of the algorithm are given below.

Given the above description of the algorithm it should be clear that there are at least two things which must be covered in detail in order to fully understand it. First, one needs to understand what controls the growth of the hypercube. Second, it is necessary to understand the mechanism by which a record gets "used" since this determines when the search is finished. The "used" marker means either that the distance from the record to the target is known, or that the record is so far away from the target that it can not be one of the nearest neighbors being sought.

[8]The quotation marks will indicate the marker "used" in order to avoid confusion with the verb used.

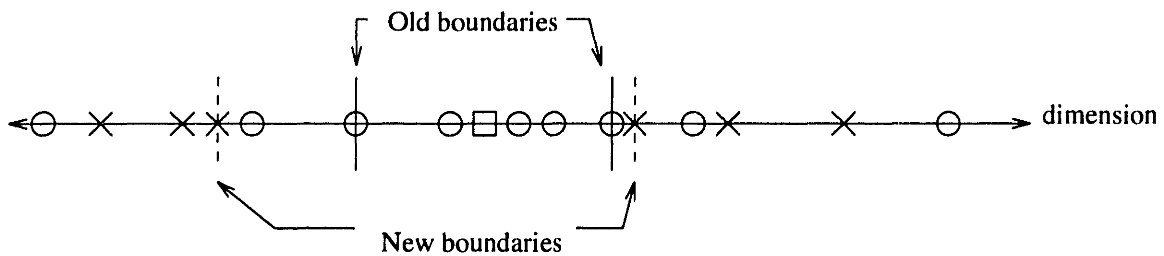


Figure 5-3: Expanding bucket along a dimension (example). - The square is the location of the target record. "Used" records appear as circles and not-"used" records as $\times$'s. The bounds of the hypercube along the dimension shown are moved to the location of the projection for the next not-"used" record.

To define the hypercube one needs two numbers per dimension, a lower bound and an upper bound. Initially, for each dimension, these two bounds are set to the value obtained by projecting the target onto each dimension. That is, for dimension x, one projects the target onto x to obtain a number γ; the lower and upper bounds for x are then set equal to γ. This results in a zero-volume hypercube. The expansion of the hypercube is done one dimension at a time. The upper bound is moved to the next larger record projection and the lower bound is moved to the next lower record projection. Projections from "used" records are skipped when moving the hypercube's bounds.

Figure 5-3 provides a graphical example of the expansion operation described in the last paragraph. In the figure, let the circles, $\times$'s and square be record projections onto a dimension (horizontal line). The square represents the target record, circles represent "used" records, and $\times$'s represent not-"used" records. The solid vertical lines are the current hypercube boundaries while the dashed vertical lines are the hypercube boundaries after expansion. As seen from the figure, the hypercube boundary always moves to the projection of a not-"used" record. In order to make the expansion efficient, a sorted list for each dimension containing (record, projection value) pairs is kept. There is one list per dimension.

Given the scheme used for expanding the hypercube, it should be clear that every time a boundary is moved, the record the boundary gets moved to has a chance to enter the hypercube. As soon as a record enters the hypercube, the distance from target to record is computed and entered into the queue. Then, the record is marked as "used". Note that at each expansion, one only needs to check the new boundaries to see if there is a new record in the hypercube. Only 2 checks are required per dimension expansion.

To check if a record is inside the hypercube, a simple shifting technique is used. First, initialize an array of size equal to the number of records in the database with ones. Each record is numbered according to its position in this array. Then, each time the projection of the record along a dimension lies inside the projection of the hypercube along the same dimension, the number at the position in the array corresponding to the record is shifted left by one. If the value gets to 2^d, where d is the dimensionality of the hypercube, then the record is inside the bucket.

Let us now go back to the "used" marker. If the only way a record could get a "used" marker were by computing the distance from it to the target, then this method would be nothing more than a highly organized full search. When a record enters the hypercube, even before the distance to the target is computed, a quick distance bound is estimated. If this distance bound is larger than the distance to the mth nearest neighbour found so far, then all records above (when the new record is at the upper bound) or below (for a lower bound) are also marked as "used". Not only that, but the distance bound is also reflected onto the opposite hypercube boundary and even more records may be labelled as "used".

The quick distance bound is based on an array which keeps track of the minimum contribution a dimension may make to the distance of any record entering the bucket. Each element in the array (one element per dimension) contains the 1-D distance from target to the nearest not-"used" record projection for the corresponding dimension. Assume one has just moved the upper hypercube bound for dimension x

and a new record entered the hypercube. The quick bound is computed using all the elements of the array except the one for dimension x, the true 1-D distance is used for that element. This true 1-D distance comes from the record that just entered the hypercube. If the quick distance bound is smaller or equal to the distance to the mth nearest neighbour, the true distance to the target is computed, added to the queue and the record is marked as "used".

On the other hand, if the distance bound is larger than that of the mth nearest neighbour, all records with a value for the projection onto dimension x which is larger than or equal to that of the record just entering the hypercube are marked as "used". Let us say that the above operation results in all records with projections at least 50 units above that of the target being marked as "used". Then, all records with projections at least 50 units below that of the target are also marked as "used". This is the reflection operation mentioned above.

Each time a record is marked as "used", the array used for the distance bound is checked to see if it needs any changes. When the expansion process reaches one of the ends for a dimension, the same bound is used for the rest of the search regardless whether the end record is marked as "used" or not. The search ends when all records are marked "used".

5.4 Experimental Results

To evaluate the two search strategies described above, a database with more than the initial 92 records was necessary. Therefore, the 92-record database was expanded to 1169 records (whistles) from 22 different animals. Table 5.3 lists the files from which the records were obtained along with the number of records from each file and the id. of the animal generating the sound. As a reminder, the first five digits of the file name identify the tape from where the samples were taken.

File	Number	Id	File	Number	Id
905780K1.lv2	41	FB 161	905780K2.lv2	47	FB 164
90579AK1.lv2	47	FB 163	90579BK1.lv2	49	FB 168
90580AK1.lv2	45	FB 50	90580BK1.lv2	54	FB 90
905810K1.lv2	46	FB 92	905890K1.lv2	46	FB 183
90588K01.lv2	55	FB 158	91545K01.lv2	35	FB 20
91537K02.lv2	54	FB 3	91543K02.lv2	41	FB 17
91546K01.lv2	58	FB 23	91546K02.lv2	5	FB 15
88206c1.lv2	35	FB 55	89506c1.lv2	30	FB 28
89506c2.lv2	54	FB 55	89512c1.lv2	74	FB 65
89512c2.lv2	48	FB 67	89516c1.lv2	83	FB 90
89516c2.lv2	81	FB 97	89518c1.lv2	13	FB 35
89518c2.lv2	59	FB 13	90580c1.lv2	44	FB 90
91549c1.lv2	25	FB 24			

Table 5.3: Contents of large database (1169 records, 22 animals).

As was expected, the terminal subspaces of the $k-d$ tree are not evenly occupied with records. Of the 1024 subspaces, only 253 of them contain any records at all. The average number of records in the terminal subspaces occupied was 4.62. The occupied subspace with the largest number of records had 60, while the one with the smallest number had one.

How the files are distributed along the occupied terminal nodes is shown in Figure 5-4. Each point in the figure represents a different file with the number of records in the file along the x axis and the number of terminal nodes occupied by those records along the y axis. The solid lines in the figure correspond to different values of average number of records per terminal node. Starting clockwise from the y axis, the first solid line is an average of 1 record per terminal node, then 2, 3, and 4. Most files correspond to averages between 1 and 2 records per terminal node.

Initially, the probability of false alarm (claiming a repetition when none has occurred) and the probability of miss (not being able to locate the correct repetition when present) were considered for the evaluation of the search procedures. However, some analysis showed that when records are stored in full precision, it is impossible to

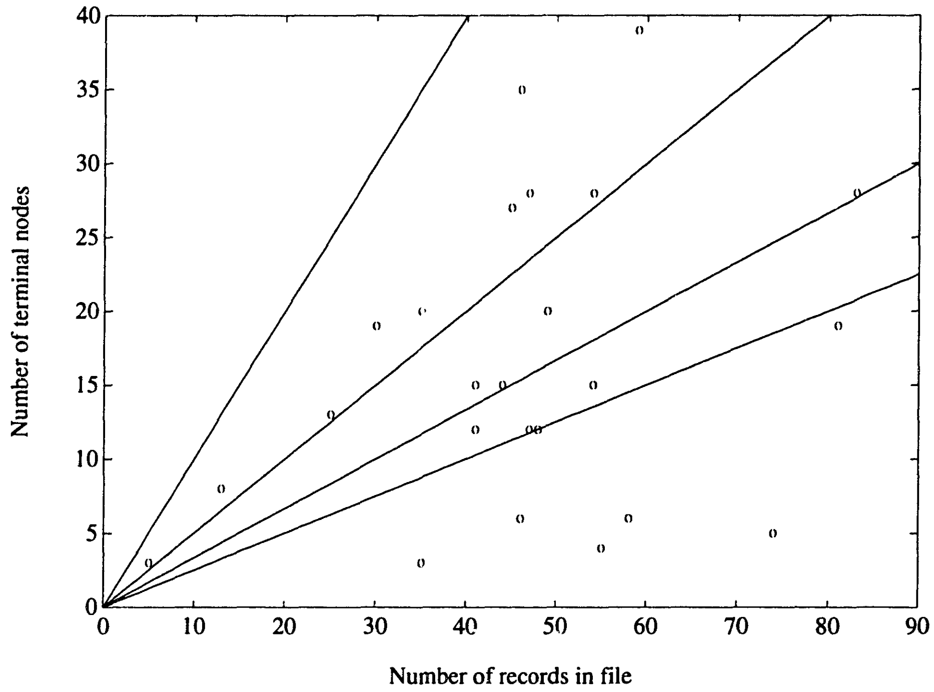


Figure 5-4: Number of terminal nodes used by each file in the database.

get a false alarm. Whichever record the search procedure claims to be the closest to the target, after comparing its distance to the target with the distance threshold used to identify repetition there is no room left for ambiguities. It can be argued that a false alarm may still occur if the distance is not a good indicator of when a repetition takes place. For example, if the duration is the only dimension used to identify the animals, any whistles of equal duration would be regarded as equal. Nevertheless, two very different whistles may last the same amount of time and therefore calling them the same and claiming a repetition can be considered a false alarm but it is a false alarm created by the use on an inappropriate measure for whistle identification. However, if the distance is an unreliable measure of whistle similarity, there is nothing the search algorithm can do about it and therefore it should not be used against the algorithm.

False alarm probabilities can be computed for the case when the space is quantized and records are no longer stored in full precision. The loss of precision can result in the quantized version of the record being close enough to the target so that it may be identified as a repetition while the full precision version is not close enough. However, since the records are not quantized, there is no need to worry about false alarms.

Similarly, it is not possible to have a miss if the search algorithm is allowed to examine as many records as necessary (even the entire database) to find the nearest neighbour. Once a search budget is established, either in terms of time, records examined or arithmetic operations, there is a chance of missing the repetition if the budget runs out before one gets to the nearest record. In the algorithms, as described, there is no budget constraint given. Nevertheless, as part of the experiments below it was examined how close to the nearest neighbour (given by the rank of the record returned as closest) one gets as a function of budget.

It would seem that one can set any desired probability of miss and false alarm independently of each other by adjusting two control variables. That is, by adjusting the amount of quantization and the search budget. However, the only time one

can really do this is when the budget is specified as a the maximum number of comparisons allowed. When the budget is specified in terms of time or computations, then the probabilities are interrelated. This is because in a quantized space, arithmetic operations can be done faster. Just consider the case of storing all distances between quantized records in a lookup table as discussed previously.

Having determined that the probability of false alarm and miss were not good criteria to use in comparing search algorithms, the four measures listed below were used instead.

1. number of records examined to *find* the nearest neighbour
2. number of records examined to *confirm* the nearest neighbour
3. number of total computations done during the search
4. total time (average) required to complete the search

In determining the number of computations, one computation was equal to one addition and two multiplications. Also, taking a square root was counted as one computation[9]. Logical operations or simple additions were not taken into account. The time reported was the average of 5 full-database searches, that is, 5×1169 searches. All times were obtained using a SUN computer (SPARCstation 2) and with the experiment measuring the time being the only user's task executing.

Note that there is a distinction between the number of records that need to be examined to find the nearest neighbour and the number of records needed to confirm the nearest neighbour. In this context, the term "find" refers to when the nearest neighbour is first encountered and the distance between it and the target computed. At this point in the search, it is not yet known that the record is indeed the nearest

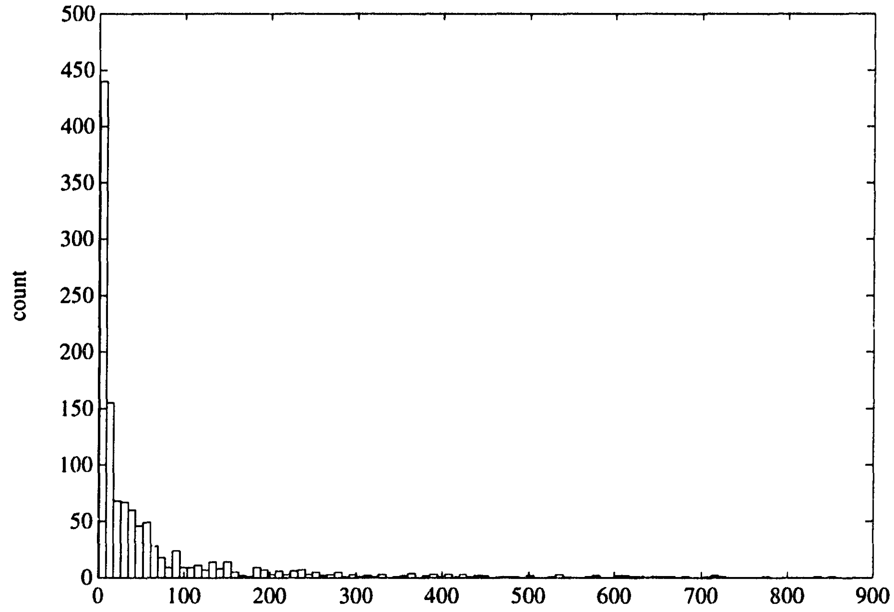
[9]The definition of a computation is based on the definition of distance. Each term required to compute the distance contributes one computation and taking the square root at the end contributes another.

neighbour. Therefore, the search must continue until the nearest neighbour is confirmed. The confirmation occurs when the search is complete. When a search budget is imposed, the number of records required to find is more important than the number of records required to confirm. That is because the faster we find the correct answer the better the chances of having the correct answer by the time the budget runs out. If there is no budget restriction, the converse is true.

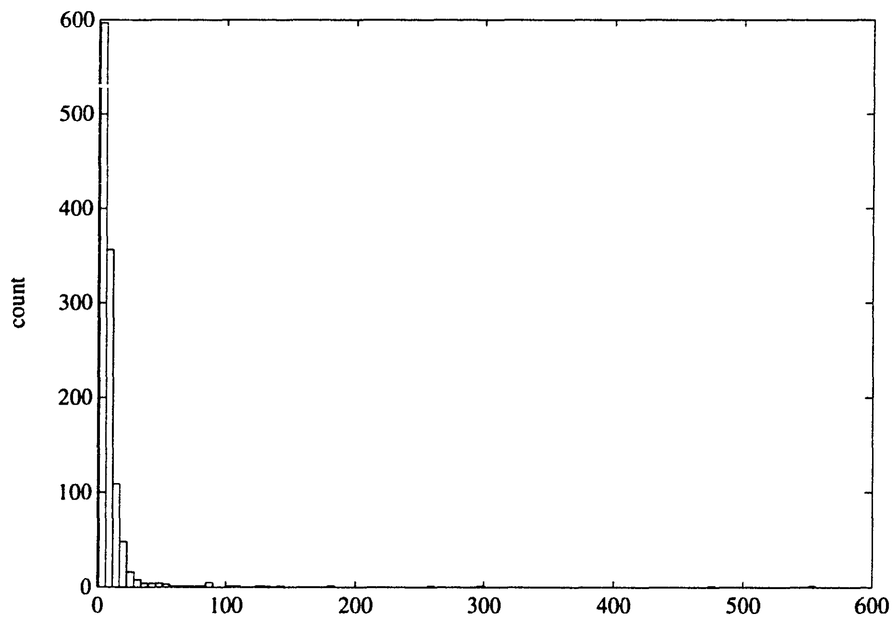
Figures 5-5 to 5-7 show histograms for the first three of the above measures. In each figure, the top histogram is obtained with the $k-d$ tree search algorithm while the bottom figure comes from the expanding bucket algorithm. For the last 2 sets of graphs, a large peak is visible at the right of the histograms for the $k-d$ tree algorithm. This spike is formed by those searches which required all the records in the database to be examined[10]. The closer the target record is to the subspace boundaries the higher the chances of having to examine additional subspaces and the higher the chances of examining the entire database. The expanding bucket approach was developed mainly to address this issue. Since in the expanding bucket the target is always at the center of the subspace the chance of a full database search is very small.

The average value of all 4 measures is shown in Table 5.4. As seen in the table, although the expanding bucket method requires 6 times fewer comparisons in order to encounter the nearest record and 3 times fewer comparisons to confirm it, the method is almost 2 times slower than the $k-d$ tree approach. The only explanation for this is that the expanding bucket expends too much time in logical operations, such as determining if a record is inside the bucket and finding the next unused record when moving the boundaries of the bucket, and that the execution time suffers as a result. In support of this conclusion, to "reach" a record in the $k-d$ tree algorithm just 10 comparisons are required (one per tree level). On the other hand, to

[10]This is not the same as saying that all terminal subspaces (nodes) were examined. Only the 253 occupied subspaces have to be visited in order to search the entire database.

