Goldsmiths University of London

Accelerometer-Based Gesture Recognition with the iPhone

by

Marco Klingmann

Supervisor: *Dr. Nikolay Nikolaev*

MSc in Cognitive Computing

London, September 2009

Abstract

The growing number of small sensors built into consumer electronic devices, such as mobile phones, allow experiments with alternative interaction methods in favour of more physical, intuitive and pervasive human computer interaction. This paper examines hand gestures as an alternative or supplementary input modality for mobile devices. The iPhone is chosen as sensing and processing device. Based on its built-in accelerometer, hand movements are detected and classified into previously trained gestures. A software library for accelerometer-based gesture recognition and a demonstration iPhone application have been developed. The system allows the training and recognition of free-from hand gestures. Discrete hidden Markov models form the core part of the gesture recognition apparatus. Five test gestures have been defined and used to evaluate the performance of the application. The evaluation shows that with 10 training repetitions, an average recognition rate of over 90 percent can be achieved.

Acknowledgements

I would like to thank the Goldsmiths staff, Dr. Nikolay Nikolaev and Dr. Mark Bishop for their support and guidance.

And everyone else, who helped me during this master thesis: Abraham Alaka, Andrea Früh, Annina Klingmann, Christoph Burgdorfer, Jürg Lehni, Roland Früh, Stephen Breakey and my parents.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Gestures, such as a wave or a head nod, form a natural part of our face-to-face communication[1]. While gestures are most often just supporting our verbal communication, they can also by themselves be a simple and very effective way of communication. Although ubiquitous in communication between people, the gesture modality has been mostly ignored in human-computer interaction.

The idea of a computer, that responds to hand gestures instead of to speech, a keyboard or a mouse or, was popularised through the film Minority Report (2002) and its famous scene where the protagonist (Tom Cruise) makes objects fly around with just a sweep of his arm (c.f. Saffer, 2008). However, in academic research, the concept of gestures as a communication channel has been around for decades. In the mid 1970s, the artist and trained computer scientist Myron Krueger, created with VIDEOPLACE one of the earliest interactive systems that responds to gestures (Kalawsky, 1993).

The advances in microelectronics in recent years have reduced the costs of numerous small and precise sensors. This has inspired a lot of research effort around multimodal interfaces. The resulting prototypes usually remain confined to special applications like sign language interpretation (c.f. Vogler and Metaxas, 2001). These small sensors are also finding their way into an increasing number of consumer electronic devices, such as mobile phones and game consoles. With the release of its game console Wii in 2006, Nintendo made the concept of controlling games via (more or less) natural gestures available to a broad public. The Wii controller device contains an embedded accelerometer sensor, allowing the detection of the controller's movements.

In June 2009, Microsoft announced with Project Natal a multimodal interaction system that supports speech recognition and full body gesture recognition and motion tracking.

Mobile devices, such as mobile phones, mobile gaming devices or wearable computers, provide new possibilities for communication and computing on the go, but they also introduce new problems due to

[1]More precisely, this is not strictly confined to face-to-face communication only. Some people also gesture heavily when they are talking on the phone. In televised communication, gestures are relevant and can have great communicative power.

small screens and input facilities. Noticing this and tying in with the research on multimodal interfaces, this project studies accelerometer-based gesture recognition with the iPhone. A hand gesture recognition application was implemented and experimented with, in order to serve as a possible supplementary or alternative interaction modality for mobile phones.

A gesture, such as a hand movement describing a circle, can not likely be repeated in exactly the same way. On top of that, the effect of normal sensor inaccuracy inevitably leads to dissimilar data samples, even if the former was possible. Therefore, 'intelligent' software algorithms, known as pattern recognition algorithms need to be employed. There is a wide variety of possible methods to choose from, including hybrid methods that combine several approaches. In this paper, so-called discrete hidden Markov models and the corresponding algorithms for training and recognition of gesture models are examined.

Accelerometer-based gesture recognition has been discussed in a number of publications (e.g. Kela et al., 2006; Prekopcsák et al., 2008) and with various input devices, such as the Wii remote[2] (Schlömer et al., 2008). One major difference to most of the existing papers is – in the herewith presented project, sensing and capture of the data as well as gesture recognition and training, is all done on the mobile device itself. Most other approaches transmit the sensor data to a nearby computer, where the training and recognition software is running, or perform off-line analysis of the data.

A short summary about different gesture recognition approaches is given in chapter 2 and relevant academic publications as well as commercial products are mentioned. Thereafter, various pattern recognition methods, which are relevant in the context of gesture recognition, are presented. Finally, hidden Markov models, the pattern recognition method utilised in this project, are introduced and the gesture recognition and training apparatus, developed in this project is outlined.

Chapter 3 provides a short theoretical introduction to Markov models and the corresponding software algorithms. Further methods and algorithms, which are of central relevance for this project, such as vector quantisation and classification with Bayes' rule, are presented as well.

Chapter 4 revisits the points presented in the theory chapter and discusses them from a more practical perspective. The theory can not provide well defined answers for all aspects of a real-world implementation. Certain configurations or values, such as the initial values for the vector quantisation process, need to be determined heuristically. Therefore, experiments are conducted in order to find suitable values. The final implementation is evaluated based on the achieved recognition results.

The software, developed in this project, is presented in chapter 5. The reusable iPhone library for accelerometer-based gesture recognition, ABGLib, denotes a concrete product of this paper. Its Objective-C programming interface is explained and the demonstration application ABGDemo, which was used during the implementation of the library is introduced as well.

In the last chapter (chapter 6), the project and its achievements are critically summarised. Possible refinements and further examinations are suggested for future continuations of this project.

[2]The Wii remote is the controller device for the Nintendo Wii game console

Chapter 2

Gesture Recognition and Methods of Pattern Recognition

2.1 Gesture Recognition

Automated gesture recognition has been investigated in various academic research projects, which yielded a number of practical applications and commercial products. This section provides a short summary about several sensor techniques that have been examined in the context of gesture recognition, thereafter, the narrower field of accelerometer-based gesture recognition is illuminated and a selection of relevant publications and commercial non-commercial products are cited.

Prior to the actual recognition of gestures, a solution must be found to automatically register the position or movements of the human body or body parts, such as arms and hands. This is sometimes also referred to as motion tracking and a variety of different techniques have been investigated concerning this task. The optical approach uses one or more video cameras to track coloured markers or skin coloured areas (e.g. Elmezain et al., 2008) in the video images. Sometimes self-organising methods, which extract motion trajectories from consecutive sequences of video images are applied in order to register body movement (e.g. Yang et al., 2002). This vision-based approach represents a vast field of research, the further exploration of which, however, lies beyond the scope of this paper.

Gesture recognition can also be realised based on magnetic motion tracking or mechanical, exoskeleton-based tracking (Bergamasco et al., 2007), but these methods are less common. More recently also acoustic methods have been explored. Based on the sounds generated by a pen Seniuk and Blostein (2009) examine the recognition of simple gestures, such as circling or scratch-out, but also the recognition of some handwritten characters and words.