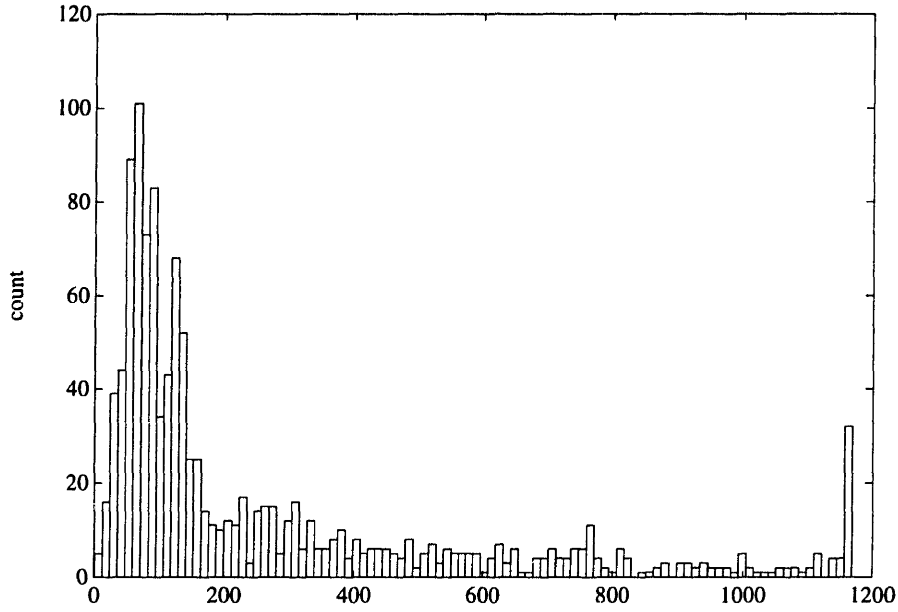


(a) $k - d$ tree algorithm

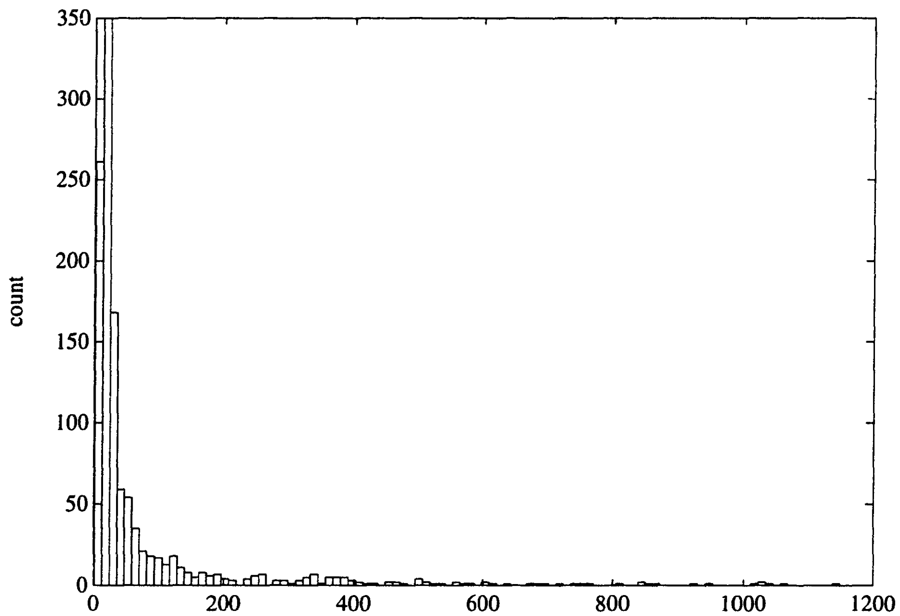


(b) expanding bucket algorithm

Figure 5-5: Histogram of number of records needed to encounter nearest neighbour.

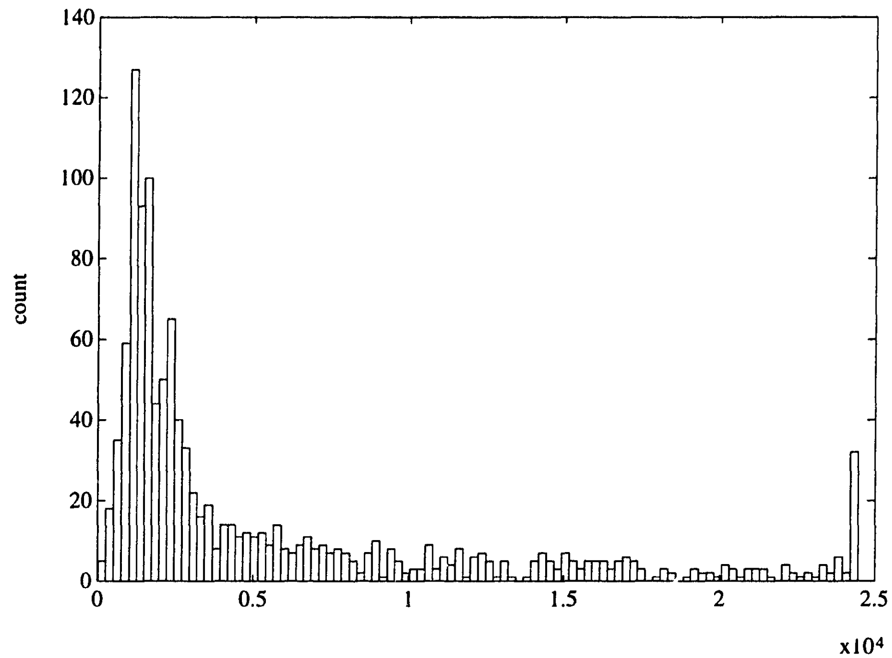


(a) $k - d$ tree algorithm

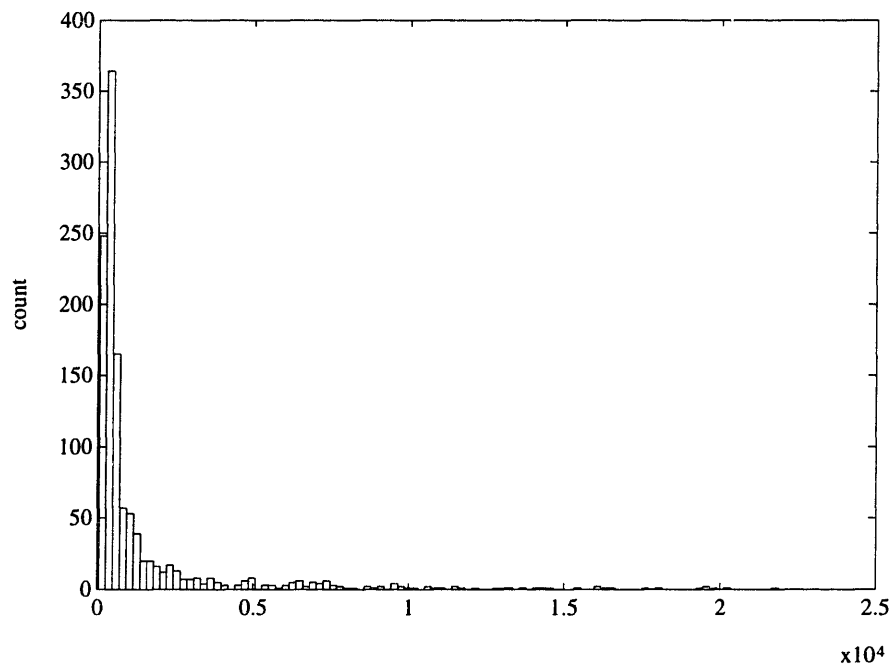


(b) expanding bucket algorithm

Figure 5-6: Histogram of number of records needed to confirm nearest neighbour.



(a) $k - d$ tree algorithm



(b) expanding bucket algorithm

Figure 5-7: Histogram of number of computations performed during search.

Measure	$k - d$ tree	Exp bucket	Ratio
records examined to encounter nearest neighbour	65.5278	10.6946	6.13
records examined to confirm nearest neighbour	270.559	73.6972	3.67
computations done during search	5542.17	1445.12	3.84
average time for search	0.046282 s	0.079688 s	(1.72)

Table 5.4: Average performance of search methods.

"reach" a record in the expanding bucket one must do 2 comparisons per dimension (32 comparisons per algorithm's iteration) and then do all the shift operations to determine if the record is inside the bucket or not. Clearly, the expanding bucket spends much more time in logical operations instead of arithmetic operations than the $k - d$ tree algorithm.

If one considers the subspace of each terminal node to be a bucket in the coding space, then it becomes clear that the $k - d$ tree uses 8-D buckets in its search[11]. On the other hand, the expanding bucket uses a 16-D bucket in its search. This prompted a set of experiments investigating how the performance of the search algorithms is affected by the dimensionality of the bucket used in the search. There are two ways of reducing the dimensionality in the expanding bucket search. For example, consider a search of dimensionality equal to 8. First, one can continue expanding the bucket in 16 dimensions but let a whistle be inside the bucket when it is inside for any 8 of the 16 dimensions. Second, the bucket itself may be made 8 dimensional. Experiments demonstrated the second approach to be faster and thus it is used for the results presented here. The dimensions used in forming the buckets were the ones with higher SNR values.

[11]Recall that the $k - d$ tree has 10 levels but 2 dimensions are used twice. Therefore, navigation is based on 8 dimensions only.

Dims	$k-d$ tree method			
	records to encounter	records to confirm	total comps	ave time
10				
8	65.528	270.56	5542.2	46.3 ms
6	79.068	343.70	6099.4	35.1 ms
3	167.19	661.82	11257.7	48.0 ms
1	333.42	951.50	16176.0	69.3 ms

Dims	expanding bucket method			
	records to encounter	records to confirm	total comps	ave time
16	10.695	73.70	1445.1	79.7 ms
10	14.871	122.76	2372.7	62.2 ms
8	15.138	147.54	2841.3	54.5 ms
6	14.180	179.91	3453.6	52.7 ms
3	38.714	306.36	5853.2	53.8 ms
1	126.43	557.54	10957.1	84.6 ms

Table 5.5: Performance of search algorithms as dimensionality changes.

Table 5.5 shows the results of the experiments. As can be seen from the table the expanding bucket method is faster when an 8 dimensional bucket is used. However, the method was not able to surpass the $k-d$ tree in speed (see Figure 5-8). Another interesting observation is that a 6-D bucket (or terminal subspace) seems to be optimal for both methods. Looking at the SNRs of the individual dimensions, the sixth dimension has an SNR of 8.23 while the one after has 5.76. It is possible that the low SNR dimensions are responsible for the decrease in performance for buckets of more than 6 dimensions. It is also possible, although less likely, that the optimal number of dimensions depends on an absolute SNR threshold, or it is completely independent of SNR. There is no good answer to the question at this moment.

Also of importance is how well the cost of the search algorithms scale with respect to database size. For this experiment the total number of records examined to confirm

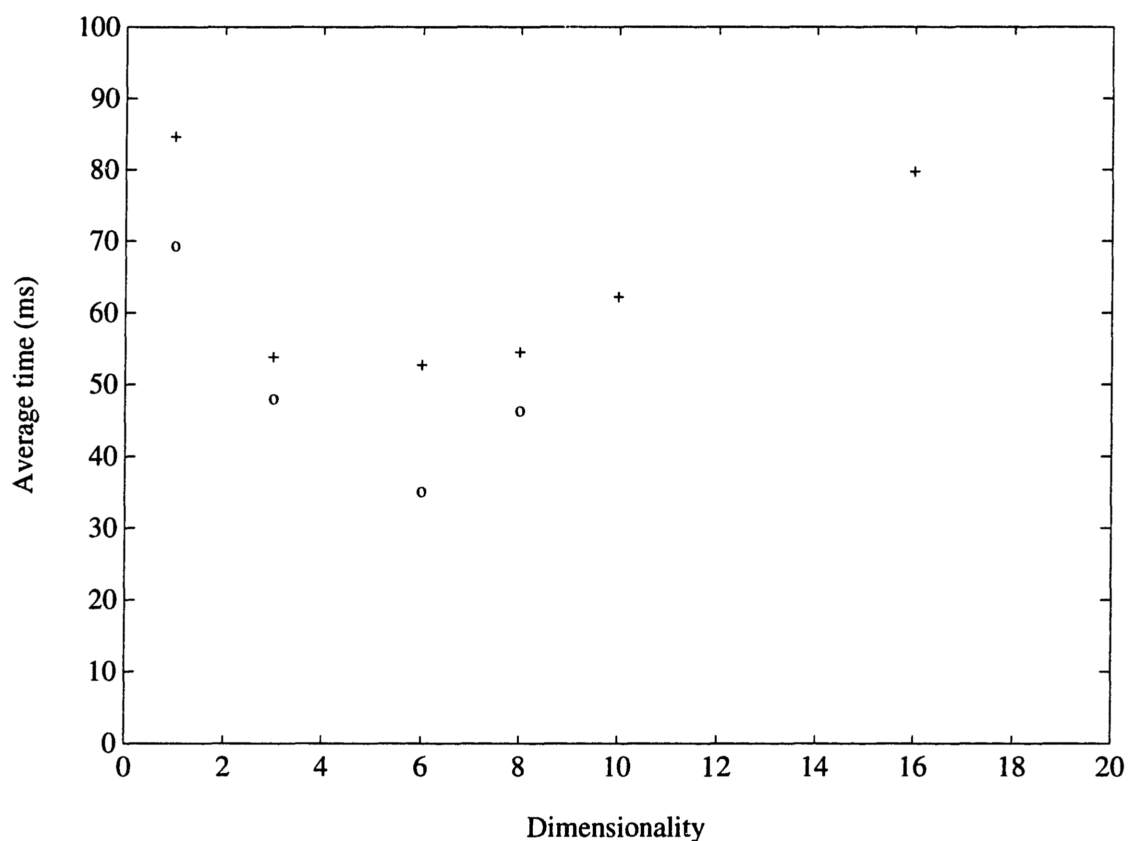


Figure 5-8: Average search time as a function of dimensionality. - Average time required for a full search of the 1169-whistle database. The x-axis indicates the dimensionality of the bucket used in the search. The dimensionality of the space (used to compute distances) stayed fix at 16.The $k-d$ tree results are denoted by 'o's and the expanding bucket results by '+'s.

Size	$k - d$ tree	expanding bucket
1169	270.5586	73.6972
585	166.9402	57.3607
390	122.4615	49.9769
293	103.8089	49.2594

Table 5.6: Records needed for confirmation as a function of database size.

the nearest neighbour is used to represent the total cost of the search. It is expected that average time and computations will scale similarly to the cost selected. Table 5.6 shows the cost obtained for 4 different database sizes. Figures 5-9 and 5-10 display the same information in the table but also include a linear (solid line) and logarithmic (dashed line) fit of the data. The $\times$'s in the figures correspond to a 2σ interval around the average cost, denoted by the circles.

Unfortunately, both algorithms do not seem to scale too well. Both algorithms seem to be linear with respect to database size. The slope of the line is small for the expanding bucket, only about 0.0288 records examined per record in the database. For the $k - d$ tree, the slope is about 0.19. Nevertheless, neither algorithm scales any better than the full search method.

If the $k - d$ tree is implemented as described in [9] the search cost should be independent of database size. By using a static $k - d$ tree, performance of the tree-based search has been degraded. However, if the tree is not static, every time there is a change in the database the entire tree has to be reconstructed. When reconstruction of the tree is necessary for each search, the static tree then offers better performance than the tree implemented in [9].

In addition to the above experiments, the behavior of the algorithms when the search was restricted by a budget was also examined. The budget used was in terms of the number of computations allowed during the search. The log-log plot (base 10) in Figure 5-11 shows the average rank of the closest record found as a function of the computational budget for 3 different database sizes. Since the expanding bucket

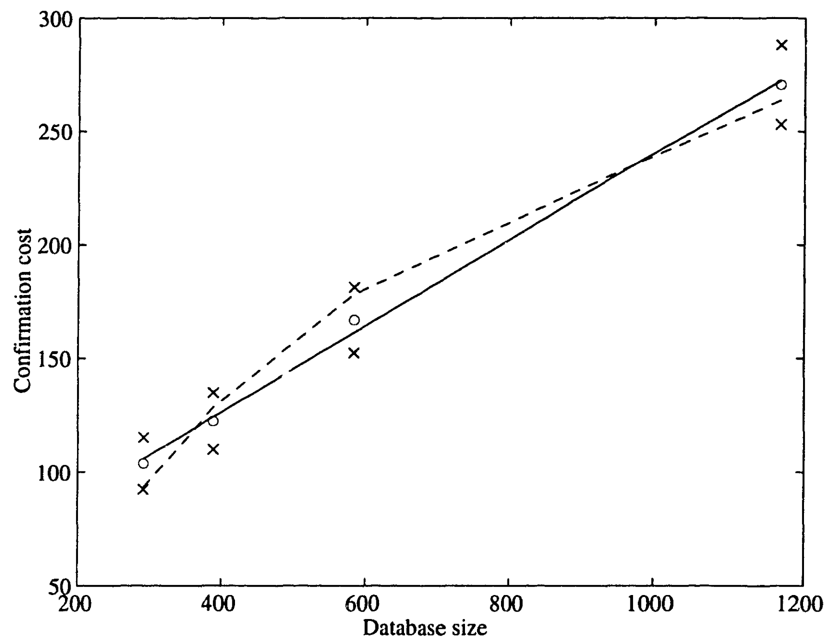


Figure 5-9: $K - D$ tree performance as a function of database size. - The average number of records examined for confirmation is denoted by a circle. The $\times$'s mark the endpoints of the 2σ confidence interval for the average. Two fits of the data are shown, linear (solid) and logarithmic (dashed).