In the context of this project, accelerometer-based gesture recognition is investigated. An accelerometer is small sensor, which can measure the acceleration of itself, or the device it is built-in respectively (see section 4.2.1 for further details). Based on the acceleration profile, which originates from the movement of the device, the classification into previously defined gestures is possible. This gesture recognition technique has been discussed in many publications and various input devices have been tried.

Accelerometer-based gesture recognition with a Nintendo Wii controller is explored by Schlömer et al. (2008). In their work, the sensor data is transmitted via Bluetooth from the controller device to a nearby PC, where the signal is processed. They also released the open-source Java library wiigee[1], which facilitates training and recognition of hand gestures, performed with a Wii controller. LiveMove by AiLive[2] is a commercial software product, which provides comparable functionality. It is aimed for game developers and provides gesture recognition methods for several accelerometer-equipped game controllers.

Similar to the work by Schlömer et al. (2008) for the Wii controller, in the paper by Prekopcsák (2008) a mobile phone is used to capture accelerometer data, transmit it via Bluetooth to a nearby computer, where the data is analysed and gesture classification is performed in quasi real-time. The paper compares two distinct pattern recognition methods, support vector machines and hidden Markov models. The results showed that both methods performed almost equally well. Support vector machines achieved an average recognition rate of 96 percent and hidden Markov models reached 97.6 precent.

While most recent approaches process the signal data on a separate computer, for the OpenMoko mobile phone family, an open-source software library[3] exists, which allows gesture recognition that runs on the device itself.

Kela et al. (2006) study accelerometer-based gesture recognition for controlling a television, a video recorder and lighting in a design environment. For a sensing device, they utilise a so-called SoapBox (Sensing, Operating and Activating Peripheral Box). This is a sensor device developed for research activities in the context of ubiquitous computing (c.f. Tuulari and Ylisaukko-oja, 2002). In Mäntylä (2001), the off-line gesture classification of pre-recorded accelerometer data is presented. The sample data is analysed with discrete hidden Markov models and the corresponding training and classification methods were implemented in MATLAB. One of the first dynamic gesture recognition systems was developed by Hofmann et al. (1998) for the recognition of 10 gestures of the German sign language. The data samples were processed off-line and it took several hours.

In this project, an iPhone software library for accelerometer-based gesture recognition and a demonstration iPhone application is developed and optimised for the iPhone's sensors and programming environment. At the time of writing, a comparable software library for gesture recognition on the iPhone is not known to exist.

[1]`http://www.wiigee.com/` (last visit 24.08.2009)
[2]http://www.ailive.net/ (last visit 24.08.2009)
[3]`http://wiki.openmoko.org/wiki/Gestures` (last visit 24.08.2009)

2.2 Pattern Recognition Methods

The recognition of accelerometer-based hand gestures is primarily a task of pattern recognition. As the iPhone can be freely moved in space – temporally varying, 3-dimensional signal data is obtained from the phone's 3-axis acceleration sensor (see 4.2.1). The sample data generated in this way, needs to be classified into a set of preliminary defined reference gestures. Gestures should be freely trainable, with minimal training effort. Data samples that do not match one of the predefined gestures well enough, should not be classified as one of them, but instead stated separately, as an unclassifiable, unknown gesture.

In previous university lectures by Dr. Nikolay Nikoloav[4] and related class works, basic knowledge about artificial neural networks (thereafter just referred to as neural networks), in particular multilayer perceptrons, has been acquired. As a consequence of this, a neural networks based method was initially favoured for the pattern recognition task.

The research phase for this project yielded a number of pattern recognition methods, which have proven successful in comparable tasks. Amongst them are artificial neural networks, in particular time-delay neural networks (Yang and Ahuja, 2001), dynamic time-warping (Corradini, 2001) and hidden markov models (Kela et al., 2006; Schlömer et al., 2008).

Dynamic time warping algorithms can measure the similarity between two sample sequences, which vary in time or speed. Good classification results can be achieved by using this method. The major drawback however is that, in order for the method to work reliably, for each gesture, a relatively high number of gesture data templates needs to be present. Therefore this method is only used for small alphabets of gestures; as the creation of good gesture templates is described as a relatively tedious process (Yang and Ahuja, 2001).

Neural networks have also been applied to similar tasks of classification, yielding good results. However, neural networks are generally not suitable to work on multi-dimensional sample data that varies over time. Their major flaw is that they typically operate with a fixed length time window, which imposes restrictions on the gestures. Extensive pre- and post-processing steps, such as the inter- and extrapolation of data samples, need to be employed in combination with special types of neural networks, such as time-delay neural networks, in order to achieve good quality classification results.

The third pattern recognition method, which was considered and eventually chosen, are so-called hidden Markov models (HMMs). They have been extensively studied primarily in the context of speech recognition. Hidden Markov models work well on temporal data that is not fixed in length. A number of training algorithms exists, allowing the automated creation of custom, free-form gestures models. They can be used to solve the segmentation problem[5], which typically occurs with continuous temporal data.

[4]`http://homepages.gold.ac.uk/nikolaev/cis311.htm` (last visit 13.08.2009)

[5]When working with a continuous stream of signal data, it can not be known a priori when a gesture begins or ends. The segmentation problem is concerned with the problem of identifying the relevant segments of the data, which are suitable to be classified.

However, the latter property of hidden Markov models, facilitating auto-segmentation, could not be exploited within the scope of this project.

Several other studies have been using hidden Markov models for gesture recognition and they have achieved good recognition results. Kela et al. (2006) state a recognition rate of up to 98.8 percent, Schlömer et al. (2008) reached between 85 to 95 percent and Prekopcsák (2008) achieved an average of 97.6 percent correctly classified gestures.

For their particular properties and the promising results, achieved in comparable studies utilising them, discrete hidden Markov models were chosen to form the basis of the pattern recognition process.

2.3 Recognition Pipeline

Hidden Markov models represent the core of the gesture recognition application, that is developed in the course of this project. To be applied in a practical application, some pre- and post-processing steps need to be applied, in order to bring the input data into a form, which can be easily processed by discrete hidden Markov models. The optimisation achieved by this additional processing steps also helps minimise computational and memory demands. This is of extra importance, since the final application should be able to run on a mobile device, where both of these resources are limited.

Based on the study of related papers (Kela et al., 2006; Schlömer et al., 2008), further literature on hidden Markov models (Fink, 2007) and experiences made during the implementation phase of this project, the following recognition and training pipeline was developed (see figure 2.1).

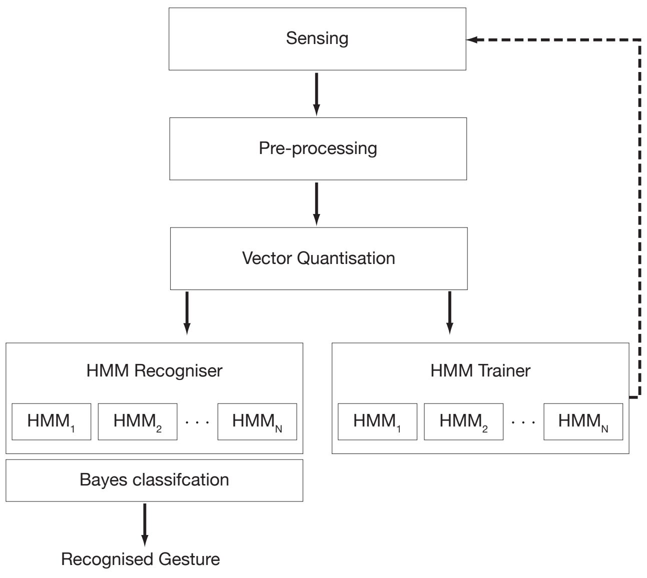


Figure 2.1: Diagram of Training and Recognition Pipeline

Sensing
Acceleration of the device (relative to free-fall) is constantly measured by the iPhone's built-in accelerometer. The acceleration is measured along three axis, resulting in a three-dimensional data signal.

Data Pre-Processing
The sensitivity of the accelerometer is relatively high. Different pre-processing filters are tested, in order to reduce the input data, without losing too much of relevant information (see section 4.3 for tested methods and filter values). These pre-processing steps are in theory not strictly necessary, their aim is to improve the performance of the application, which in turn accounts for the desired real-time functionality.

Vector Quantisation
Vector quantisation is a process of mapping a large input vector space onto a smaller subset of so-called prototype vectors. This can also be described in terms of cluster analysis: The original input data vectors are divided into a fixed number of clusters. For each cluster, there is one vector, which is most representative for its cluster, the so-called prototype vector. The vector quantisation process 'converts' the large number three-dimensional acceleration vectors into a manageable amount of prototype vectors. Now each of the prototype vectors, a symbol is assigned, resulting in a discrete set of symbols. In this case, simply the integer indices from the list of prototype vectors serve as the observation symbols. This pre-processing step is needed in order to generate a manageable amount of observation symbols, that can be processed efficiently by discrete hidden Markov models.

HMM Trainer
The hidden Markov model training module is responsible for creating a hidden Markov model (see section 3.2) for each gesture, that should be in the recognition pool. This gesture model is then trained, or optimised, based on multiple training sequences. Training sequences are created by recording several repetitions of the desired hand gesture (data pre-processing and vector quantisation steps are applied too).

HMM Recogniser
The HMM recognition module is responsible for assessing how well the input data, obtained from a gesture just performed, matches the available gesture models. The pool of available gesture HMMs is established through the HMM trainer module. Speaking in HMM terms, the HMM recogniser calculates the probability that the observation sequence was generated by a given hidden markov model. For each of the previously learned gesture models, this probability distribution is computed.

Bayes Classification
Unfortunately, the probability distributions obtained from the HMM recogniser can not be directly used for the final classification of the input data. This is because the maximum probabilities of each of the gesture models are extremely diverging. By applying Bayes' rule (see section 3.3) in combination with the average probabilities of the training sequences, normalised probability values are generated. Finally, the gesture HMM with the highest probability after Bayes' rule, is returned as the recognised gesture.

Chapter 3

Theory

3.1 Vector Quantisation

The purpose of a so-called vector quantiser is to map vectors from a large input data space onto a finite set of typical reproduction vectors. These reproduction or prototype vectors are a small subset of the original vector space. During this mapping, ideally no information should be lost, but at the same time the complexity of the data representation should be significantly reduced.

When processing signal data with a digital computer, these digital representations are already always finite. However, this data space is still very large, and it needs to be broken down further, before it can be processed with discrete Hidden Markov Models in an efficient manner. This is especially important, since the data is processed on a mobile device, who's processor is vastly slower than the one typically found in today's desktop computers.

The reduction in complexity is achieved by grouping similar vectors together. Each input vector is mapped onto a corresponding prototype vector of a given codebook. This mapping is done according to the nearest-neighbour condition. From this point of view, a vector quantiser can also be seen as a method of cluster analysis.

There are two problems to be solved regarding vector quantisers. Firstly, the number of prototype vectors needs to be decided upon. This number is an essential parameter for characterising a vector quantiser, and it has major influence on its quality. The second problem concerns the selection of prototype vectors from the input data space. The set of prototype vectors is also referred to as a codebook, and the number of prototype vectors is the codebook size.

The first problem about the codebook size needs to be handled very much application dependent. There is no general algorithm to determine this number. Depending on the sample data, this is a trade-off between minimising the quantisation error (larger codebook size) and maximising the reduction in complexity (smaller codebook size). For solving the second problem of finding appropriate prototype vectors, several algorithms exist. None of them can find an optimal solution for a given set of sample data and a given codebook size. For this project the so-called Lloyd's algorithm was used. It is described later in this chapter.

The following formal definition of vector quantisers and the description of the Lloyd's training algorithm is freely adapted from (Fink, 2007, chapter 4) while the mathematical notation has been retained.

3.1.1 Definition

A vector quantiser Q is defined as a mapping of a k-dimensional[1] vector space $\mathbb{R}^k$ onto a finite subset $Y \subset \mathbb{R}^k$

$$Q : \mathbb{R}^k \mapsto Y$$

Let the codebook be defined as the set $Y = y_1, y_2, ...y_N$. The prototype vectors y_i are sometimes also referred to as reproduction vectors or as code words. N denotes the size of the codebook.

As a vector quantiser Q of size N maps every vector x form the input space to exactly one prototype vector y_i, it defines a complete and disjoint partition of $\mathbb{R}^k$ into regions or cells $R_1, R_2, ...R_N$. In the cell R_i lie all those vectors $x \in \mathbb{R}^k$, which were mapped to the prototype vector or code word y_i.

$$\bigcup_{i=0}^{N} R_i = \mathbb{R}^k \qquad \text{and} \qquad R_i \cap R_j = \emptyset \quad \forall_{i,j} \text{ with } i \neq j$$

The so-called nearest-neighbour condition describes the optimal selection of the partition $\{R_i\}$ for a given codebook Y. Every partition cell R_i needs to be determined so that it contains all vectors $x \in \mathbb{R}^k$, which have minimal distance from the associated prototype vector y_i. This means that the corresponding vector quantiser Q maps a sample vector x onto its nearest prototype vector of the codebook.

$$Q(x) = y_i \qquad \text{if} \qquad d(x, y_i) \leq d(x, y_j)\ \forall j \neq i$$

In the above equation $d(.,.)$ denotes a general distance function. It can be substituted with a specific distance function. In this paper, the Euclidean distance[2] in three dimensional vector space was used:

$$d(x, y) = ||y - x|| = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2 + (y_3 - x_3)^2}$$

If the distance between vector x has minimal distance to more than only one prototype vectors, it can be mapped to any one of those prototype vectors. The resulting quantisation error is not affected by the actual choice (Fink, 2007, p. 48).

[1]In this application, the vector space is going to be three dimensional, because of the nature of the sample vectors, which are retrieved from the iPhone's 3-axis accelerometer

[2]The vector quantiser implemented in the ABG Library operates with the squared Euclidean distance in order to save square root operations for performance reasons.