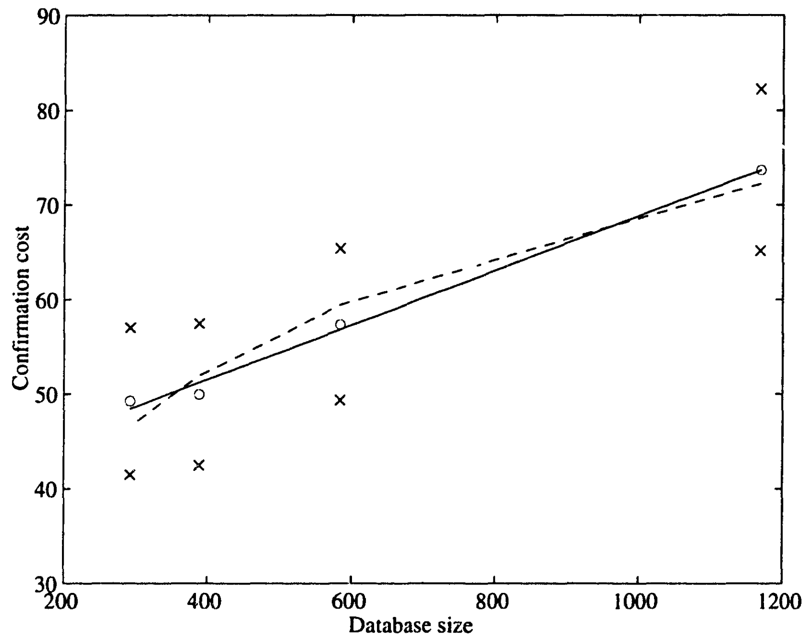


Figure 5-10: Expanding bucket performance as a function of database size. - The average number of records examined for confirmation is denoted by a circle. The $\times$'s mark the endpoints of the 2σ confidence interval for the average. Two fits of the data are shown, linear (solid) and logarithmic (dashed).

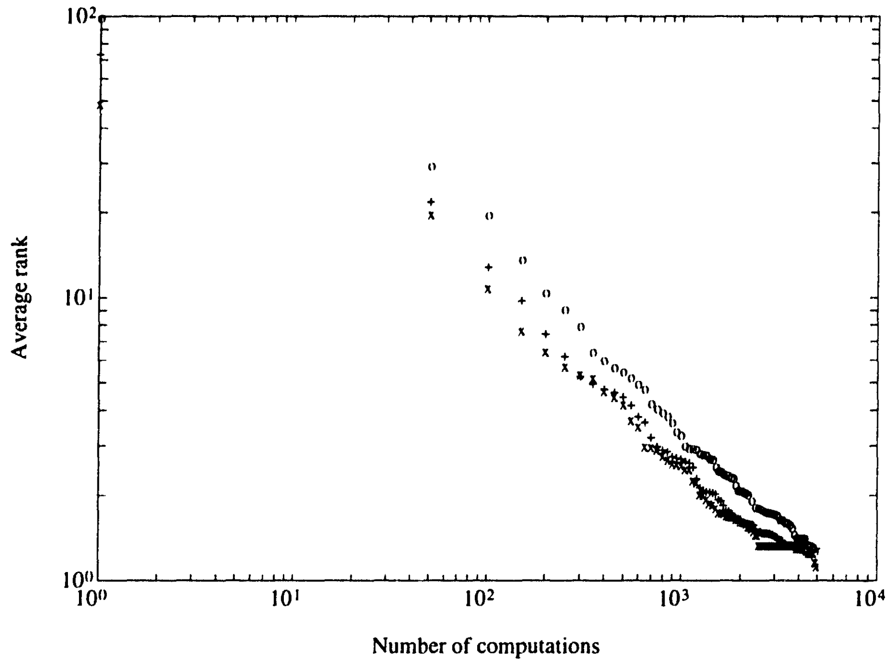
method requires fewer computations than the $k-d$ tree method, it is no surprise to see that, for a given budget, the expanding bucket performs better than the $k-d$ tree. It should be noted that the performance of the expanding bucket method improves faster as the budget increases than that of the $k-d$ tree method.

When the specified budget is insufficient to confirm the search, the performance shown in Figure 5-11 is expected to vary with the contents of the database. It was noticed that for the $k-d$ tree algorithm the performance not only varies with content, but also with organization of the database. The performance when the search is restricted by a budget was determined again for the $k-d$ tree but using a different organization for the records in the tree. Results are shown in Figure 5-12. The old results are marked with circles and the new ones with "+"s. As can be seen from the figure, the new organization results in worst performance. When the budget is large enough the two performances converge.

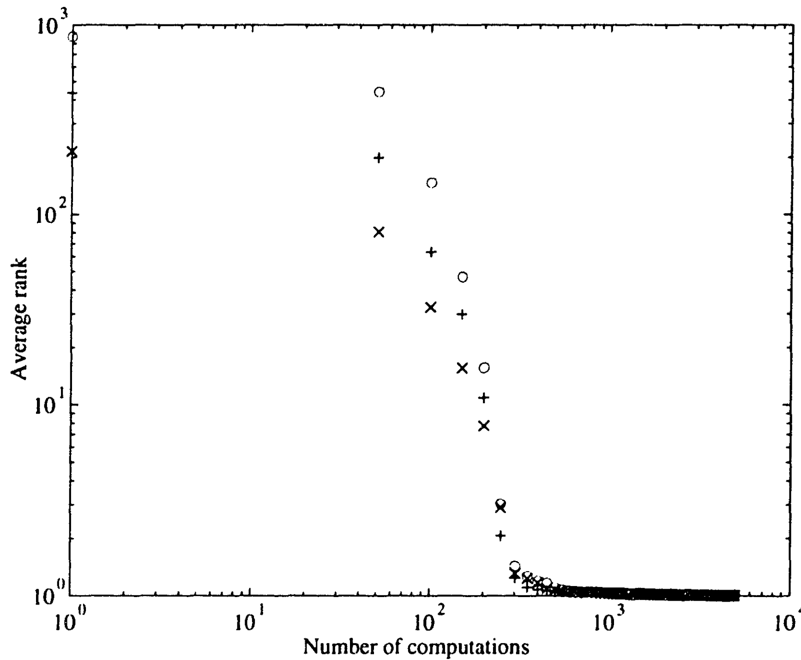
Figure 5-13 shows how the rank and the average distance (using all records) relate to each other. The figure shows the average distance (circle) and the maximum and minimum distance ($\times$'s) for each rank. The average distance gets progressively worse as the rank increases, but there is not a big absolute difference in distances. Also, the range of possible values, those between maximum and minimum, is quite large.

Another interesting experiment conducted with the database was to search for all instances in which the nearest whistle to the target was not produced by the same animal. A total of 92 distinct pairs of whistles (about 8 % of the database) fit this category. While examining these records (whistles) it became evident that they were very different from other whistles produced by the same animal and therefore the search algorithm was correct in matching the target to a whistle produced by a different animal.

The same experiment described in the last paragraph was performed using the 180-whistle database of Table 4.9 and 11 distinct pairs of whistles were found. Most of the pairs came from animal FB 163 (whistles 26, 28, 29, 32, 35, 36, 39, and 40).



(a) $k-d$ tree algorithm



(b) expanding bucket algorithm

Figure 5-11: Average rank of nearest neighbour versus search budget - The x axis shows $\log_{10}$(budget) and the y axis the $\log_{10}$(average rank) of the closest record found by the time the budget ran out. Budget is specified as the number of computations allowed for the search. The database sizes are: 1169 ("o"), 585 ("+"), and 293 ("×").

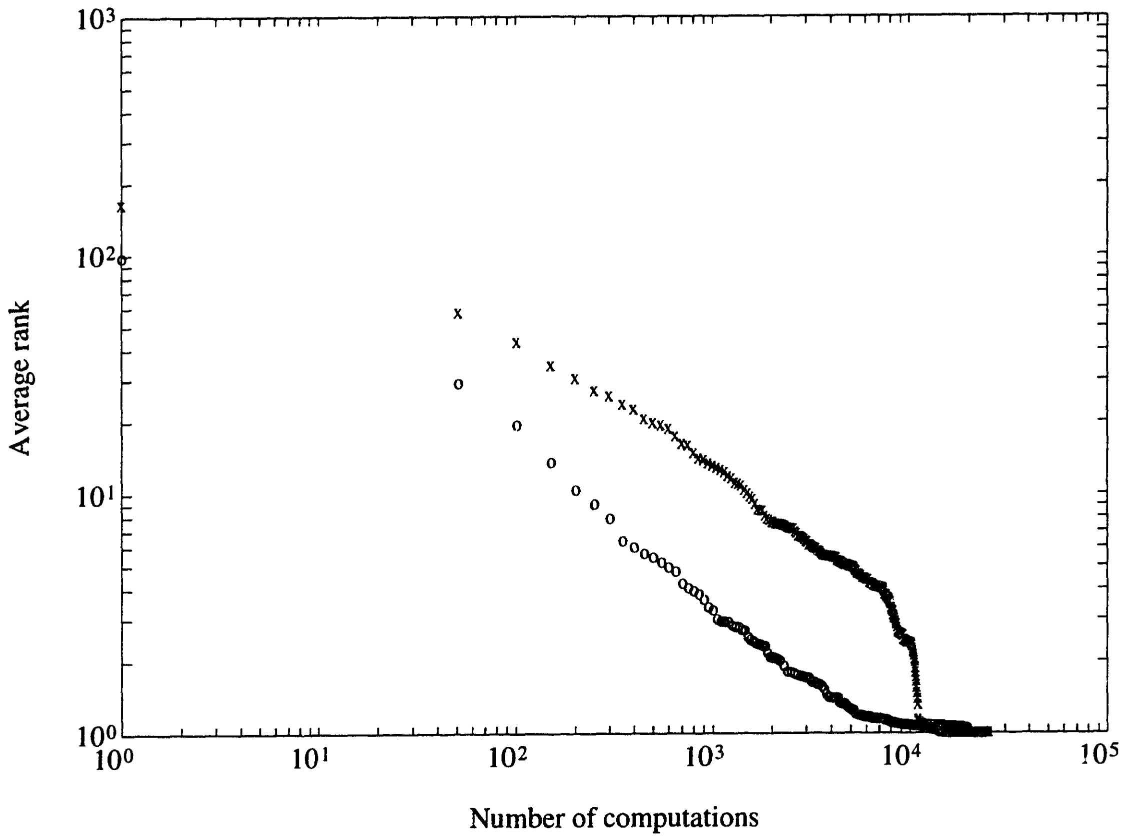


Figure 5-12: $K - D$ tree sensitivity to database organization. - The average rank of nearest neighbour is shown as a function of search budget for two different database organizations. When the budget is insufficient to confirm the search, $k - d$ tree performance is highly dependent on how the records are placed in the tree.

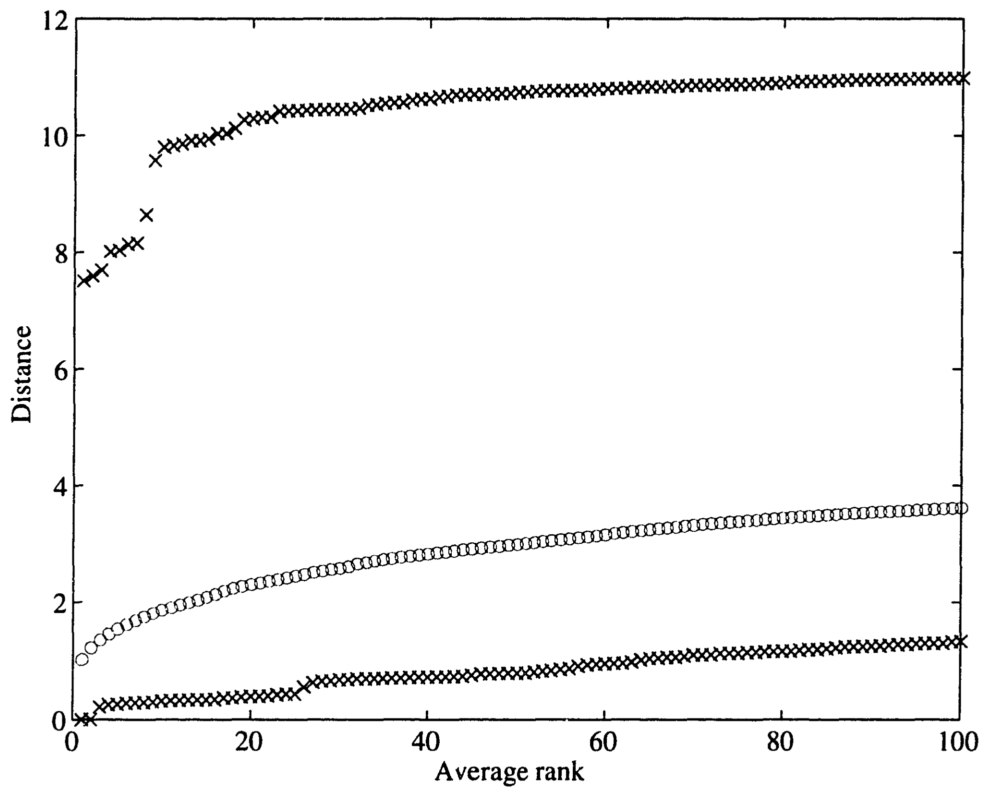


Figure 5-13: Distance as a function of rank - The average distance for each rank is marked with "o". The "×" marks the maximum and minimum values obtained at each rank.

These whistles were also hard to classify by humans (see Table 4.12) as evidenced by the number of times they were placed in a single cluster or the wrong cluster. There was one other animal accounting for the remaining pairs, FB 183 (whistles 2, 7, and 8). All other whistles that appear as misidentified in Table 4.9 (e.g. FB 90, whistle 20) were correctly matched using whistle distance.

Finally, since the distance between records in the coding space has been proposed as a way of telling how similar dolphin whistles are, a set of experiments was conducted to investigate. In each experiment, a random whistle (record) from the database was selected and plotted on the same page as other records obtained as the distance from the target grows. The results are shown in Figures 5-14 to 5-16. As one compares the whistles plotted, going from the nearest at the top-left of the figure to the one farthest away at the bottom-right, it is evident that distance does reflect how similar one whistle is to another.

Plots of distance versus rank (Figure 5-17) showed no clear breakpoints in distance. That is, there is no point at which the distance jumps up marking a clear boundary between those whistles that are similar and those that are not. What was found instead was that distance goes up quickly as one goes down in rank (moving away from the very similar whistles in the same cluster). Then, for a wide range of rank values the distance increases slightly only to go up again quickly as the extreme cases are reached.

To summarize, experiments with both search algorithms have shown that although the expanding bucket has a clear advantage in term of the number of distances that need to be computed to locate the nearest records, its speed is slower than that of the $k-d$ tree. Speeds do become comparable when 6 dimensions are used and both methods achieve their best speeds. The number of records examined in the search seems to grow linearly in both algorithms. It was also noted that when there is a budget constraining the search, the expanding bucket can yield better results than the $k-d$ tree algorithm.

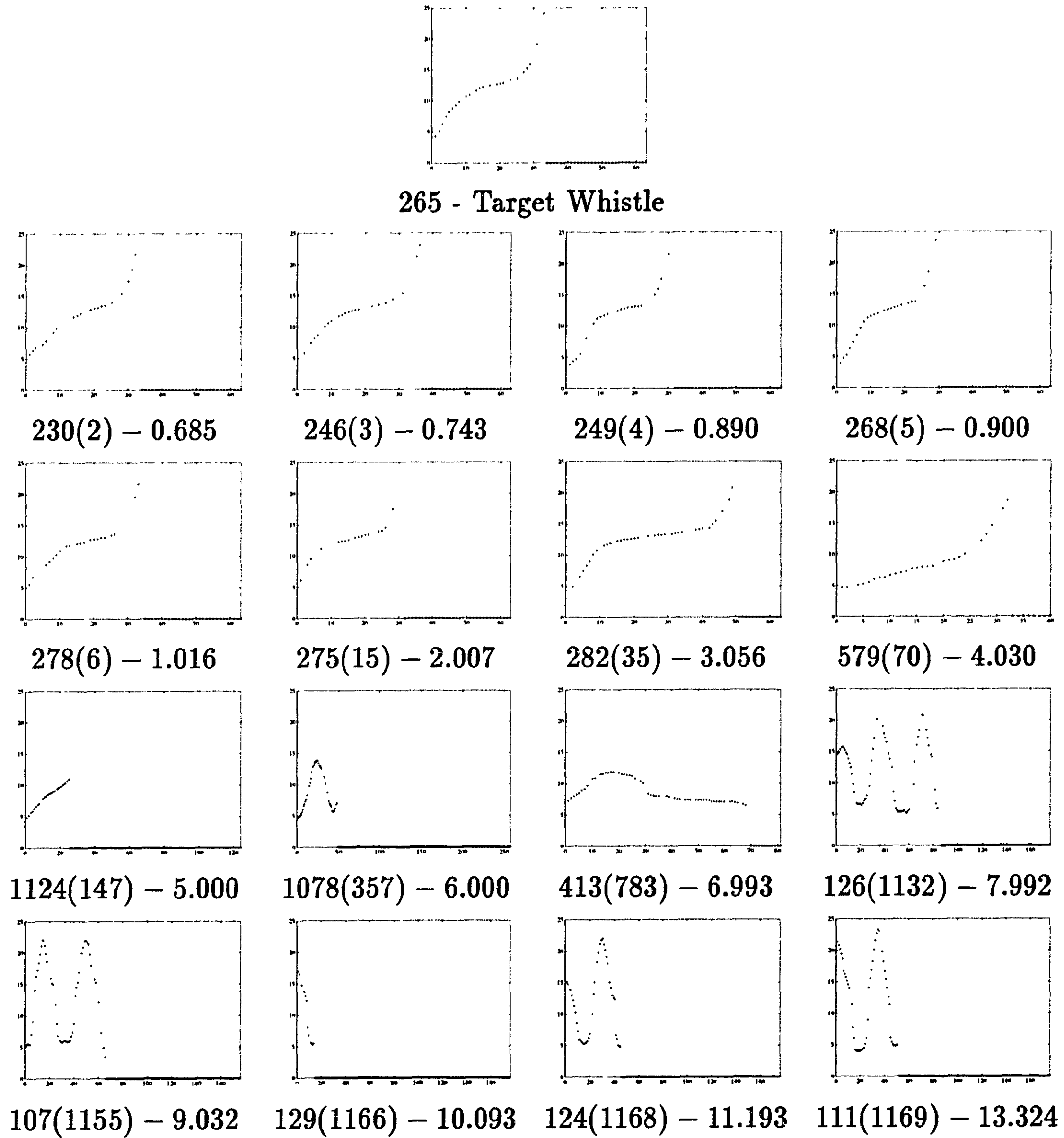


Figure 5-14: Distance validation - example 1: For whistle #265 (Animal FB90) in database various other whistles are shown as a function of distance. Each whistle is identified by its number in the database, rank in parenthesis and the distance to the target.