The optimal choice of a codebook Y for a given partition R_i is defined by the centroid condition. For a cell R_i of the partition the optimal reproduction vector is given by that prototype vector y_i, that corresponds to the centroid of the cell:

$$y_i = cent(R_i)$$

The centroid of a cell R here is defined as the vector $y^* \in R$ from that all other vectors $x \in R$ in the statistical average have minimal distance.

$$y^* = cent(R) = \underset{y \in R}{\operatorname{argmin}} \, \varepsilon\{d(X,y)|X \in R\}$$

The random variable X is used here to represent the distribution of the data vectors x in the input space.

A sample vector $x \in \mathbb{R}^k$ that was mapped onto a prototype vector y will in general be different form the source vector. Therefore from every single quantisation operation results an individual quantisation error $\epsilon(x|Q)$. This error can be described by the distance between the original and the quantised vector:

$$\epsilon(x|Q) = d(x, Q(x))$$

when using Euclidean distance:

$$\epsilon(x|Q) = ||x - Q(x)||$$

Finally, we also introduce index $I = \{1, 2, ...N\}$ which denotes the index of codebook Y. Later in this paper, it will be shown how this index serves as the alphabet of possible observation symbols for discrete hidden Markov models.

3.1.2 Lloyd's Algorithm

In the previous section, the elements of a vector quantiser and optimality conditions, such as the nearest-neighbour rule were introduced. However, what was previously referred to as the second problem vector quantiser, has not been answered yet: Given a sample set of input vectors $\omega = \{x_1, x_2 \dots x_T\}$ and the codebook size N – how can N optimal prototype vectors be determined, which represent the codebook Y?

There are various algorithms that can find a near-optimal solution to this problem. The one, that is used in this project, is called Lloyd's algorithm, sometimes also called the Voronoi iteration. According to Fink (2007), in the literature, this algorithm is often referred to by mistake as k-means algorithm.

The Lloyds algorithm is an iterative algorithm, achieving a better solution at each iteration. At the beginning of the procedure, N suitable prototype vectors are chosen. This initial codebook is referred to as Y^0. The choice of the initial codebook influences the quality of the vector quantiser. It needs, however, be done heuristically. A common approach is to randomly select N vectors from the input vectors. In the corresponding implementation section on vector quantisation (section 4.4), other methods are evaluated as well.

The next step is to map every vector x_t from the sample set to the nearest prototype vector y_i^m of the current codebook Y^m. This mapping also determines the partitions R_i^m. From the newly formed partitions, an updated codebook can be computed. The centroids of the new partitions serve as the prototype vectors for the updated codebook. Since Euclidean distance is used for the centroid condition. The centroid for a given partition can simply be calculated as the mean vector of that partition.

The algorithm terminates, when the relative decrease of the quantisation error is less than the specified lower bound $\Delta_{\epsilon_{min}}$.

The following summary of the Lloyd's algorithm has been adapted from from Fink (2007).

Given

- sample set of input vectors $\omega = \{x_1, x_2 \ldots x_T\}$
- desired codebook size N
- lower bound $\Delta_{\epsilon_{min}}$ for the relative improvement of the quantisation error

1. Initialisation

- Choose a suitable initial codebook Y^0 of size N
- set iteration count $m \leftarrow 0$

2. Partitioning

for the current codebook Y^m, determine the optimal partition by classifying all vectors x_t with $t = 1 \ldots T$ into cells
$R_i^m = \{x | y_i^m = \underset{y \in Y^m}{\operatorname{argmin}}\, d(x, y)\}$

also compute the average quantisation error
$\bar{\epsilon}(Y^m) = \frac{1}{T} \sum_{t=1}^{T} \min_{y \in Y^m} d(x_t, y)$

3. Codebook Update

for all cells R_i^m with $i = 1 \ldots N$ calculate new reproduction vectors
$y_i^{m+1} = \text{cent}(R_i^m)$
these constitute the new codebook $Y^{m+1} = \{y_i^{m+1} | 1 \leq i \leq N\}$

4. Termination

calculate the relative decrease of the quantisation error with respect to the last iteration
$\Delta_{\epsilon_m} = \dfrac{\bar{\epsilon}(Y^{m-1} - \bar{\epsilon}(Y^m)}{\bar{\epsilon}(Y^m)}$
if the relative decrease was large enough: $\Delta_{\epsilon_m} > \Delta_{\epsilon_{min}}$
    set $m \leftarrow m + 1$ and continue with step 2.
otherwise stop.

3.2 Hidden Markov Models

The following description of Markov models is mainly based on the the classic paper by Rabiner (1989) but also on Fink (2007). The mathematical notation was used as proposed by Fink (2007), but it largely corresponds to the one found in Rabiner (1989).

Hidden Markov models can be divided into two main types: discrete hidden Markov models and continuos hidden Markov models. There are also hybrid types, such as semi-continuous hidden Markov models. Discrete Markov models operate with probability distributions of symbols form a finite alphabet of symbols. Continuous hidden Markov models operate with probability densities on continuous data, represented as mixture models. In this paper, only the discrete models are examined. Therefore, the following theory section, including mathematical formulas, only applies for discrete hidden Markov models.

3.2.1 Markov Chains

Before diving into full blown hidden Markov models, at this point, so-called Markov chain models shall be presented first. As it will be shown later, hidden Markov models are fundamentally based on Markov chain models. Markov chains can be used for the statistical description of symbol and state sequences.

A discrete Markov chain model can be visualised as a finite state automaton with connections between all states (see figure 3.1). The finite set of N distinct states of the system is defined as:

$$S = \{1, 2, ..., N\}$$

At discrete time steps, the system undergoes a change of state. The change of state happens according to the set of state transition probabilities, these can be thought of as the connections between the state nodes. Because there may be connections pointing back to the same state (a_{11}, a_{22}, a_{33} in figure 3.1 represent such connections), it is possible that the system ends up in the same state. The time instants associated with the state changes are denoted as $t = 1, 2, ..., T$ (for a finite time interval of length T). The actual state of the system at time t can be denoted as S_t.

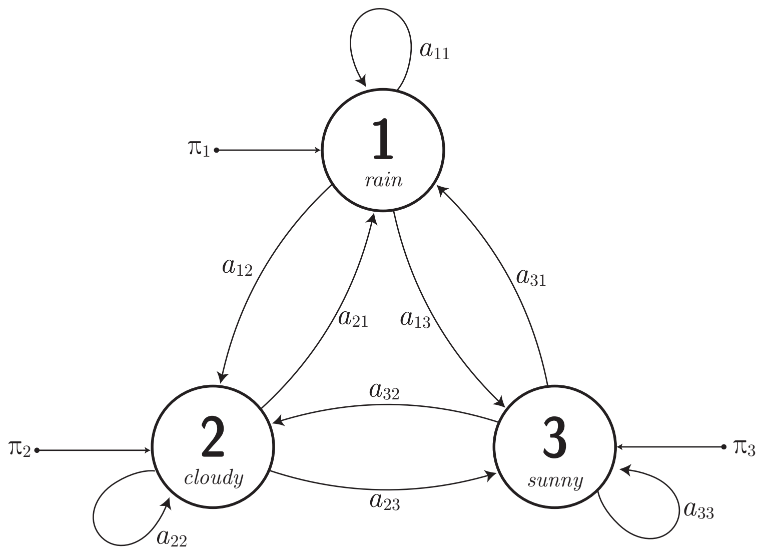


Figure 3.1: A Markov chain with 3 states

The behaviour of the system at a given time t is only dependent on the previous state in time. This is the so-called Markov property and therefore the probabilistic description of the system's states can be characterised as follows:

$$P(S_t|S_1, S_2, ...S_{t-1}) = P(S_t|S_{t-1})$$

The set of state transition probabilities is denoted as the matrix A:

$$A = \{a_{ij}\}$$
$$a_{ij} = P(S_t = j|S_{t-1} = i) \qquad 1 \leq i, j \leq N$$

It has not yet been defined how the initial state of the system is determined. Therefore the vector π is introduced. It holds the set of start probabilities for each state:

$$\pi = \{\pi_i|\pi_i = P(S_1 = i)\}$$

The state transition matrix A and the start probability vector π_i obey the following standard stochastic constraints:

$$\sum_{j=1}^{N} a_{ij} = 1$$
$$\sum_{i=1}^{N} \pi_i = 1$$

Expressed verbosely, this means that the state probabilities for each state in the model (each row of the matrix) must sum up to 1. And the same applies for the start probabilities.

Now that all fundamental elements of a Markov chain have been introduced, the concrete example of a 3-state system is presented. This graphic example was adapted from Rabiner (1989).

The example of a 3-state Markov model of the weather shall be considered (see figure 3.1 on page 13) Therefore, it is assumed that once a day (e.g., at noon), the weather is observed as being one of the following:

State 1: rain

State 2: cloudy

State 3: sunny

The weather on day t can only be a single one of the three states above. The matrix A of the state transition probabilities is given as:

$$A = \{a_{ij}\} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 0.4 & 0.3 & 0.3 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{bmatrix}$$

The current weather shall be sunny (state 3), therefore the start vector π of the model looks as follows:

$$\pi = \{\pi_i\} = \begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \\ 1.0 \end{bmatrix}$$

Now the question is asked: What is the probability (according to the given model) that the weather for the next 7 days will be "sunny–sunny–rainy–rainy–sunny–cloudy–sunny"? To be able to pose this question more formally, the observation sequence O is definded:

$$\begin{aligned} O &= \{\text{sun}, \text{sun}, \text{rain}, \text{rain}, \text{sun}, \text{clouds}, \text{sun}\} \\ &= \{S_1 = 3, S_2 = 3, S_3 = 1, S_4 = 1, S_5 = 3, S_6 = 2, S_7 = 3\} \\ &= \{3, 3, 1, 1, 3, 2, 3\} \end{aligned}$$

Note that the symbols $\{1, 2, 3\}$ refer to the states of the 3-state Markov chain.

The question about the probability can be expressed and evaluated for the given model and observation sequence as:

$$\begin{aligned} P(O|Model) =& P(3, 3, 1, 1, 3, 2, 3|Model) \\ =& P(3) \cdot P(3|3) \cdot P(3|3) \cdot P(1|3) \cdot P(1|1) \cdot P(3|1) \cdot P(2|3) \cdot P(3|2) \\ =& \pi_3 \cdot a_{33} \cdot a_{33} \cdot a_{31} \cdot a_{11} \cdot a_{13} \cdot a_{32} \cdot a_{23} \\ =& 1.0 \cdot 0.8 \cdot 0.8 \cdot 0.1 \cdot 0.4 \cdot 0.3 \cdot 0.1 \cdot 0.2 \\ =& \mathbf{1.536 \times 10^{-4}} \end{aligned}$$

There are more questions that can be answered by utilising Markov chain models, please see Rabiner (1989) for further details.

3.2.2 Hidden Markov Model Definition

Hidden Markov models describe a two-stage stochastic process. The first stage is exactly a Markov chain as defined in the previous section. Therefore hidden Markov models can be seen as extended versions of Markov chain models.

In an HMM, a Markov chain represents the so-called hidden part of the model. It is called hidden, because in an HMM, the individual states do not directly correspond to some observable event and are therefore hidden. Instead, the observable event is itself another probabilistic function. It is a probabilistic function that depends (and only depends) on the current state. This probabilistic function represents the second stage of the HMM process.

In the example 3-state weather model discussed in the previous section, each of the three states corresponds directly to an observable meaningful event, namely to a rainy, cloudy or sunny day. In contrast, the states of an HMM do not have any obvious meaning[3], they are primarily just states $S = \{1, 2, \dots N\}$ in which the model can be in. From this point of view, the properties of the hidden states of an HMM are similar to those of a multilayer perceptron's hidden nodes[4].

A hidden state does not correspond to an observation event directly, instead, each hidden state maintains an associated probability distribution for each possible observation symbol of the observation alphabet. These probability distributions are relevant for the second stage of the HMM process, and are also called observation likelihoods or emission probabilities.

The set of emission probabilities of an HMM can be grouped together in matrix B:

$$\mathbf{B} = \{b_{jk}\}$$
$$b_{jk} = P(O_t = o_k | S_t = j)$$

O_t defines the observation symbols (or emissions) at time t. The observations belong to the discrete alphabet V of size K:

$$V = \{o_1, o_2, \dots o_K\}$$

In the context of this paper, the codebook index I, which is defined in section 3.1.1 on vector quantisation, serves as the observation alphabet.

The example model of the weather Markov chain, which was defined in the previous section, shall be considered again. This example model shall now be extended into a full blown hidden Markov model in order to further illustrate the additional elements that characterise a hidden Markov model. Therefore, to each of the three states (these are now the hidden states), the complete alphabet of possible observation

[3]This shall not mean that the hidden states of an HMM do not have any meaning at all. In fact, the inspection of the model states can be exploited in the context of data segmentation. And of course they also account for producing the system's output.

[4]A multilayer perceptron is a type of artificial neural networks, for more on this topic, please see class notes by Dr. Nikolay Nikolaev: `http://homepages.gold.ac.uk/nikolaev/cis311.htm` (last visit 13.08.2009) or Haykin (1999).

symbols is assigned. The probability relationship between the state and an individual emission symbol is described by the probability distribution b_{jk}. The observation alphabet is typically larger than the number of hidden states in the model. Therefore the existing alphabet {rain, clouds, sun} is extended to {rain, clouds, sun, fog, snow}. A visualisation of this example weather HMM can be seen in figure 3.2.

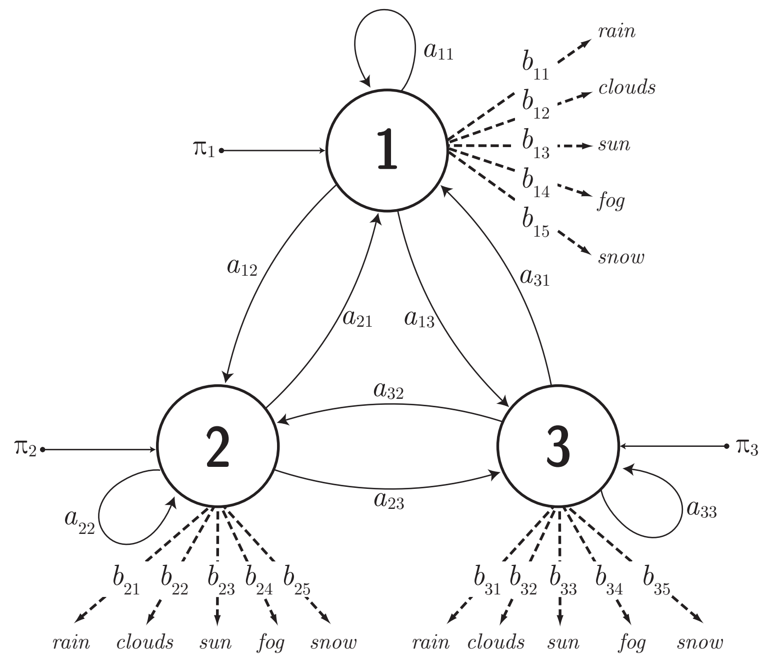


Figure 3.2: HMM with 3 hidden states and observation alphabet {rain,clouds,sun,fog,snow}

Finally, the formal definition of a hidden Markov model is summarised, including the parts that are overlapping with Markov chain models:

A hidden Markov model can be formally denoted as $\lambda = (S, A, B, \pi, V)$ but usually in literature, e.g. in Rabiner (1989) the short form $\lambda = (A, B, \pi)$ is used. The individual components are:

$S = \{1, 2, \dots N\}$	the set of N hidden states
$A = \{a_{ij}\}$	the state transition probability matrix, each a_{ij} representing the probability that the system changes from state i to state j
$B = \{b_{jk}\}$	the emission probability matrix, each b_{jk} representing the probability that state j generates observation o_k
$\pi = \{\pi_i\}$	the vector of initial state probabilities, each π_i representing the probability that the system starts in state i

$V = \{o_1, o_2, \dots o_K\}$	the alphabet of K observation symbols

A given HMM λ can be used as a generator for an observation sequence O of length T:

$$O = O_1, O_2, \dots O_T$$

3.2.3 Three Basic Problems for HMMs

In order for HMMs to be of any practical use, Rabiner (1989) poses three fundamental problems that need to be solved.

Problem 1: Evaluation

> "Given the observation sequence $O = O_1, O_2, \dots O_T$, and a model $\lambda = (A, B, \pi)$, how do we efficiently compute $P(O|\lambda)$, the probability of the observation sequence, given the model?" (Rabiner, 1989)

To be able to compute the probability that an observed sequence was produced by a given model, is extremely relevant for the gesture recognition application presented in this paper. This problem can also be seen as one of scoring: How well does a given observation sequence match a given model? Therefore, the solution to this problem lies at the heart of the gesture recognition process.

This problem corresponds to the question covered in section 3.2.1 about the likelihood of a weather forecast to be generated by the simple 3-state Markov chain. As a consequence, an obvious solution to this problem is to extend the existing method for hidden Markov models. The major difference here is that the state sequence an HMM runs through, in order to produce a given observation sequence, can not be known in advance.

Because the state sequence can not be known, every hidden state (and its emission probabilities) needs to be considered at every time time t. For an HMM with N hidden states and an observation sequence of length T, there are N^T possible hidden sequences. The resulting algorithm is exponential in the length T of the observation sequence. Therefore, this method is out of question when it comes to practical applications. Luckily, as it will be shown in section 3.2.4, there is a more efficient solution to this problem, namely the so-called forward algorithm.

Problem 2: Decoding

> "Given the observation sequence $O = O_1, O_2, \dots O_T$, and a model λ, how do we choose a corresponding state sequence $Q = q_1, q_2, \dots q_T$ which is optimal in some meaningful sense (i.e. best 'explains' the observations)?" (Rabiner, 1989)

This problem is not of immediate relevance for the presented gesture recognition task and was therefore not further examined. For the sake of completion it is mentioned here that a common solution to this problem is provided by the Viterbi algorithm, e.g. see Fink (2007).

Problem 3: Training

> "How do we adjust the model parameters $\lambda = (A, B, \pi)$ to maximize $P(O|\lambda)$?" (Rabiner, 1989)

This problem refers to the training of a model. The model's parameters are to be optimised in order to increase its probability for the generation of a given observation sequence. Such an observation sequence is called training sequence.

The ability to train HMMs is crucial for the gesture recognition application. The adaptation of the models parameter's to an observed training sequence, allows the creation of a model that best describes a certain gesture.

Several training algorithms for HMMs have been developed. The most widely used method, according to Fink (2007), is the Baum-Welch algorithm 3.2.4.

3.2.4 Forward Algorithm

The forward algorithm provides a solution to calculate the likelihood that a particular observation sequence O was generated by a given hidden Markov model. Formally this can be written as $P(O|\lambda)$.

In contrast to the 'brute fore' method, that has been mentioned under problem 1, the forward algorithm exploits the Markov property (the probability of a particular state dependent only on the previous state, also see section 3.2.1) in order to reduce the number of computations required significantly.

First, an auxiliary set of variables the so-called forward variables $\alpha_t(i)$ is introduced. It defines, for a given model λ, the probability that the first part of an observation sequence up to O_t is generated and that at time t the system is in state i.

$$\alpha_t(i) = P(O_1, O_2, \ldots O_t, s_t = i|\lambda) \tag{3.1}$$

1 Initialisation

The algorithm is initialised by obtaining the value for α at time $t = 1$.

$\alpha_1(i) = \pi_i b_i(O_1) \qquad 1 \leq i \leq N$

2 Recursion

for all times t, $t = 1 \dots T-1$:

$$\alpha_{t+1}(j) = \sum_{i=1}^{N} \Big[\alpha_t(i) a_{ij}\Big] b_j(O_{t+1}) \qquad 1 \leq j \leq N \tag{3.2}$$

3 Termination

Finally, all N probabilities α at time T are summed up.

$$P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

For the implementation in actual C/Objective-C program code, please see file `ABGHMM.h`, methods `forwardProc` and `getProbability`.

3.2.5 Backward Algorithm

The backward algorithm by itself does not solve a new problem. Like the forward algorithm, it also calculates the probability $P(O|\lambda)$. But the backward algorithm starts at time $t = T$, the end of the observation sequence, working backwards – while the forward algorithm starts with time $t = 1$. Roughly speaking it can be seen as the inverse of the forward algorithm. The backward algorithm is presented at this point, because it is used within the Baum-Welch training algorithm (see next section).

Analogous to the forward variable $\alpha_t(i)$, a the backward variable $\beta_t(i)$ is defined:

$$\beta_t(i) = P(O_{t+1}, O_{t+1}, \dots O_t, s_t = i|\lambda) \tag{3.3}$$

1 Initialisation

$\beta_T(i) = 1$

2 Recursion

for all times t, $t = T-1 \dots 1$:

$$\beta_t(j) = \sum_{j=1}^{N} \alpha_{ij} b_j(O_{t+1}) \beta_{t+1}(j) \qquad 1 \leq j \leq N$$

3 Termination

$$P(O|\lambda) = \sum_{i=1}^{N} \pi_i b_i(O_1) \beta_1(i)$$

For the implementation in actual C/Objective-C program code, please see file `ABGHMM.h`, methods `backwardProc` and `getProbabilityBackwards`.