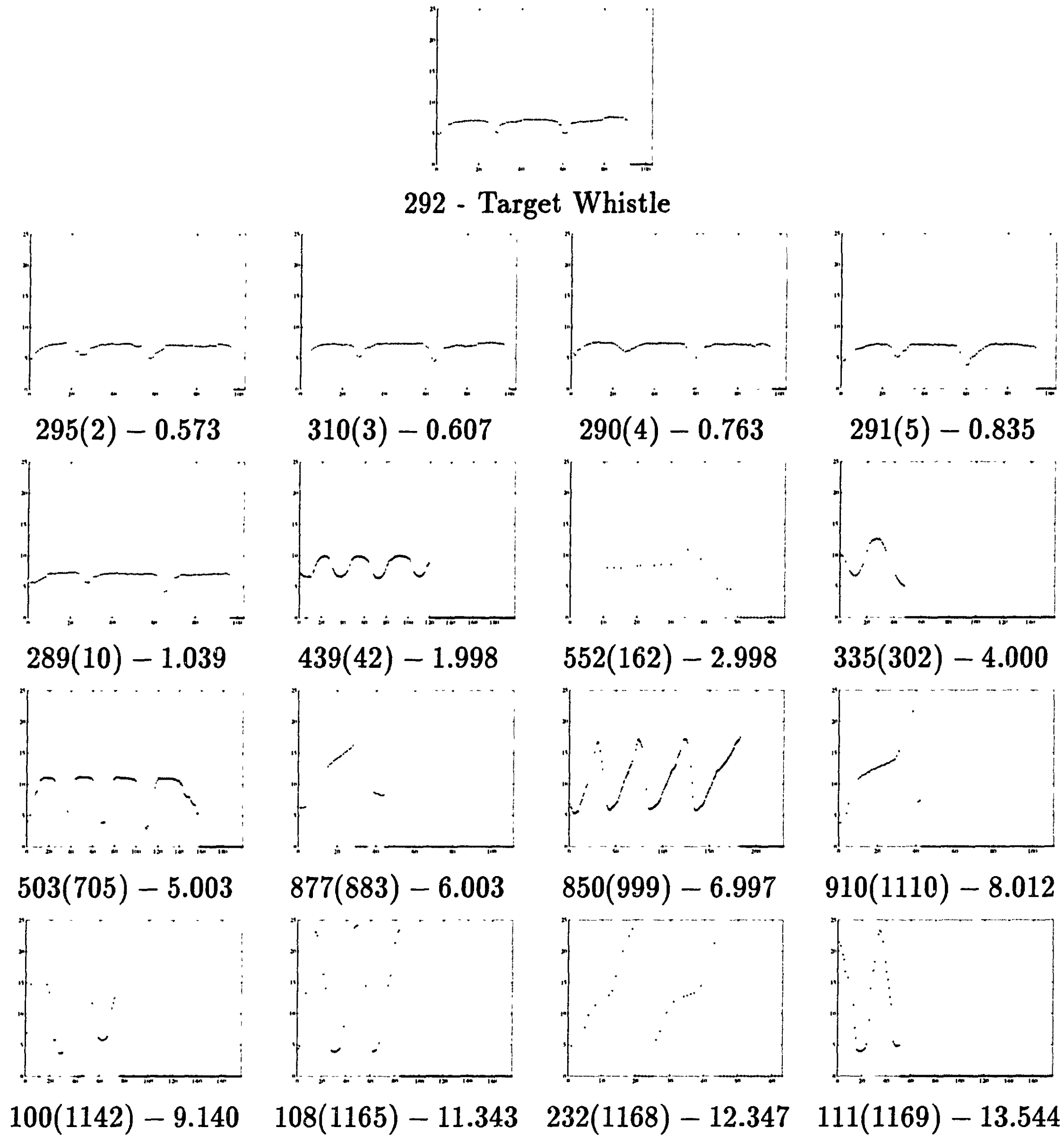


Figure 5-15: Distance validation - example 2: For whistle #295 (Animal FB92) in database various other whistles are shown as a function of distance. Each whistle is identified by its number in the database, rank in parenthesis and the distance to the target.

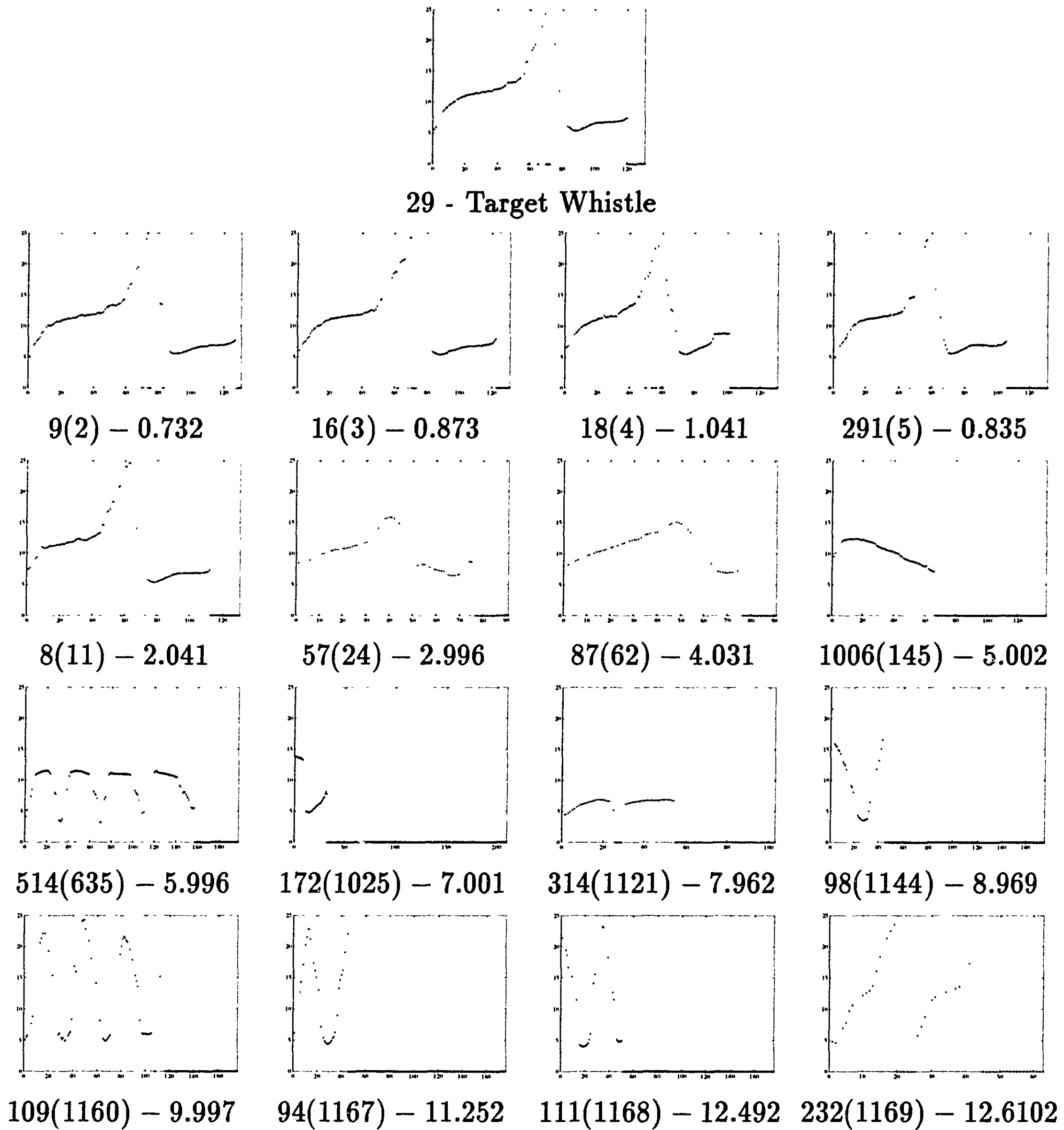


Figure 5-16: Distance validation - example 3: For whistle #29 (Animal FB161) in database various other whistles are shown as a function of distance. Each whistle is identified by its number in the database, rank in parenthesis and the distance to the target.

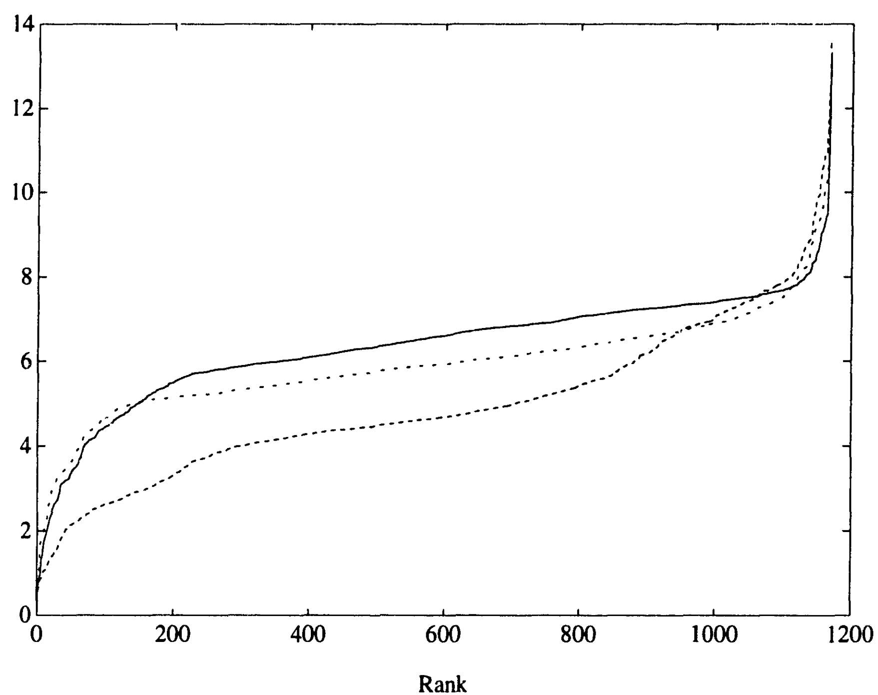


Figure 5-17: Distance as a function of rank for 3 random whistles - The three random whistles are the ones shown as target in Figures 5-14 to 5-16. Looking at rank 600, the top line is animal FB 265, middle line is FB 29 and bottom line is FB 292.

Chapter 6

Conclusions

The work presented here has covered the complete development of a system for the identification of compressed sounds. Using the dolphin's whistle database at WHOI a fully operational system has been implemented based on the ideas presented in this document. All of the results presented here have been obtained from actual experiments using this system. In this respect, the thesis has been successful in achieving its goal of developing a system for identifying compressed signals in a database.

This chapter contains two sections. The first section is a combination summary and conclusions. It covers the major aspects of the system while pointing out things that were learned along the way. The second section covers ideas for future work and possible modifications to the system to improve its performance.

6.1 Summary and Conclusions

The development of the system was based on three levels. During level I, discussed in Chapter 2, it was learned that of three possible information carrying streams (tone, amplitude and clicks) only one was useful for whistle identification. A signal detector, appropriate for identifying the presence of strong high-frequency signals in background noise was also developed. This detector is not application specific and

can be used in other areas as long as the signal to be identified has the appropriate characteristics.

Specific to the application, during level I it was discovered that animals tend to have more control over the frequencies in their own signature whistle than over the frequencies of imitated whistles. This was evident from a study of standard deviation for repeatable frequencies in the signature whistles. The variation of the copied frequency can be as high as 3 times that of the original frequency.

The compression achieved by level I was typically about a factor of 12. This compression was obtained by a combination of silence removal and quantization of the power density spectrum of the signal.

Level II, covered in Chapter 3, concentrated in the extraction of a single frequency-vs.-time trace out of the spectrogram produced in level I. A suboptimal algorithm, followed by a manual editing of the trace (when necessary) was the solution implemented. The chapter includes a formulation of the tracing problem in such a way as to make the use of dynamic programming algorithms a viable (and optimal) solution to the tracing problem. This optimal solution was not implemented, however, due to its complexity. A future implementation of the tracing algorithm could benefit by using the formulation to implement the optimal solution.

The compression of the level II system was shown to be an additional factor of 257 (for a total factor of 3084). However, if reversibility is desired, information in addition to the frequency-vs.-time trace has to be stored and the compression factor goes down. Another observation made during level II compression was that typically, about 3 Karhunen-Loeve eigenvectors are sufficient to capture about 50 percent of the variability in the dolphin signal. If a higher variability capture is desired, many more eigenvectors are required. For example, 12 eigenvectors only capture 55 percent of the variability.

Level III, covered in Chapters 4 and 5, is perhaps the most important contribution of this work. The work in this level combines compression with efficient storage and

retrieval to obtain the signal identification system. In level III, the output of level II is converted to a point (one point per whistle) in a multi-dimensional coding space. The compression factor achieved by level III alone is about 1. Then, when doing signal identification, the multi-dimensional space is searched to find the nearest neighbors to the signal to be identified.

Methods for the development and evaluation of coding spaces are discussed in Chapter 4. The search algorithms are covered in Chapter 5. For the creation of the coding space, the best approach is to start with a small list of dimensions which are expected to be reliable discriminators among the different signals. Familiarity with the signals in the database is important when deciding what this set should be. The use of dimensions which can only take on a small number of discrete values is discouraged. It has been observed that continuous dimensions work better than discrete dimensions. After creating the initial set of dimensions, then each dimension in the set is evaluated based on the SNR defined in Equation 4.1. Other measures are available for the evaluation of individual dimensions but SNR worked the best.

In addition to the SNR of each dimension, the correlation coefficient between each pair of dimensions should also be computed. Correlations between dimensions in the space should be kept low if the information content added by each dimension is to be maximized. Once the high correlation problem is resolved, dimensions are selected in order of decreasing SNR value. Any number of dimensions may be selected. However, it was observed that as the number of dimensions increases, the performance of the search algorithms goes up, reaches a maximum and starts to decrease again. For the database used in this thesis the maximum performance was obtained with a 6-dimensional coding space. Based on the SNR values of the dimensions, one may conjecture that the search algorithms work best when no dimensions with less than half the SNR of the best dimension are allowed in the space. However, the generality of this is unknown.

For the coding space used in this work, the approximate maximum size of the potential single-dolphin vocabulary was estimated to be over one billion. This is based on estimates of coding space and average cluster volume computed for a small (92 whistles in 8 clusters) database. Two estimates of the potential vocabulary size were found based on different approximations to the average cluster volume. The other estimate, based on extrapolating the maximum potential vocabulary size for coding spaces of 5 dimensions or less to the 16-dimensional space, was about 10^{17} whistles. The true potential vocabulary size is expected to lie somewhere between the two estimates.

The above potential vocabulary estimate is based on whistles which are the signature whistles of the animals making the sound. If one includes mimicry in this vocabulary estimate, the clusters should be bigger and the potential vocabulary estimate smaller. The reason for the increase in cluster volume is because the animals' frequency control has been observed to be as much as three times worse during mimicry, and therefore the variability along each frequency-based dimension can be 3 times larger. Since there are 14 frequency-based dimensions in the coding space, the cluster volume can be 6 orders of magnitude larger (assuming independent dimensions). This results in a potential vocabulary estimate larger than 300 whistles. As seen in Figure 4-5 one estimate of the potential shared-dolphin vocabulary is quite stable (around a few hundred whistles) for coding spaces of 6 or more dimensions.