3.2.6 Baum-Welch Algorithm

The Baum-Welch algorithm, sometimes also called forward-backward algorithm, is an iterative training method for hidden Markov Models. Training here means maximising the likelihood that a model λ emits observation sequence O. The Baum-Welch algorithm represents a variant of the expectation-maximisation (EM) algorithm, which in general optimises parameters of multi-stage stochastic processes (c.f. Dempster et al., 1977).

At each iteration, the algorithm produces new estimates for the model parameters, the new model and parameters are denoted as $\hat{\lambda} = (\hat{A}, \hat{B}.\hat{\pi})$.

In order to describe the algorithm, the following auxiliary variables are defined. The first variable $\gamma_t(i)$ represents the probability of the system being in state i at time t. It can be calculated as follows, by drawing on the forward and backward variables defined in the previous sections:

$$\gamma_t(i) = P(S_t = i|O, \lambda) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)} \tag{3.4}$$

The second variable defines the probability of a transition from state i to state j at time t, it is denoted as $\gamma_t(i,j)$. This probability can be calculated, based on the forward and backward variables, as follows:

$$\gamma_t(i,j) = P(S_t = i, S_{t+1} = j|O, \lambda) = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)} \tag{3.5}$$

By using the above formulas and the concept of counting event occurrences, the re-estimates of the parameters of an HMM can be written as:

$$\hat{a}_{ij} = \frac{\text{expected number of transitions form state } i \text{ to state } j}{\text{expected number of transitions from state } i}$$

$$= \frac{\sum_{t=1}^{T-1} \gamma_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$\hat{b}_j(o_k) = \frac{\text{expected number of times in state } j \text{ and observing symbol } o_k}{\text{expected number of times in state } i}$$

$$= \frac{\sum_{t:O_t=o_k} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

The start probabilities π_i can be seen as a special case of the transition probabilities a_{ij}. The following equation determines the improved start probabilities:

$$\hat{\pi}_i = \gamma_1(i)$$

Now that it has been described how an improved set of model parameters can be calculated, an overview of the overall algorithm is given:

1 Initialisation
Choice of suitable initial model $\lambda = (A, B, \pi)$ with initial estimates for

- π_i start probabilities
- a_{ij} transition probabilities
- b_{jk} emission probabilities

2 Optimisation

Compute updated estimates $\hat{\lambda} = (\hat{A}, \hat{B}, \hat{\pi})$
according to the re-estimation formulas presented above.

3 Termination

If the quality measure $P(O|\hat{\lambda})$ was considerably improved by the updated model $\hat{\lambda}$ with respect to λ
    let $\lambda \leftarrow \hat{\lambda}$ and continue with step 2
otherwise stop.

The Baum-Welch algorithm was at first implemented according the formulas presented above. During the implementation phase of this project, some extensions and amendments were made. The algorithm was extended to work with multiple observation sequences (see section 4.5.2). Thereafter, all probability calculations were moved to the negative-logarithmic scale (see section 4.5.3), because of problems with numerical underflow. For this reasons, the current implementation, found in source file `ABGHMM.m`, does not any more directly correspond the formulas in this section.

For more detailed description and mathematical proofs for the three HMM algorithms presented in this and the two preceding sections, the reader is referred to the literature on HMM, e.g., see Fink (2007); Jurafsky and Martin (2008); Rabiner (1989).

3.3 Classification with Bayes

Bayes rule provides a solution for computing the so-called posterior event B, $P(B|A)$, based on prior knowledge about the random events $P(A)$ and $P(B)$ and the posterior probability of event A, $P(A|B)$.

It can therefore virtually reverse the conditional dependence of the events A and B. Bayes' rule in its general form is given by:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

Given is the gesture pool of N hidden markov models $\lambda_1, \lambda_2, \ldots, \lambda_N$, each model λ_j representing a gesture available for the recognition process. Now, the question is, which model λ_j is most likely to generate the sample observation sequence O. This question can be more formally posed as:

$$P(\lambda_j|O)$$

One problem with hidden Markov models is that $P(O|\lambda_j)$, the emission probability of observations O given the models λ_j, yields probability values that are varying in scale (compare the values for $P(O|\lambda)$ in appendix A.3.2), depending on the actual gesture model. Therefore, these probabilities can not be directly compared with each other to find out, which of the N HMMs actually generated the observation O most likely.

By taking into account a models average probability of its training sequences and denoting this as the models probability: $P(\lambda_j)$, the posterior probability $P(\lambda_j|O)$ can be computed by applying Bayes' rule as seen in Fink (2007):

$$P(\lambda_j|O) = \frac{P(O|\lambda_j)P(\lambda_j)}{\sum\limits_{i=1}^{N} P(O, \lambda_i)} = \frac{P(O|\lambda_j)P(\lambda_j)}{\sum\limits_{i=1}^{N} P(O|\lambda_i)P(\lambda_j)}$$

The model λ_j with the biggest value for $P(\lambda_j|O)$, represents the classified gesture. A nice side effect of the Bayes' rule is, that the computed values for $P(\lambda_j|O)$ add up to 1.

Chapter 4

Implementation

4.1 Test Gestures

The theory chapter (chapter 3) provides a general description of the relevant methods and software algorithms, which are applied in this project. Some of these methods require the definition of application specific constants or initial values. For example the codebook size and initial prototype vectors for the vector quantiser need to be chosen. For some of these parameters, an optimal value can only be determined empirically and heuristically. In order to run experiments, which yield comparable results, five simple hand gestures are defined. Based on the results for these test gestures, optimal values for the according constants and model parameters are sought. The five test gestures are: Circle, Square, Triangle, Z and Bowling. They are depicted in figure 4.1. Gestures 1-4 are all drawn in a plane in front of the user. The bowling gesture refers to throwing a bowling ball (but without actually throwing the device).

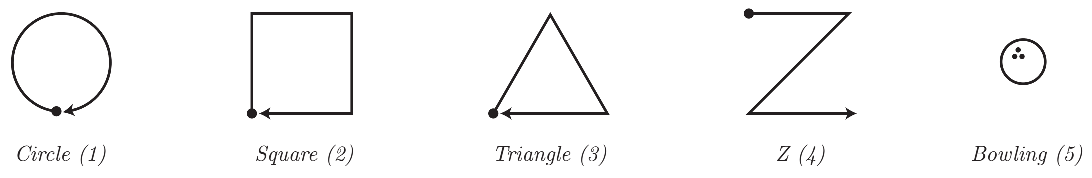


Figure 4.1: Test Gestures.

4.2 Device and Sensing

The Apple iPhone is not the only mobile phone, which has a built-in accelerometer. This device is chosen, because an iPhone 3G is available for this project and because there is not yet a software library for accelerometer based hand gesture recognition for the iPhone[1]. Further reasons that qualify the device for this project are: it has relatively good processing power, which should allow gesture-recognition close to real-time; it can be programmed in a comfortable way, through the software development kit (SDK) provided by Apple; basic knowledge of C and Objective-C, the programming languages used with the iPhone, had been acquired prior to this project.