Two different search algorithms were compared in this thesis. They were the $k-d$ tree optimized by Friedman [9] and the expanding bucket. Although the expanding bucket requires a lower number of records (whistles) to be examined for both finding and confirming the nearest neighbors, its search time was never better than that of the $k-d$ tree. The best times for both algorithms were comparable (35 ms vs. 53 ms for a 1169-whistle database) but the $k-d$ tree was faster. However, one may speculate that if the algorithms are implemented in a computer system without a math

coprocessor, the lower number of records required for the expanding bucket search will make it come out ahead.

For the (approximately) 1000-whistle database a search time of 50 ms (expanding bucket) means that the system can recognize about 1200 whistles per minute. If the content of the database is expanded to 50000 whistles and one assumes search time scales linearly so that each search takes an average of 2.5 seconds, the system can then process about 24 whistles per minute. A human, with a similar vocabulary capacity (50000 words) can process at least 250 words per minute. Our system is an order of magnitude slower.

Also, when the search is constrained by a budget, the performance of the expanding bucket can be superior to that of the $k-d$ tree. The reason for this is that the expanding bucket finds (encounters) the nearest neighbour faster than the $k-d$ tree algorithm. Thus, the expanding bucket has a better chance of having the correct result by the time the budget runs out and the search is stopped.

As to how the two search algorithms scale with respect to database size, the following was discovered. The average number of records examined by the expanding bucket algorithm during a search seems to be linear with respect to the database size. The line has a slope of approximately 0.03, which means that for large databases one can expected to search about 3 percent of the database. For the $k-d$ tree, the growth in the average number of records examined per search was also linear with respect to database size, with a slope of about 0.19.

6.2 Future Work

Given the experimental aspects of this thesis, at many times implementation issues dominated the decisions. It would be interesting to go back and re-examine some issues without concern for speed of implementation or complexity.

One example is the power density spectrum estimator used in level I. Recall that Burg's AR spectral estimation technique was evaluated and dismissed because it did not offer substantially better performance for the complexity. However, the use of AR techniques could potentially solve many of the problems encountered while doing frequency-vs.-time trace estimation. Also, it may be possible to use the AR model order as another silence detector. For example, a model order above 4 might indicate signal rather than background.

Another example is the tracing algorithm. Implementation of the optimal solution and comparison of that solution with the "real" trace (and the suboptimal trace) is worth exploring. An optimal tracing algorithm will make the level II compression autonomous by making the manual trace editing unnecessary. Also, such an algorithm can be of use in other analysis software used by WHOI scientists.

The relation of search efficiency to the dimensionality of the space is an area which can yield important results. It has already been shown that lower dimensional spaces (less than 16 dimensions) result in faster searches. The SNR of the individual dimensions seems the most likely candidate to be responsible for this behavior. The relation between SNR and optimal coding space size should be explored further.

However, the most obvious direction for future work is the application of the coding and search techniques covered in Chapters 4 and 5 to other areas. The techniques described are generic in nature. All that is required to apply them to new areas is the user's knowledge regarding what dimensions make sense to use in forming the coding space. The system can be used for other biological signals, even classification of other marine mammals. Another area where this work can be of use is in diagnostic applications, where a large database of signal-condition pairs is available and the condition of something (engine, heart, pipeline, etc.) should be determined based on a current signal sample. Exploring such examples could be rewarding.

Appendix A

Source Code - Declarations

```
/* Copyright (c) 1992, 1993 Kevin G. Christian
 * All rights reserved.
 *
 * Level 3 database include file. Declarations.
 *
 * Note: When compiling for the SUN, don't forget to define 'UNIX'.
 *
 *
 * THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES      10
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */

/*********************************************************************
 *                         enum declarations                         *
 *********************************************************************/
/*
  Note: The order of the dimensions listed below determines how the data
  in the ovariance, dvariance (db_dim.c), and dim_th (db_find.c) is
  interpreted. These arrays will need changes if the list below is changed.   20
*/
enum dim_names_enum {                    /* Dimension ids in SNR order */
 eigenvector2=0,
 mean_plus_median_freq,
 eigenvector1,
 freq_range_ave,
 wduration,
 eigenvector3,
 eigenvector4,
 eigenvector5,                                                                30
```

```
  median_1st_der,
  mean_minus_median_freq,
  cross_rate_meanf,
  duty_cycle,
  eigenvector7,
  mean_1st_der,
  eigenvector6,
  eigenvector8,
  ndimensions                          /* Number of dimensions */
  };                                                                         40
enum db_type_enum {
  db_kdtree,
  db_expand
  };
#define db_type db_kdtree              /* Format of data base */

/*********************************************************************
 *                     Typedef declarations                          *
 *********************************************************************/
typedef char Tag[64];                                                        50
typedef float Dimension;
typedef float *Ftrace;
typedef struct Tag_record_struct {
  Tag tag;
  long usage;
} Tag_record;
typedef struct Record_struct {
  Dimension dim[ndimensions];
  long tag_id;
  long gid;                                                                  60
} Record;
typedef struct Whistle_struct {
  Dimension dim[ndimensions];
  Ftrace trace;
  long duration;
} Whistle;
#ifndef UNIX
typedef struct {
  short dx;           /* X distance of upper left corner from (0,0)     */
  short dy;           /* Y distance of upper left corner from (0,0)     */   70
  short sizex;        /* Width of window                                */
  short sizey;        /* Height of window                               */
  float x_axs[2];     /* X axis values                                  */
  float y_axs[2];     /* Y axis values                                  */
} G_WINDOW;
#endif /* UNIX */
```

```
/*********************************************************************
 *                     Subroutine Declarations                      *
 *********************************************************************/      80
/* db_main.c */
int redisplay_dimensions(void);

/* db_io.c */
char init_kl(char *fname);
float project_vectorn(float *trace, long d, short ev);
char init_lv2(char *fname);
long read_trace(long number, Whistle *output);
void db_write_matlab(char *fname);
char read_db_header(FILE *ptr, short *nd, char *type, long *rtot, long *ttot);   90
char db_read(char *fname);
char db_write(char *fname);

/* db_dim.c */
char compute_dimensions(Ftrace trace, long d, Dimension *dim);
void copy_dimensions(Dimension *src, Dimension *dest);
void dim_add(Record *new);
void dim_delete(Record *old);
char update_variances(void);
void update_weights(void);                                                        100

/* db_find.c */
void init_find(void);
void update_find_distances(void);
long find_node(Record *target);
struct tree_info_struct {
  long levels;
  long nbuckets;
  long ubuckets;
  long maxbucket;                                                                 110
  long minbucket;
  float avebucket;
} tree_info(void);
struct recdist {
  long rec_id;
  float distance;
  long found_at;
  long added_at;
  long oper_at;
} * (*find)(Record *target, short depth);                                         120
void (*find_add)(long rec_id);
void (*find_delete)(long rec_id);
struct recdist *find_raw(Record *target, short depth);

/* db_screen.c */
void init_screen(void);
void end_screen(void);
void supdate_dbfile(char *name, char do_refresh);
```

```
void supdate_rtotal(long rtotal, long ttotal, char do_refresh);
void supdate_rnumber(long number, char do_refresh);                                   130
void supdate_rdimensions(float *data, char do_refresh);
void supdate_wnumber(long wn, char do_refresh);
void supdate_wdimensions(float *data, char do_refresh);
void supdate_tag(Tag tag, char do_refresh);
void supdate_file(char *fname, char do_refresh);
void supdate_find(char *type, long target, struct recdist *queue, short depth,
                 long searches, long ops, long op_sum, long conf_at, char do_refresh);
void bell(void);
void error_message(char *m);
void notify_message(char *m);                                                         140
void message(short number, char *lines[ ]);
char *read_line(char *prompt, char *def, char do_history);

/* db_utils.c */
void oops(char *m);
char *check_malloc(unsigned long bytes, char *var);
char hit_escape(void);
float stopwatch(short start);
char *string_copy(char *s);
char is_prefix(const char *s1, const char *s2);                                       150
#ifndef UNIX
char *index(char *s, char c);
void stopit(int sig);
#endif /* UNIX */
long first_in_range(char *range, long last);
long last_in_range(char *range, long last);
long next_in_range(void);
void abort_range(void);
void delete_tag(long tag_id);
long add_tag(Tag t);                                                                  160
void extend_records(void);
void extend_stored(void);
void clear_stored(void);

#ifndef UNIX
/* db_graph.c */
void reset_dos_window(G_WINDOW *gwin);
void erase_dos_window(G_WINDOW gwin);
short init_display(void);
void disable_display(void);                                                           170
void draw_cursor(short row, short col);
short g_window_position(G_WINDOW *win, float dxp, float dyp, float sizexp, float sizeyp);
short g_window_clear(G_WINDOW win);
short g_window_labels(G_WINDOW win, char axis, short how_many, short precision, short tlen);
short g_window_ticks(G_WINDOW win, char axis, short how_many, short tlen);
short g_window_plot(G_WINDOW win, float data[ ], short n, short clear);
#endif /* UNIX */
```

```
/*****************************************************************         180
 *           Definitions for command input and parsing               *
 *****************************************************************/
#define HSIZE 10                          /* Size of history stack */
#define COMMAND_PROMPT "DB_LV3> " /* Program prompt */
#define ENTER_KEY 13
#define ESCAPE 27
#define BKSPACE 8
#define DELETE 127
#define TAB '\t'
                                                                             190
#ifndef UNIX
#define UP_KEY 'H'
#define DOWN_KEY 'P'
#define RIGHT_KEY 'M'
#define LEFT_KEY 'K'
#else
#define UP_KEY 'A'
#define DOWN_KEY 'B'
#define RIGHT_KEY 'C'
#define LEFT_KEY 'D'                                                         200
#endif /* UNIX */

/*****************************************************************
 *                       Other definitions                           *
 *****************************************************************/
#define VERSION "Release 0.9 (January 25, 1993)"
#ifdef UNIX
#define FREE(_x) {free((void *)_x); _x=NULL;}
#define DDELAY 400000
#else                                                                        210
#define FREE(_x) {free((char *)_x); _x=NULL;}
#define DDELAY  50000
#endif /* UNIX */

#define CORRUPT_DBFILE " CORRUPT"  /* Code word to indicate invalid DB */
#define VAR_TOLERANCE 0.20         /* Tolerable % deviation in variance */
#define ZC_LPERCENT 0.05           /* % of wstl len zro-cros must last */
#define CRANGE 5                   /* how many to average in freq range */
#define DB_BLKPWR 7                /* Power of 2 for blocking factor */
#define DB_BLOCK (1<<DB_BLKPWR)  /* DB blocking factor - POWER OF 2 */      220
/* Macros to access records and tags. Done this way because a straight
   linear array wouldn't fit in the PC */
#define record(_x) (dbase[(_x)>>DB_BLKPWR][(_x)&(DB_BLOCK-1)])
#define tag(_x) (dbase_tags[(_x)>>DB_BLKPWR][(_x)&(DB_BLOCK-1)])
```

Appendix B

Source Code - Tracing Algorithm

Function list

Function name	Line number
binary_search	40
cleanup_peaks	161
end_path	304
find_bandwidths	124
find_peaks	93
find_trace	237
is_harmonic	54
link_peaks	323
pcostfcn	286
peak_dumping	214
select_background	578
trace_dumping	187

```
/* Copyright (c) 1991 Kevin G. Christian
 * All rights reserved.
 *
 * Routine to convert a whistle list into a curve. 7/3/91
 * From the whistle list, peaks are selected from each power spectrum
 * and placed into a peaks array.  Then, the peak array is traversed by
 * a tracing algorithm selecting one single peak to be part of the trace.
 *
 * THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES      10
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */
#include <stdio.h>
#ifndef APOLLO
#include <malloc.h>
#else
free(char *);
#endif /* APOLLO */
#ifdef UNIX
int fprintf(FILE *, char *, ...);                                      20
int fclose(FILE *);
int abs(int);
void bcopy(char *, char *, int);
#else
#include <stdlib.h>
#include <string.h>
#endif /* UNIX */

#include "level2.h"
                                                                       30

#define ATT_TOL_G 10
#define ATT_TOL_L 50
#define CPL 28                          /* characters per line */

static float link_peaks(peaks far *, long, struct peak_pair, double, short *);
static void fixup_trace(peaks far *, short *, long, short);

/* Return the location where value should go to keep plist ordered */
static short binary_search(peaks far plist, short s, short e, float value)   40
{
  short half=(s+e)/2;
  if (s == e) return e;

  if (value>=plist[half].mag)
    return binary_search(plist,s,half,value);
  else
    return binary_search(plist,half+1,e,value);
}
```

```
                                                                          50

/* Try to determine if indices are harmonically related. To avoid
   being fooled by noise, only up to the 4th harmonic will be accepted. */
static short is_harmonic(Findx index1, Findx index2)
{
  short temp;

  if (index1<0 || index2<0) oops("is_harmonic");
  if (index1==0 || index2==0) return 0;
  if (index1 > index2) {                                                  60
    temp = index1;
    index1 = index2;
    index2 = temp;
  }

  if (--index1 != 0) {
    temp = index2%index1;
    if ((temp==0 && index2!=index1 && index2/index1<5) ||
        (temp==1 && (index2-1)!=index1 && (index2-1)/index1<5) ||
          (temp==(index1-1) && (index2+1)!=index1 && (index2+1)/index1<5))  70
          return 1;
  }
  if (++index1 != 0) {
    temp = index2%index1;
    if ((temp==0 && index2!=index1 && index2/index1<5) ||
        (temp==1 && (index2-1)!=index1 && (index2-1)/index1<5) ||
          (temp==(index1-1) && (index2+1)!=index1 && (index2+1)/index1<5))
          return 1;
  }
  if (++index1 != 0) {                                                    80
    temp = index2%index1;
    if ((temp==0 && index2!=index1 && index2/index1<5) ||
          (temp==1 && (index2-1)!=index1 && (index2-1)/index1<5) ||
          (temp==(index1-1) && (index2+1)!=index1) && (index2+1)/index1<5)
          return 1;
  }
  return 0;
}

                                                                          90
/* Find peaks in power spectrum.  In the case of a flat portion, the
   peak is the last frequency in the flat segment. */
static void find_peaks(peaks far plist, float *data, short l)
{
  short j,k;
  float pvalue;

  if (plist==NULL || data==NULL || l<=0) oops("find_peaks");

  for (j=0; j<PEAKS; j++) {                                              100
```

```
    plist[j].index = 0;
    plist[j].mag = 0.0;
  }

  pvalue = data[0];
  for (j=1; j<l-1; j++) {
    short inpos;
    if (pvalue<data[j] && data[j]>data[j+1]) {
      if (data[j] <= plist[PEAKS-1].mag) continue;
      inpos = binary_search(plist,0,PEAKS-1,data[j]);                    110
      for (k=PEAKS-1; k>inpos; k--) plist[k] = plist[k-1];
      plist[inpos].mag = data[j];
      plist[inpos].index = j;
    }
    if (data[j] == data[j+1]) continue;
    pvalue = data[j];
  }

  return;
}                                                                        120


/* Find bandwidth of each peak in the list. */
static void find_bandwidths(peaks far plist, float *data, short l)
{
  short k;
  Findx bw;

  if (plist==NULL || data==NULL || l<=0) oops("find_bandwidths");
                                                                         130
  for (k=0; k<PEAKS && plist[k].index!=0; k++) {
    bw = plist[k].index-1;
    while (1) {    /* do left side */
      if (bw < 1) break;
      if (bw < 2) {
           if (data[bw] <= data[bw-1]) break;
      } else {
           if (data[bw]<=data[bw-1] && data[bw]<=data[bw-2]) break;
      }
      --bw;                                                              140
    }
    plist[k].lbw = bw;

    bw = plist[k].index+1;
    while (1) {    /* do right side */
      if (bw>l-2) break;
      if (bw>l-3) {
           if (data[bw]<=data[bw+1]) break;
      } else {
           if (data[bw]<=data[bw+1] && data[bw]<=data[bw+2]) break;     150
      }
```

```
      ++bw;
    }
    plist[k].hbw = bw;
  }
  return;
}

/* Resolve problem of overlapping bandwidths, it assumes peaks are in
   decreasing order */                                                     160
static void cleanup_peaks(peaks far plist)
{
  short j,k,i=0,absorbed=0;

  if (plist==NULL) oops("cleanup_peaks");

  while(i<PEAKS-1) {
    if (plist[i].index == 0) break;
    for (j=i+1; j<PEAKS && plist[j].index!=0; j++) {
      if (plist[j].index<=plist[i].hbw && plist[j].index>=plist[i].lbw){     170
          plist[i].lbw=plist[i].lbw<plist[j].lbw ? plist[i].lbw : plist[j].lbw;
          plist[i].hbw=plist[i].hbw>plist[j].hbw ? plist[i].hbw : plist[j].hbw;
          for (k=j; k<PEAKS-1; k++) plist[k] = plist[k+1];
          plist[PEAKS-1].index = 0;
          plist[PEAKS-1].mag = 0.0;
          absorbed = 1;
      }
    }
    if (!absorbed) ++i;
    else absorbed = 0;                                                      180
  }
  return;
}


/* Put the trace in an ASCII file.  Add a few things to make it readable */
static void trace_dumping(Findx *trace, peaks far *plist, short len, long dur, float qual)
{
  FILE *fl;
  short j;                                                                  190

  if (trace==NULL || plist==NULL || len<=0 || dur<=0 || qual<0) oops("trace dumping");

  fl = fopen(DUMP_TRACE_FILE,"a");
  if (fl == NULL) oops("can't open trace file");

  /* Put whistle info out first */
  fprintf(fl,"Trace dump (len, duration, quality):  %d %ld %f\n",len,dur,qual);
  fprintf(fl, "number trace max tr_ampl max_ampl\n");
  /* Then whistle trace: number, trace, maximum, trace ampl, max ampl */    200
  for (j=0; j<dur; j++) {
    short k;
```

```
    fprintf(fl, "%d %d %d ", j, trace[j], plist[j][0].index);
    for (k=0; trace[j]!=plist[j][k].index; k++) ;
    fprintf(fl, "%lf %lf\n", k<PEAKS?plist[j][k].mag:0.0, plist[j][0].mag);
  }

  fclose(fl);
  return;
}                                                                    210


/* Put the trace in an ASCII file.  Add a few things to make it readable */
static void peak_dumping(peaks far *plist, short npeaks, long dur)
{
  FILE *fl;
  short j,k;

  if (plist==NULL || npeaks<=0 || dur<=0) oops("peak dumping");
                                                                     220
  fl = fopen(DUMP_TRACE_FILE,"a");
  if (fl == NULL) oops("can't open trace file");

  fprintf(fl, "Peak List\n");
  for (j=0; j<dur; j++) {
    fprintf(fl, "%d (%lf)", j, plist[j][0].mag);
    for (k=0; k<npeaks; k++) fprintf(fl, " %d", plist[j][k].index);
    fprintf(fl,"\n");
  }
                                                                     230
  fclose(fl);
  return;
}


/* Given a whistle list find the trace */
Findx *find_trace(struct whstl far *w, Qdata *bkg, short l, long d, peaks far **plist)
{
  static double globalmax=0.0;
  extern char dump_trace;                                            240
  peaks far *peak_list;
  struct peak_pair localmax;
  Findx *trace;
  short j,k;
  float *data, quality;

  if (w==NULL || bkg==NULL || l<=0 || d<=0) oops("find_trace");

  /* Initialize */
  peak_list = (peaks far *)check_fmalloc(d*sizeof(peaks), "peak_list");   250
  trace = (Findx *)check_malloc(d*sizeof(Findx),"trace");
  localmax.mag = 0.0;
```

```
  /* Make peak list */
  for (j=0; j<d; j++, w=w→next) {
    if (w==NULL || w→qps==NULL) oops("trace died traversing whistle");
    data = float_Qdata_minus_background(w→qps, bkg, l);
    find_peaks(peak_list[j], data, l);
    find_bandwidths(peak_list[j], data, l);
    cleanup_peaks(peak_list[j]);                                          260
    FREE(data);

    if (peak_list[j][0].mag > localmax.mag) {           /* update maxs */
      localmax.mag = peak_list[j][0].mag;
      localmax.index = j;
    }
    if (localmax.mag > globalmax) globalmax = localmax.mag;
  }

  /* Make trace */                                                        270
  quality = link_peaks(peak_list, d, localmax, globalmax, trace);
  fixup_trace(peak_list, trace, d, l);

  if (dump_trace) {
    trace_dumping(trace, peak_list, l, d, quality);
    peak_dumping(peak_list, PEAKS, d);
  }
  *plist = peak_list;
  return trace;
}                                                                         280


/* Tracing algorithm "predictor based" cost function. The desired end point,
   p3, is predicted given p1 and p2 already in trace. Cost is the difference
   in value. */
static double pcostfcn(Findx p1, Findx p2, Findx p3, short idxp)
{
  double cost,penalty=1.0 + (double)idxp/PEAKS;

  if (p1<0 || p2<0 || p3<0 || idxp<0) oops("pcostfcn");                 290

  /* linear prediction */
  if (!p1 && p2) return ((double)abs(p3-p2)*penalty);
  if (p1 && !p2) return ((double)abs(p3-p1)*penalty);
  cost = (double)abs(2*p2 - p1 - p3)*penalty;
  return cost;
}
```

```
/* Routine to determine if it's time to stop the path (found silence??)            300
   Decision is based on comparing; a. the current peak magnitude (cmag) to
   the local max in the path, b. the current min cost (min1) to a minimum
   average trace cost (min2/m). */
static char end_path(double cmag, double lmax, double *ncost, double *cost, short m)
{
  double min1,min2;
  short j;

  if (ncost==NULL || cost==NULL || m<=0) oops("end_path");
  min1 = ncost[0];                                                                   310
  min2 = cost[0];
  for (j=1; j<PEAKS; j++) {
    if (min1 > ncost[j]) min1 = ncost[j];
    if (min2 > cost[j]) min2 = cost[j];
  }

  if (cmag*ATT_TOL_G<lmax && min1>(2.0*min2/m)) return 1;
  return 0;
}
                                                                                     320

/* Tracing algorithm */
static float link_peaks(peaks far *plist, long d, struct peak_pair localm, double globalm,
                        Findx *trace)
{
#ifndef UNIX
 extern G_WINDOW wsp; /* spectrogram window */
#endif /* UNIX */
 struct peak_pair segm;   /* mag/index of largest peak in segment */
 double cost[PEAKS],      /* previous trace costs (j-1) */                         330
        ncost[PEAKS],     /* current trace costs (j) */
        micost[PEAKS],    /* minimum incremental trace costs (j-1:j) */
        min,              /* temporary variable to hold min value */
        ccost=0;          /* cummulative cost of trace */
 char  *used;             /* keep track of points used in trace */
 Findx *list_space,       /* chunk of memory for holding trace lists */
       *list[PEAKS],      /* previous traces (j-1) */
       *nlist[PEAKS];     /* current traces (j) */
 short mindex,            /* temporary variable, index of minimum */
       mseg,              /* length of trace lists */                                340
       first, last,       /* limits of segment traced [first,last) */
       j, k, l;           /* general purpose variables */

 /* check inputs */
 if (plist==NULL || d<=0 || localm.index<0 || globalm<0 || trace==NULL) oops("link_peaks");

 /* Allocate space */
 used = (char *)check_calloc(d, sizeof(char), "used array");

 while(1) { /* repeat until no more segments are found */                           350
```

```
/* find segment bounds and segm (segment max) */
for (j=0; j<d && used[j]; j++) ;
first = j;
if (first == d) break; /* all used */
segm.mag = plist[first][0].mag;
segm.index = first;
for (j=first+1; j<d && !used[j]; j++) {
  if (plist[j][0].mag > segm.mag) {
        segm.mag = plist[j][0].mag;                                    360
        segm.index = j;
  }
}
last = j;

/* handle too short segments */
if (last-first<OVERHANG) {
  for (j=first; j<last; j++) {
        used[j] = 1;
        trace[j] = 0;                                                  370
  }
  continue;
}

/* handle too weak segments */
if (segm.mag*ATT_TOL_G < localm.mag) {
  for (j=first; j<last; j++) {
        used[j] = 1;
        trace[j] = 0;
  }                                                                    380
  continue;
}

/*  a long, strong segment. Just what the doctor ordered. */
mseg = segm.index>(first+last)/2 ? segm.index-first+1 : last-segm.index;
list_space = (Findx *)check_malloc(++mseg*2L*PEAKS*sizeof(Findx), "list_space");

/* Going forward */
for (j=0; j<PEAKS; j++) {
  list[j] = list_space + j*mseg;                                       390
  nlist[j] = list[j] + PEAKS*mseg;
}
if (segm.index+1 >= last) goto bkw;          /* skip part if needed */
for (j=0; j<PEAKS; j++) {                    /* initialize cost */
  list[j][0] = plist[segm.index][0].index;
  list[j][1] = plist[segm.index+1][j].index;
  cost[j] = pcostfcn(0, list[j][0], plist[segm.index+1][j].index, j);
}
used[segm.index] = used[segm.index+1] = 1;
                                                                       400
for (j=segm.index+2; j<last; j++) {          /* follow trail */
```

```
    short m=j-segm.index;
    if (!plist[j][0].index) {                    /* no peaks in list (ps=bkg) */
         last = j;                               /* break the segment */
         break;
    }
    for (k=0; k<PEAKS; k++) {                    /* k=end peak */

         if (!plist[j][k].index) {               /* no peak to end at */
           nlist[k] = NULL;                                                        410
           continue;
         }

         l=mindex=0;
         if (!list[l]) oops("no peak in last window?!?");
         min=ncost[k] = cost[l] +
           pcostfcn(list[l][m-2], list[l][m-1], plist[j][k].index, k);
         for (l=1; l<PEAKS; l++) {               /* l=attempted trace */
           if (!list[l]) continue;
           ncost[k] = cost[l] +                                                    420
             pcostfcn(list[l][m-2], list[l][m-1], plist[j][k].index, k);
           if (min > ncost[k]) {
             min = ncost[k];
             mindex = l;
           }
         }

         ncost[k] = min;                  /* keep track of minimal cost */
         micost[k] = min - cost[mindex];
#ifdef UNIX                                                                        430
      bcopy((char *)list[mindex],(char *)nlist[k],mseg*sizeof(Findx));
#else
         nlist[k] = memcpy(nlist[k],list[mindex],mseg*sizeof(Findx));
#endif /* UNIX */
         nlist[k][m] = plist[j][k].index;
    } /* minimum path ending at each peak found by this point */

    /* is it time to end the path? */
    if (end_path(plist[j][0].mag,localm.mag,micost,cost,m)) {
         last = j;                                                                 440
         break;
    }

    for (k=0; k<PEAKS; k++) {             /* transfer cost and trace lists */
         Findx *temp;
         cost[k] = ncost[k];
         temp = list[k];
         list[k] = nlist[k];
         /* make sure nlist[k] remains a valid storage place! */
         if (temp != NULL) nlist[k] = temp;                                        450
         else nlist[k] = nlist[0] + mseg*k;
    }
```

```
    used[j] = 1;
  }

  /* Done with forward part of segment. Add in crossover cost */
  if (last<d && used[last] && trace[last]!=0) {
    short m=last-segm.index;
    for (l=0; l<PEAKS; l++) {                    /* l=attempted trace */
        if (!list[l]) continue;                                              460
        cost[l] += pcostfcn(list[l][m-2], list[l][m-1], trace[last], 0);
    }
  }

  /* Find optimal trace using the cost from last window in segment.
     Copy optimal trace to 'trace'. */
  min = cost[mindex=0];
  for (j=1; j<PEAKS; j++) {
    if (!list[j]) continue;
    if (min > cost[j]) {                                                     470
        min = cost[j];
        mindex = j;
    }
  }
  for (j=segm.index,k=0; j<last; j++) trace[j] = list[mindex][k++];
  ccost += min;

  /* Going backwards */
bkw:for (j=0; j<PEAKS; j++) {
    list[j] = list_space + j*mseg;                                           480
    nlist[j] = list[j] + PEAKS*mseg;
  }
  if (segm.index-1 < first) goto fin;
  for (j=0; j<PEAKS; j++) {
    list[j][0] = plist[segm.index][0].index;
    list[j][1] = plist[segm.index-1][j].index;
    cost[j] = pcostfcn(0, list[j][0], plist[segm.index-1][j].index, j);
  }
  used[segm.index] = used[segm.index-1] = 1;
                                                                             490
  for (j=segm.index-2; j>=first; j--) {          /* follow trail */
    short m=segm.index-j;
    if (!plist[j][0].index) {                    /* no peaks in list (ps=bkg) */
        first = j+1;                             /* break the segment */
        break;
    }
    for (k=0; k<PEAKS; k++) {                    /* k=end peak */

        if (!plist[j][k].index) {                /* no peak to end at */
          nlist[k] = NULL;                                                   500
          continue;
        }
        l=mindex=0;
```

```
        if (!list[l]) oops("no peak in last window?!?");
        min=ncost[k] = cost[l] +
          pcostfcn(list[l][m-2], list[l][m-1], plist[j][k].index, k);
        for (l=1; l<PEAKS; l++) {                    /* l=attempted trace */
          if (!list[l]) continue;
          ncost[k] = cost[l] +
            pcostfcn(list[l][m-2], list[l][m-1], plist[j][k].index, k);        510
          if (min > ncost[k]) {
            min = ncost[k];
            mindex = l;
          }
        }

        ncost[k] = min;                         /* keep track of minimal cost */
        micost[k] = min - cost[mindex];
#ifdef UNIX
        bcopy((char *)list[mindex],(char *)nlist[k],mseg*sizeof(Findx));         520
#else
        nlist[k] = memcpy(nlist[k],list[mindex],mseg*sizeof(Findx));
#endif /* UNIX */
        nlist[k][m] = plist[j][k].index;
    }

    /* is it time to end the path? */
    if (end_path(plist[j][0].mag,localm.mag,micost,cost,m)) {
        first = j+1;
        break;                                                                   530
    }

    for (k=0; k<PEAKS; k++) {               /* transfer cost and trace lists */
        Findx *temp;
        cost[k] = ncost[k];
        temp = list[k];
        list[k] = nlist[k];
        /* make sure nlist[k] remains a valid storage place! */
        if (temp != NULL) nlist[k] = temp;
        else nlist[k] = nlist[0] + mseg*k;                                       540
    }
    used[j] = 1;
  }

  /* Done with backward part of segment. Add in crossover cost. */
  if (first>0 && used[first-1] && trace[first-1]!=0) {
    short m=segm.index-first+1;
    for (l=0; l<PEAKS; l++) {                        /* l=attempted trace */
        if (!list[l]) continue;
        cost[l] += pcostfcn(list[l][m-2], list[l][m-1], trace[first-1], 0);      550
    }
  }

  /* Find optimal trace using the cost from last window in segment.
```

```
      Copy optimal trace to 'trace'. */
    min = cost[mindex=0];
    for (j=1; j<PEAKS; j++) {
      if (!list[j]) continue;
      if (min > cost[j]) {
           min = cost[j];                                          560
           mindex = j;
      }
    }
    for (j=segm.index,k=0; j>=first; j--) trace[j] = list[mindex][k++];
    ccost += min;

fin:FREE(list_space);
  }

  FREE(used);                                                      570
  return (float)(ccost/d);
}


/* Check if new background has whistle in it.  Right now only way to tell
for sure is by checking if peaks are harmonically related. If a better and
more general method is found, it should be used here too. */
Qdata *select_background(Qdata *old, Qdata *new, short l)
{
  peaks bpeaks;                                                    580
  float *fnew;

  if (new == NULL) oops("select background");
  if (old == NULL) return new;

  fnew = float_Qdata(new, l);
  find_peaks(bpeaks, fnew, l);
  find_bandwidths(bpeaks, fnew, l);
  cleanup_peaks(bpeaks);
  FREE(fnew);                                                      590

  if (is_harmonic(bpeaks[0].index, bpeaks[1].index)) {
    FREE(new);
    return(old);
  } else {
    FREE(old);
    return(new);
  }
}
```

Appendix C

Source Code - Search Algorithms

Function list

Function name	Line number
ball_within_bounds	336
bounds_overlap_ball	354
build_node	443
build_tree	490
expand_sorted_dimension	575
find1	260
find2	656
find_add1	198
find_add2	601
find_delete1	228
find_delete2	628
find_node	185
find_raw	57
init_find	835
renumber_last	856
reset_sorted_dimension	559
reset_tree	163
search	389
sorted_search	587
tree_info	518
update_queue	303

```
/* Copyright (c) 1992, 1993 Kevin G. Christian
 * All rights reserved.
 *
 * Level 3 subroutines for finding closest whistle in data base. The
 * k-d tree algorithm is implemented by the 'find1' subroutine and the
 * expanding bucket by the 'find2' subroutine.
 *
 *
 * Note: I have assumed that record #0 only gets added to an empty DB.
 *       Thus, find_add clears it's internal structures when it gets asked to      10
 *       add record #0.
 *
 * THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include "db_lv3.h"                                                                20

#ifndef UNIX
#undef HUGE
#define HUGE 3.4e38
#endif /* UNIX */

/* Subroutine Declarations (system) */

/* Subroutine Declarations (local) */
static void search(long node);                                                     30
static void renumber_last(long rec_id);

/* Global Variables (common) */
unsigned long operations = 0;          /* # of multiply-adds */
unsigned long op_sum = 0;              /* cummulative # of multiply-adds */
unsigned long searches = 0;            /* # of searches done */
long budget = 0;                       /* computational budget,ignore if 0 */
long found_at=0;                       /* when was element found */
#ifdef DEBUG
float find_time=0;                     /* time (secs) of last find */              40
#endif /* DEBUG */

/* Local Variables (common) */
static struct recdist *queue=NULL;     /* list of recs and distance pairs */
static long rdnum=0;                   /* # allocated to queue */
static long added_at=0;                /* when was element added to queue */
static Record gtarget;                 /* global target */
static short gdepth=0;                 /* # of neighbors to find */
```

```
                                                                       50
/*******************************************************************
 *                        Full Search Method                        *
 *                                                                  *
 * Search the entire db one record at a time. Implemented for      *
 * comparison purposes only.                                        *
 *******************************************************************/
struct recdist *find_raw(Record *target, short depth)
{
  extern Record **dbase;
  extern long rtotal;                                                  60
  extern float *dweight;
  float dist, temp;
  short j, k, l;

  if (depth<=0) oops("find_raw");

  /* Prepare queue */
  if (rdnum < depth) {
    if (queue != NULL) FREE(queue);
    rdnum = depth;                                                     70
    queue = (struct recdist *)check_malloc((long)depth*sizeof(struct recdist), "find queue");
  }
  for (j=0; j<depth; j++) queue[j].distance = (float)HUGE;

  /* Compute distance and see if it should go into queue */
  for (k=0; k<rtotal; k++) {
    for (dist=0.0, j=0; j<ndimensions; j++) {
      temp = record(k).dim[j] - target→dim[j];
      dist += temp*temp/dweight[j];
    }                                                                  80
    dist = sqrt(dist);

    for (j=0; j<depth; j++)
      if (dist < queue[j].distance) {
          for (l=depth-1; l>j; l--) queue[l] = queue[l-1];
          queue[j].rec_id = k;
          queue[j].distance = dist;
          queue[j].found_at = queue[j].added_at = k;
          queue[j].oper_at = 0;
          break;                                                       90
      }
  }
  return queue;
}
```

```
/*********************************************************************
 *                        k-d Tree Support                           *
 *                                                                   *
 * Implementation of an optimized k-d tree as described in:          *
 * J.H. Friedman, J.L. Bentley & R.A. Finkel. An Algorithm for Finding *     100
 * Best Matches in Logarithmic Expected Time. ACM Trans on Math Soft. *
 * Vol 3, No 3, Sept 1977, pp 209-226                                *
 *********************************************************************/

typedef struct frec_struct frec;          /* find-record structure */
struct frec_struct {
 long rec_id;
 frec *next;
#ifdef DEBUG
 long gid;                                                                   110
#endif /* DEBUG */
};
typedef union tree_node_union {           /* k-d tree node */
 struct node {
  long dim_id;
  Dimension threshold;
 } n;
 struct leaf {
  long bcount;
  frec *ptr;                                                                 120
 } l;
} tnode;

/*
#define L_SON(_x) (2*_x + 1)
#define R_SON(_x) (2*(_x+1))
I've modified the kdtree to make the root node be location 1.
*/
#define L_SON(_x) (_x<<1)
#define R_SON(_x) ((_x<<1) + 1)                                              130
#define tree_levels 10                /* number of levels in tree */
#define mx_duplication 2              /* # of times a dim. may be used in tree */

static tnode *kdtree = NULL;              /* k-d tree */
static long tree_size = (1<<(tree_levels+1));/* allocation of kdtree array */
static Dimension bounds[ndimensions][2];  /* quadrant bounds (0:upper 1:lower) */
static char done=0;                       /* flag to indicate search is done */

static long node_ids[tree_levels] = {     /* dim. being split at each level */
 eigenvector2, mean_plus_median_freq, eigenvector1, eigenvector2,           140
 mean_plus_median_freq, freq_range_ave, wduration, eigenvector3,
 eigenvector4, duty_cycle};
```

```
static Dimension dim_th[ndimensions][(1<<mx_duplication)-1] = {
  {-2.47822949218750e03,  2.32041491699219e03,  5.15524609375000e03}, /*  1 */
  { 1.65100410156250e04,  1.94444755859375e04,  2.10188193359375e04},
  {-3.42333605957031e03,  9.49136425781250e03,  1.83064746093750e04},
  { 5.83984375000000e03,  7.65625000000000e03,  1.18066406250000e04},
  { 0.59000000000000e02,  1.00500000000000e02,  1.23500000000000e02}, /*  5 */
  {-1.05558525390625e04, -0.11920003662109e04,  0.97090327148438e04},
  {-3.27658447265625e03, -1.00513757324219e03,  8.88266601562500e03},          150
  {-7.57022753906250e03,  1.06381439208984e03,  5.64623339843750e03},
  { 0.00000000000000000,  4.76837158203125000,  9.53674316406250000},
  {-1.02124169921875e03,  0.24944970703125e03,  0.68306201171875e03}, /* 10 */
  { 0.01639344170690000,  0.03390804678202000,  0.04620512388647000},
  {-0.43019481003284000,  0.14193363487720000,  0.44747653603554000},
  {-8.69503906250000e02,  1.17019067382812e03,  3.32283312988281e03},
  { 2.49913682937622e01,  3.29626750946045e01,  3.65627841949463e01},
  {-1.45921234130859e03,  0.70457702636719e03,  2.64211865234375e03}, /* 15 */
  {-1.84552960205078e03,  0.28040019989014e03,  1.59640124511719e03}
};                                                                              160

/* Empty all the buckets in the tree */
static void reset_tree(void)
{
  long node;
  frec *hold;

#ifdef DEBUG
  for (node=1; node<(tree_size>>1); node++)
    if (kdtree[node].n.dim_id < 0) oops("Misplaced leaf.");                    170
#endif /* DEBUG */

  for (node=tree_size>>1; node<tree_size; node++) {
    kdtree[node].l.bcount = -1L;
    while (kdtree[node].l.ptr != NULL) {
      hold = kdtree[node].l.ptr->next;
      FREE(kdtree[node].l.ptr);
      kdtree[node].l.ptr = hold;
    }
  }                                                                             180
  return;
}

/* Given a record, return the node number where it belongs. */
long find_node(Record *target)
{
  long node;

  if (target==NULL || kdtree==NULL) oops("find_node");
  for (node=1; kdtree[node].n.dim_id>=0; ) {                                    190
    if (target->dim[kdtree[node].n.dim_id] <= kdtree[node].n.threshold) node = L_SON(node);
    else node = R_SON(node);
  }
```

```
  return node;
}

/* Add a new record to the tree */
void find_add1(long rec_id)
{
  extern Record **dbase;                                                    200
  extern long rtotal;
  Record new;
  frec *rec;
  long node;

  if (rec_id<0 || rec_id>rtotal || kdtree==NULL) oops("find_add1");

  /* If adding record #0, clear internal structure first. */
  if (rec_id == 0) reset_tree();
                                                                            210
  new = record(rec_id);
  node = find_node(&new);

  rec = (frec *)check_malloc(1L*sizeof(frec), "tree entry");
  rec→rec_id = rec_id;
  rec→next = kdtree[node].l.ptr;
  kdtree[node].l.ptr = rec;
  kdtree[node].l.bcount−−;

#ifdef DEBUG                                                                220
  rec→gid = record(rec_id).gid;
#endif

  return;
}

/* Delete a record from the tree */
void find_delete1(long rec_id)
{
  extern Record **dbase;                                                    230
  extern long rtotal;
  Record target;
  frec **rhandle, *rec;
  long node;

  if (rec_id<0 || rec_id>=rtotal || kdtree==NULL) oops("find_delete1");

  target = record(rec_id);
  node = find_node(&target);
                                                                            240
  for (rhandle=&kdtree[node].l.ptr; *rhandle!=NULL; rhandle=&(rec→next)) {
    rec = *rhandle;
#ifdef DEBUG
    if (record(rec→rec_id).gid != rec→gid) oops("synch failure");
```

```
#endif /* DEBUG */
    if (rec→rec_id == rec_id) {
      *rhandle = rec→next;
      FREE(rec);
      kdtree[node].l.bcount++;
      renumber_last(rec_id);                                              250
      return;
    }
  }

  oops("find_delete didn't find record");
  return;
}

/* Make the necessary calls to find the closest whistles to target. */
struct recdist *find1(Record *target, short depth)                         260
{
  short j;

  if (depth<=0 || kdtree==NULL) oops("find1");

  if (rdnum < depth) {
    if (queue != NULL) FREE(queue);
    rdnum = depth;
    queue = (struct recdist *)check_malloc((long)depth*sizeof(struct recdist), "find queue");
  }                                                                          270
  for (j=0; j<depth; j++) {
    queue[j].distance = (float)HUGE;
    queue[j].rec_id = 0;
  }
  for (j=0; j<ndimensions; j++) {
    bounds[j][0] = (Dimension)HUGE;
    bounds[j][1] = -bounds[j][0];
  }

  update_weights();                                                          280
  gtarget = *target;
  gdepth = depth;
  done = 0;
  found_at = added_at = operations = 0;
  searches++;
#ifdef DEBUG
  {
    char buf[64];
    stopwatch(1);
    search(1L);                                                              290
    sprintf(buf, "Elapsed search time:  %g secs.", (find_time=stopwatch(0)));
/*    notify_message(buf); */
  }
#else
  search(1L);
```

```
#endif /* DEBUG */
  op_sum += operations;

  return queue;
}                                                                    300

/* Update queue list. queue[0] is closest element */
static void update_queue(long rec_id)
{
  extern Record **dbase;
  extern float *dweight;
  Record ltarget;
  float dist, temp;
  short j, k;
                                                                     310
  ltarget = record(rec_id);
  for (dist=0.0, j=0; j<ndimensions; j++) {
    operations++;
    temp = ltarget.dim[j] − gtarget.dim[j];
    dist += temp*temp/dweight[j];
  }
  operations++;
  dist = sqrt(dist);

  found_at++;                                                        320
  for (j=0; j<gdepth; j++)
    if (dist < queue[j].distance) {
      for (k=gdepth−1; k>j; k−−) queue[k] = queue[k−1];
      queue[j].rec_id = rec_id;
      queue[j].distance = dist;
      queue[j].added_at = ++added_at;
      queue[j].found_at = found_at;
      queue[j].oper_at = operations;
      break;
    }                                                                330
  return;
}

/* Determine if a ball with radius equal to the distance to the mth
   closest record to target is fully contained in 'quadrant' */
static char ball_within_bounds(void)
{
  extern float *dweight;
  float t;
  short j;                                                           340

  for (j=0; j<ndimensions; j++) {
#ifndef UNIX
    if (queue[gdepth−1].distance >= 0.9*HUGE) return 0;
#endif /* UNIX */
    t = queue[gdepth−1].distance * sqrt(dweight[j]);
```

```
    if ((bounds[j][0]-gtarget.dim[j]) < t || (gtarget.dim[j]-bounds[j][1]) < t) return 0;
  }
  return 1;
}                                                                                    350

/* Determine if any of the bounds crosses a ball with radius equal to the
   distance to the mth closest record to target. */
static char bounds_overlap_ball(void)
{
  extern float *dweight;
  float dsum=0.0, t, temp;
  short j;

#ifdef UNIX                                                                          360
  if (queue[gdepth-1].distance < HUGE)
#else
  if (queue[gdepth-1].distance < 0.9*HUGE)
#endif /* UNIX */
    t = queue[gdepth-1].distance*queue[gdepth-1].distance;
  else
    t = queue[gdepth-1].distance;

  for (j=0; j<ndimensions; j++) {
#ifdef DEBUG                                                                         370
    if (bounds[j][0] <= bounds[j][1]) oops("corrupted bounds");
#endif /* DEBUG */
    if (gtarget.dim[j] < bounds[j][1]) {              /* lower than lower bound */
      temp = gtarget.dim[j]-bounds[j][1];
      dsum += temp*temp/dweight[j];
      operations++;
    } else if (gtarget.dim[j] > bounds[j][0]) {       /* higher than upper bound */
      temp = bounds[j][0]-gtarget.dim[j];
      dsum += temp*temp/dweight[j];
      operations++;                                                                  380
    }
    if (dsum > t) return 0;
  }

  return 1;
}

/* Main search routine */
static void search(long node)
{                                                                                    390
  frec *rec;
  long d=kdtree[node].n.dim_id;
  Dimension t=kdtree[node].n.threshold, temp;

  if (node<=0 || node>=tree_size) oops("search");
  if (budget && operations>budget) done = 1;
  if (done) return; /* kgc */
```

```
  /* If node is terminal, search every record in bucket */
  if (d < 0) {                                                          400
    for (rec=kdtree[node].l.ptr; rec!=NULL; rec=rec→next) {
      if (budget && operations>budget) { done = 1; break;}
      update_queue(rec→rec_id);
    }
    if (kdtree[node].l.ptr && ball_within_bounds()) done = 1;
    return;
  }

  /* If node is not terminal, go down the tree */
  if (gtarget.dim[d] <= t) {             /* search the left son */      410
    temp = bounds[d][0];
    bounds[d][0] = t;
    search(L_SON(node));
    bounds[d][0] = temp;
  } else {                               /* search the right son */
    temp = bounds[d][1];
    bounds[d][1] = t;
    search(R_SON(node));
    bounds[d][1] = temp;
  }                                                                     420

  if (done) return;

  /* I went down but I'm still searching. Go for path not travelled */
  if (gtarget.dim[d] <= t) {             /* now consider right son */
    temp = bounds[d][1];
    bounds[d][1] = t;
    if (bounds_overlap_ball()) search(R_SON(node));
    bounds[d][1] = temp;
  } else {                               /* now consider left son */    430
    temp = bounds[d][0];
    bounds[d][0] = t;
    if (bounds_overlap_ball()) search(L_SON(node));
    bounds[d][0] = temp;
  }

  if (ball_within_bounds()) done = 1; /* kgc */

  return;
}                                                                       440

/* Determine if a node is internal or a leaf and fill appropriately. */
static void build_node(long node)
{
  short level;

  if (node<=0 || node>=tree_size) oops("build_node");
```

```
  for (level=0; node >= (1<<level); level++) ;
  level--;                                                        450

#ifdef DEBUG
  if (level<0 || level>tree_levels) oops("Invalid level.");
#endif /* DEBUG */

  if (level<tree_levels) {              /* internal node */
    Dimension t;
    float temp;
    long dim = node_ids[level];
    short lo, hi, asize=(1<<mx_duplication)-1;                    460

    for (lo=0; dim_th[dim][lo]<=bounds[dim][1] && lo<asize; lo++) ;
    for (hi=asize-1; dim_th[dim][hi]>=bounds[dim][0] && hi>=0; hi--) ;
#ifdef DEBUG
    if (hi<lo) oops("Exceeded mx_duplication specification.");
    if ((hi-lo)%2) oops("Failure in build_node");
#endif /* DEBUG */
    t = dim_th[dim][(hi+lo)/2];

    kdtree[node].n.dim_id = dim;                                  470
    kdtree[node].n.threshold = t;

    temp = bounds[dim][0];
    bounds[dim][0] = t;
    build_node(L_SON(node));
    bounds[dim][0] = temp;

    temp = bounds[dim][1];
    bounds[dim][1] = t;
    build_node(R_SON(node));                                      480
    bounds[dim][1] = temp;
  } else {                              /* leaf node */
    kdtree[node].l.bcount = -1L;
    kdtree[node].l.ptr = NULL;
  }
  return;
}

/* Initialize k-d tree. */
static void build_tree(void)                                      490
{
  long j;

  if (tree_levels<0 || tree_levels>24) oops("init_tree");

  /* Allocate space for tree */
  kdtree = (tnode *)check_malloc(tree_size*sizeof(tnode), "k-d tree");

  /* Fill nodes of tree */
```

```
  for (j=0; j<ndimensions; j++) {                                          500
    bounds[j][0] = (Dimension)HUGE;
    bounds[j][1] = -bounds[j][0];
  }
  build_node(1L);

#ifdef DEBUG
  for (j=1; j<(tree_size>>1); j++)
    if (kdtree[j].n.dim_id<0 || kdtree[j].n.dim_id>=ndimensions) oops("Invalid tree.");
  for (j=(tree_size>>1); j<tree_size; j++)
    if (kdtree[j].l.bcount != -1) oops("Invalid tree.");                  510
#endif /* DEBUG */

  return;
}


/* Make tree info public */
struct tree_info_struct tree_info(void)
{
  extern long rtotal;                                                      520
  struct tree_info_struct td;
  long j, use;

  td.levels = tree_levels;
  td.nbuckets = tree_size>>1;
  td.ubuckets = 0;
  td.maxbucket = 0;
  td.minbucket = rtotal;
  td.avebucket = (float)0.0;
                                                                           530
  for (j=tree_size>>1; j<tree_size; j++) {
    if ( (use=-kdtree[j].l.bcount-1) > 0) {
      td.ubuckets++;
      if (use>td.maxbucket) td.maxbucket = use;
      if (use<td.minbucket) td.minbucket = use;
      td.avebucket += use;
    }
  }

  if (td.ubuckets) td.avebucket /= (float)td.ubuckets;                     540
  return td;
}
```

```
/*********************************************************************
 *                    Expanding bucket scheme                        *
 *                                                                   *
 * The search is conducted by slowly expanding a search 'bucket'     *
 * centered at the target whistle. Expansion is done along single    *
 * dimensions.                                                       *
 *********************************************************************/        550
#define S_BLOCK 128                        /* increment for sorted array */
static long *sorted_dimension[ndimensions]; /* rec id's sorted along dim */
                                           /* 0 - smallest */
static long s_size=0, s_alloc=0;           /* size,alloc of sorted array*/
static char *used;                         /* mark whistles */
static long *member;                       /* 1 = whistle a member in dim */

/* Initial allocation of sorted arrays */
static void reset_sorted_dimension(void)
{                                                                                 560
  extern long rtotal;
  short j;

  s_alloc = rtotal/S_BLOCK + 1;
  s_alloc *= S_BLOCK;
  for (j=0; j<ndimensions; j++) {
    if (sorted_dimension[j] != NULL) FREE(sorted_dimension[j]);
    sorted_dimension[j] = (long *)check_malloc(s_alloc*sizeof(long), "sorted array in find");
  }
  s_size = 0;                                                                     570
  return;
}

/* Expansion of the sorted arrays */
static void expand_sorted_dimension()
{
  short j;
  s_alloc += S_BLOCK;
  for (j=0; j<ndimensions; j++) {
    sorted_dimension[j] = (long *)realloc(sorted_dimension[j], s_alloc*sizeof(long));   580
    if (sorted_dimension[j] == NULL) oops("Died expanding sorted_dimension.");
  }
  return;
}

/* Binary search of an array sorted by value */
static long sorted_search(float value, short d, long lo, long hi)
{
  extern Record **dbase;
  long middle;                                                                    590

  for (middle=(lo+hi)/2; hi-lo>0; middle=(hi+lo)/2) {
    if (value > record(sorted_dimension[d][middle]).dim[d]) lo = middle+1;
    else hi = middle;
```

```
  }

  return hi;
}

/* Add a new record to the sorted arrays */                                    600
void find_add2(long rec_id)
{
 extern Record **dbase;
 extern long rtotal;
 long pos;
 register long *s, *d;
 short j;

 if (rec_id<0 || rec_id>rtotal) oops("find_add2");
                                                                               610
 if (rec_id==0) reset_sorted_dimension();
 if (s_size >= s_alloc) expand_sorted_dimension();

 for (j=0; j<ndimensions; j++) {
   if (s_size==0 || record(rec_id).dim[j]<=record(sorted_dimension[j][0]).dim[j]) pos = 0;
   else if (record(rec_id).dim[j] >= record(sorted_dimension[j][s_size-1]).dim[j]) pos = s_size;
   else pos = sorted_search(record(rec_id).dim[j], j, 0L, s_size-1);

   for (s=&sorted_dimension[j][s_size-1], d=&sorted_dimension[j][s_size];
          s>=&sorted_dimension[j][pos]; s--, d--)  *d = *s;                    620
   sorted_dimension[j][pos] = rec_id;
 }
 s_size++;
 return;
}

/* Delete a record from the sorted arrays */
void find_delete2(long rec_id)
{
 extern Record **dbase;                                                        630
 extern long rtotal;
 long pos;
 register long *s, *d;
 short j;

 if (rec_id<0 || rec_id>=rtotal || s_size==0) oops("find_delete2");

 for (j=0; j<ndimensions; j++) {
  /* Remove deleted record from the array */
  if (sorted_dimension[j][0] == rec_id) pos = 0;                               640
  else if (sorted_dimension[j][s_size-1] == rec_id) pos = s_size-1;
  else pos = sorted_search(record(rec_id).dim[j],j,0L,s_size-1);
  while (sorted_dimension[j][pos]!=rec_id && pos<s_size-1) ++pos;
  if (sorted_dimension[j][pos] != rec_id) oops("delete2 fail (1)");
  for (s=&sorted_dimension[j][pos+1], d=&sorted_dimension[j][pos];
```

```
        s<=&sorted_dimension[j][s_size-1]; s++, d++)  *d = *s;
  }
  /* rename last record to rec_id */
  s_size--;
  renumber_last(rec_id);                                                    650
  return;
}

/* Find closest whistle using expanding bucket approach */
/* This version expands the bucket ONE whistle in each direction. */
struct recdist *find2(Record *target, short depth)
{
  extern Record **dbase;
  extern float *dweight;
  extern long rtotal;                                                       660
  float min_c[ndimensions], psum, temp, hold;
  long sbound[ndimensions][2], mbound[ndimensions][2], left, k, whistle;
  long inside=(1<<(ndimensions));
  register long bound, *lp;
  short j;

  if (depth<=0 || s_alloc==0 || s_size>rtotal || ndimensions>30) oops("find2");

  /* Setup the queue where values are returned */
  if (rdnum < depth) {                                                      670
    if (queue != NULL) FREE(queue);
    rdnum = depth;
    queue = (struct recdist *)check_malloc((long)depth*sizeof(struct recdist), "find queue");
  }
  for (j=0; j<depth; j++) {
    queue[j].distance = (float)HUGE;
    queue[j].rec_id = 0;
  }

  /* Initialize some variables and arrays */                                680
  gtarget = *target; gdepth = depth;
  update_weights();
  found_at = added_at = operations = 0;
  left = s_size;
#ifdef DEBUG
  stopwatch(1);
#endif /* DEBUG */
  used = (char *)calloc(rtotal,sizeof(char));
  member = (long *)malloc(rtotal*sizeof(long));
  if (member==NULL || used==NULL) oops("No memory for arrays in find2");   690
  for (lp=member; lp<&member[rtotal]; lp++) *lp = 1;
  for (j=0; j<ndimensions; j++) {
    k=sbound[j][0]=sbound[j][1] = sorted_search(target->dim[j],j,0L,s_size-1);
    temp = target->dim[j]-record(sorted_dimension[j][k]).dim[j];
    min_c[j] = temp*temp/dweight[j];
    mbound[j][0] = mbound[j][1] = k;
```

```
  operations++; if (budget && operations>budget) {left=0; break;}
 }

 while (left) {                                                          700

  for (j=0; j<ndimensions && left; j++) {

   /* Handle the lower bound */
   if (!used[whistle=sorted_dimension[j][bound=sbound[j][0]]]) {
        member[whistle] <<= 1;
        if (member[whistle] >= inside) {
          hold = min_c[j];
          temp = target->dim[j]-record(whistle).dim[j];
          min_c[j] = temp*temp/dweight[j];                               710
          operations++; if (budget && operations>budget) {left=0; break;}
          for (psum=0.0, k=0; k<ndimensions; k++) psum += min_c[k];
          operations++; if (budget && operations>budget) {left=0; break;}
          /* See if we can ignore ALL whistles below current */
          if (sqrt(psum) >= queue[depth-1].distance) {
            /* Yes we can! */
            for (k=bound; k>=0; k--) {
              char *p = &used[sorted_dimension[j][k]];
              if (!(*p)) { left--;  *p = 1; }
            }                                                            720
            /* Reflect on upper end */
            if ( (temp=target->dim[j]+target->dim[j]-record(whistle).dim[j]) <=
                  record(sorted_dimension[j][s_size-1]).dim[j]) {
              for (k = sorted_search(temp, j, sbound[j][1],s_size-1); k<=s_size-1; k++) {
                  char *p = &used[sorted_dimension[j][k]];
                  if (!(*p)) { left--;  *p = 1; }
              }
            }
          } else {
            /* No we can't. */                                           730
            update_queue(whistle);
            if (budget && operations>budget) {left=0;break;}
            used[whistle] = 1;
            left--;
          }
          min_c[j] = hold;
          /* The minimum contribution of a dimension is the smallest distance
             between the target and the closest UNUSED whistle along the
             specific dimension. Once a whistle gets used, all min_c must be
             checked to see if they are based on the whistle that got USED. */   740
          for (k=0; k<ndimensions; k++) {
            lp = mbound[k];
            if (sorted_dimension[k][*lp] == whistle) {
              for ( ;*lp>0 && used[sorted_dimension[k][*lp]]; ) --(*lp);
              temp = fabs(target->dim[k] - record(sorted_dimension[k][*lp]).dim[k]);
              for (++lp; *lp<s_size-1 && used[sorted_dimension[j][*lp]]; ) ++(*lp);
              if (temp < fabs(target->dim[k] - record(sorted_dimension[k][*lp]).dim[k])) {
```

```
                min_c[k] = temp*temp/dweight[k];
                operations++; if (budget && operations>budget) {left=0; break;}
            }                                                                   750
          }
        }
      }
  }

  /* Handle the upper bound */
  if (!used[whistle=sorted_dimension[j][bound=sbound[j][1]]]) {
      member[whistle] <<= 1;
      if (member[whistle] >= inside) {
        hold = min_c[j];                                                        760
        temp = target->dim[j]-record(whistle).dim[j];
        min_c[j] = temp*temp/dweight[j];
        operations++; if (budget && operations>budget) {left=0; break;}
        for (psum=0.0, k=0; k<ndimensions; k++) psum += min_c[k];
        operations++; if (budget && operations>budget) {left=0; break;}
        /* See if we can ignore ALL whistles above current */
        if (sqrt(psum) >= queue[depth-1].distance) {
          /* Yes we can ! */
          for (k=bound; k<=s_size-1; k++) {
            char *p = &used[sorted_dimension[j][k]];                            770
            if (!(*p)) { left--; *p = 1; }
          }
          /* Reflect on lower end */
          if ( (temp=target->dim[j]+target->dim[j]-record(whistle).dim[j]) >=
                record(sorted_dimension[j][0]).dim[j]) {
            for (k = sorted_search(temp,j,0L,sbound[j][0]) - 1; k>=0; k--) {
                char *p = &used[sorted_dimension[j][k]];
                if (!(*p)) { left--;  *p = 1; }
            }
          }                                                                     780
        } else {
          /* No we can't. */
          update_queue(whistle);
          if (budget && operations>budget) {left=0;break;}
          used[whistle] = 1;
          left--;
        }
        min_c[j] = hold;
        /* The minimum contribution of a dimension is the smallest distance
           between the target and the closest UNUSED whistle along the          790
           specific dimension. Once a whistle gets used, all min_c must be
           checked to see if they are based on the whistle that got USED. */
        for (k=0; k<ndimensions; k++) {
          lp = &mbound[k][1];
          if (sorted_dimension[k][*lp] == whistle) {
            for (;*lp<s_size-1 && used[sorted_dimension[k][*lp]];) ++(*lp);
            temp = fabs(target->dim[k] - record(sorted_dimension[k][*lp]).dim[k]);
            for (--lp; *lp>0 && used[sorted_dimension[k][*lp]]; ) --(*lp);
```

```
            if (temp < fabs(target->dim[k] - record(sorted_dimension[k][*lp]).dim[k])) {
                min_c[k] = temp*temp/dweight[k];                                          800
                operations++; if (budget && operations>budget) {left=0; break;}
            }
          }
        }
      }
    }
    /* Expand the bucket */
    lp = sbound[j];
    if (*lp > 0) --(*lp);
    for ( ; *lp>0 && used[sorted_dimension[j][*lp]]; --(*lp)) ;                          810
    if ( *(++lp) < s_size-1) ++(*lp);
    for ( ; *lp<s_size-1 && used[sorted_dimension[j][*lp]]; ++(*lp)) ;
  }
 }

 FREE(used);
 FREE(member);
#ifdef DEBUG
 {
  char buf[64];                                                                           820
  sprintf(buf, "Elapsed search time:  %g secs.",(find_time=stopwatch(0)));
/*   notify_message(buf); */
 }
#endif /* DEBUG */
 searches++;
 op_sum += operations;
 return queue;
}

/**********************************************************************                  830
 *                   Miscellaneous Utilities                          *
 **********************************************************************/

/* Setup fcn pointer to use correct method. */
void init_find(void)
{
 switch(db_type) {
 case db_kdtree:
  find = find1;
  find_delete = find_delete1;                                                             840
  find_add = find_add1;
  build_tree();
  break;
 case db_expand:
  find = find2;
  find_delete = find_delete2;
  find_add = find_add2;
  break;
 default:
```

```
    oops("Unknown db_type.");                                              850
  }
  return;
}

/* Pass given rec_id to last record. Used after a delete. */
static void renumber_last(long rec_id)
{
  extern Record **dbase;
  extern long rtotal;
                                                                           860
  if (rec_id == rtotal-1) return;
  switch (db_type) {
  case db_kdtree:
   {
     frec *rec;
     Record target;
     long node;

     target = record(rtotal-1);
     node = find_node(&target);                                            870
     for (rec=kdtree[node].l.ptr; rec!=NULL; rec=rec→next)
          if (rec→rec_id == rtotal-1) {
            rec→rec_id = rec_id;
            return;
          }
     oops("Re-numbering operation failed.");
   }
   break;
  case db_expand:
   {                                                                       880
     long target=rtotal-1, pos;
     short j;
     for (j=0; j<ndimensions; j++) {
          if (sorted_dimension[j][0] == target) pos = 0;
          else if (sorted_dimension[j][s_size-1] == target) pos = s_size-1;
          else pos = sorted_search(record(target).dim[j],j,0L,s_size-1);
          while (sorted_dimension[j][pos]!=target && pos<s_size-1) ++pos;
          if (sorted_dimension[j][pos] != target) ocps("Re-numbering operation failed.");
          sorted_dimension[j][pos] = rec_id;
     }                                                                     890
   }
   break;
  default:
   oops("Unknown db type.");
   break;
  }
  return;
}
```

Bibliography

[1] Whitlow W. L. Au, Robert W. Floyd, and Jeffrey E. Haun. Propagation of Atlantic bottlenose dolphin echolocation signals. *Journal of the Acoustical Society of America*, 64(2):411–422, August 1978.

[2] Carlos Cabrera. *Evaluation of Measurements on a Frequency Trace Representation of Dolphin Whistles*, August 1992. 6.961 Introduction to Research in EECS - Final Report.

[3] M.C. Caldwell, D.K. Caldwell, and P.L. Tyack. A review of the signature whistle hypothesis for the Atlantic bottlenose dolphin, *Tursiops truncatus.* In S. Leatherwood and R. Reeves, editors, *The bottlenose dolphin: recent progress in research*, pages 199–234. Academic Press, San Diego, CA, 1990.

[4] Christopher W. Clark, Peter Marler, and Kim Beeman. Quantitative analysis of animal vocal phonology: an application to swamp sparrow song. *Ethology*, 76:101–115, 1987.

[5] Douglas Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, June 1979.

[6] C.J. Date. *An Introduction to Database Systems*, volume 1 of *The Systems Programming Series.* Addison-Wesley, Reading, MA, 4 edition, 1986.

[7] H.C. Du and R.C.T. Lee. Symbolic gray code as a multikey hashing function. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, PAMI-2(1):83–90, January 1980.

[8] P. Cheesman *et. al.* AUTOCLASS: A Bayesian classification system. In *Proc. Fifth Machine Learning Workshop*, pages 54–64, 1988.

[9] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions in Mathematical Software*, 3(3):209–226, September 1977.

[10] King Sun Fu. *Syntactic pattern recognition and applications.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[11] Cheong K. Gan and Robert W. Donaldson. Adaptive silence deletion for speech storage and voice mail applications. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(6):924–927, June 1988.

[12] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[13] Terrance Howald. Personal communication. Author at the Woods Hole Oceanographic Institution, May 1993.

[14] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. Inst. Elec. Radio Eng.*, 40(9):1098–1101, 1952.

[15] Fumitada Itakura. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-23:67–72, February 1975.

[16] N. S. Jayant and P. Noll. *Digital Coding of Waveforms: principles and applications to speech and video.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[17] Richard A. Johnson and Dean W. Wichern. *Applied Multivariate Statistical Analysis.* Prentice-Hall, 2 edition, 1988.

[18] J. F. Lynch Jr., J. G. Josenhans, and R. E. Crochiere. Speech/silence segmentation for real-time coding via rule based adaptive endpoint detection. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, 1987.

[19] Steven M. Kay. *Modern Spectral Estimation: theory and application.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[20] Steven M. Kay and Stanley L. Marple. Spectrum analysis - a modern perspective. *Proceedings of the IEEE*, 69(11):1380–1419, November 1981.

[21] Lori F. Lamel, Lawrence R. Rabiner, Aaron E. Rosenberg, and Jay G. Wilpon. An improved endpoint detector for isolated word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-29(4):777–785, August 1981.

[22] Robert E. Larson and John L. Casti. *Principles of Dynamic Programming: Part I. Basic Analytical and Computational Methods.* Marcel Dekker, Inc., New York, NY, 1978.

[23] Kai-Fu Lee and Hsiao-Weun Hon. Large-vocabulary speaker-independent continuous speech recognition using HMM. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 123–126, 1988.

[24] Richard P. Lippmann. Review of neural networks for speech recognition. *Neural Computation*, 1:1–38, 1989.

[25] J. Makhoul, S. Roucos, and H. Gish. Vector quantization in speech coding. *Proceedings of the IEEE*, November 1985.

[26] Stanley L. Marple. *Digital Spectral Analysis: with applications.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[27] W.D. Maurer and T.G. Lewis. Hash table methods. *Computing Surveys*, 7(1):5–19, March 1975.

[28] S. Hamid Nawab and Thomas F. Quatieri. Signal reconstruction from short-time fourier transform magnitude. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-31(4):986–998, August 1983.

[29] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*, chapter 11.6. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[30] Douglas O'Shaughnessy. *Speech Communication: Human and Machine*, chapter 10. Addison-Wesley, Reading, MA, 1987.

[31] L.R. Rabiner and B.H. Juang. An introduction to hidden Markov models. *IEEE Acoustics, Speech, and Signal Processing Magazine*, pages 4–16, January 1986.

[32] L.R. Rabiner and R.W. Schafer. *Digital Processing of Speech Signals*, chapter 7. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[33] Douglas G. Richards, James P. Wolz, and Louis M. Herman. Vocal mimicry of computer-generated sounds and vocal labeling of objects by bottlenose dolphin, *Tursiops truncatus. Journal of Comparative Psychology*, 98(1):10–28, 1984.

[34] Laela S. Sayigh, Peter L. Tyack, Randall S. Wells, and Michael D. Scott. Signature whistles of free-ranging bottlenose dolphins *Tursiops truncatus*: stability and mother-offspring comparisons. *Behavioral Ecology and Sociobiology*, 26:247–260, 1990.

[35] Stephanie Seneff. TINA: A probabilistic syntactic parser for speech understanding systems. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 711–714, 1989.

[36] Dennis Shasha and Tson-Li Wang. New techniques for best-match retrieval. *ACM Transactions on Information Systems*, 8(2):140–158, April 1990.

[37] Harry F. Smith. *Data Structures: Form and Function*, chapter 6. Harcourt Brace Jovanich, 1987.

[38] Peter De Souza. A statistical approach to the design of an adaptive self-normalizing silence detector. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-31(3):678–684, June 1983.

[39] Stephen F. Weiss. A probabilistic algorithm for nearest neighbour searching. In R.N. Oddy, S.E. Robertson, C.J. van Rijsbergen, and P.W. Williams, editors, *Information Retrieval Research*, chapter 21, pages 325–333. Butterworths, Boston, MA, 1981.

[40] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications od the ACM*, 30(6):520–540, June 1987.

[41] Thomas P. Yunck. A technique to identify nearest neighbours. In Belur V. Dasarathy, editor, *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques*, chapter 7, pages 341–346. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[42] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, September 1978.