For this project and the actual software implementation, an iPhone 3G is used. In this paper iPhone serves as a general term and does not mean a specific iPhone model[2]. If not specifically remarked otherwise, all findings of this paper should apply equally to all iPhone models. Although, this could not be empirically verified within the scope of this paper.

4.2.1 Accelerometer

The iPhone's built-in accelerometer is a LIS302DL[3] micro electro-mechanical system (MEMS). This is 3-axis accelerometer with nominal measure range of ± 2g. Acceleration data can be obtained at a frequency of up to 100Hz. The alignment of the sensor's axes with the device casing can be seen in figure 4.2.

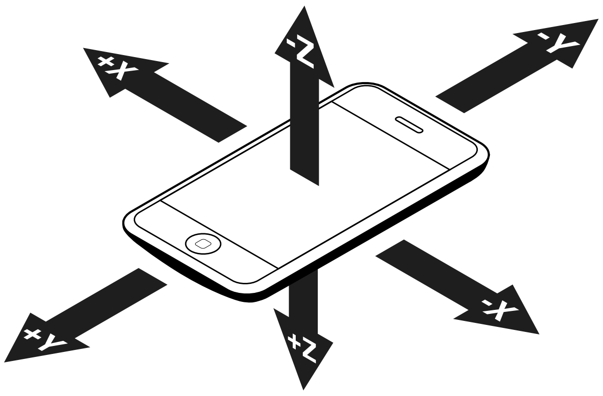


Figure 4.2: The iPhone's Accelerometer Axes.

An accelerometer is a device that measures the acceleration it experiences. On the Earth's surface this includes gravity. Therefore, an iPhone lying on a perfect horizontal table yields the acceleration values of $1g$, upwards the vertical axis, $\mathbf{a} = \{a_x = 0, a_y = 0, a_z = -1\}$. In free-fall or in outer space, the acceleration measured would be 0 along all axis: $\mathbf{a} = \{a_x = 0, a_y = 0, a_z = 0\}$. An accelerometer can not distinguish between gravitational acceleration and acceleration caused by movement, so-called inertial acceleration. The reason for this can be explained by Einstein's equivalence principal, which says that the effects of gravity on an object are indistinguishable form acceleration of the reference frame (Eshbach et al., 1990). When held fixed in a gravitational field, for example by placing it on a table (in order to

[1]At the time of writing, no iPhone library for accelerometer based hand gesture recognition, is known to exist.

[2]At the time of writing three different iPhone models have been released by Apple: (the original) iPhone, iPhone 3G and iPhone 3GS

[3]LIS302DL data sheet: `http://www.stm.com/stonline/products/literature/ds/12726.pdf` (last access 13.08.2009)

prevent it form free-fall), the reference frame for an accelerometer (its own casing) accelerates upwards with respect to a free-falling reference frame. The effect of this reference frame is indistinguishable from any other acceleration experienced by the instrument.

Therefore, there's always the influence of gravity along the vertical axis, but along the horizontal axes, it behaves more intuitively and measures the acceleration caused by movement of the device directly.

Although the SI[4] unit for acceleration is m/s^2, in engineering it is frequently quantified in g (also called g-force, although it is not a force but an acceleration). 1g, which is equal to standard gravity, is defined as $9.81m/s^2$.

4.2.2 Retrieving Sensor Data

The primary purpose of the iPhone's accelerometer is to determine the orientation of the device. Depending on how the device is held, the phone's operating system is able to notice the following four states, based on the accelerometer measurements: Portrait, Portrait Upside Down, Landscape Left, Landscape Right. Custom applications can register with the operating system to be notified whenever the device is tilted, in order to rotate the user interface accordingly. It should be clear that the data retrieved in this way, is not suitable as a bases for a gesture recognition system.

Fortunately, the iPhone API also provides a way to retrieve more low level accelerometer values. The update frequency at which the values are retrieved is configurable within the range of 10Hz – 100Hz. In this project an update frequency of 60Hz is used. This value proved to provide enough resolution for capturing gesture-like hand movements. Also, at this rate, the data can still be processed in real-time by the iPhone 3G. If the update frequency is much higher, some updates are dropped.

The following Objective-C code snippet[5] registers the current object for the retrieval of accelerometer values at a frequency of 60Hz:

Listing 4.1: Register for accelerometer updates

```
- (void)enableAccelerometerEvents {
    UIAccelerometer *acc = [UIAccelerometer sharedAccelerometer];
    acc.updateInterval = 1/60;  // 60 Hz
    acc.delegate = self;
}
```

At the specified frequency, the method `accelerometer:didAccelerate:` gets called and the accelerometer values of the three axis (also see figure 4.2) can be retrieved as shown below:

[4]The International System of Units (abbreviated SI from the French le système international d'unités)

[5]This paper does not go into details of the Objective-C and C programming languages, which were utilised in this project. For an short introduction to Objective-C, the following resource is recommended: `http://developer.apple.com/mac/library/referencelibrary/GettingStarted/Learning_Objective-C_A_Primer/index.html` (last access 17.08.2009)

Listing 4.2: Retrieving acceleration values

```
- (void)accelerometer:(UIAccelerometer*)accelerometer
        didAccelerate:(UIAcceleration*)acceleration {

    UIAccelerationValue x, y, z;

    x = acceleration.x;
    y = acceleration.y;
    z = acceleration.z;

    // Process the data...
}
```

The acceleration values are delivered already in quantities of g-force (see section 4.2.1). This makes some sort of manual calibration, as seen for the Wii controller accelerometers (Schlömer et al., 2008), obsolete. The values are floating point values with double precision (`UIAccelerationValue` in listing 4.2 is defined as a typedef of double).

The metering range of iPhone 3G's accelerometer, used for this project, was found to be ±2.3g along each axis, which is slightly more than the ±2.0g stated in the sensor's specification.

It needs to be considered, as with all real world sensors, there is always a certain amount of measuring inaccuracy. Furthermore, the factory-made g-force calibration might be equal across all produced sensors, due to differences in the manufacturing process. However, the sensor data should vary within a small range only and can be expected to be roughly the same on all devices. Also, the employed pattern recognition methods should balance any effects of sensor inaccuracies, although this has not been tested within the scope of this paper. Since the resulting software product of this project, the iPhone application ABGDemo, allows the training of custom free-from gestures directly on the device itself, these device specific variations are then not of relevance.

4.3 Data Pre-Processing

4.3.1 Threshold Filter

The first pre-processing step consists of removing similar vectors from the input data. As a measure for similarity, the Euclidean distance between the current and the previous vector is taken. In case the distance between the two vectors exceeds a certain threshold ϵ, it is considered in the prepossessed result set ω', otherwise it is omitted. For the input data set $\omega = \{x_1, x_2, \dots x_T\}$, this pre-processing filter can be denoted as:

$$\omega' = \big\{x \in \omega \,\big|\, \exists x : ||x_t - x_{t-1}|| > \epsilon\big\}$$

Considering the relatively high data rate of 60 Hz and the high sensitivity of the accelerometer, consecutive data vectors are almost identical when the data results from a natural hand movement. As a consequence, by removing certain vectors from the input data set, not too much information is lost. On the other hand the amount of input data can be significantly reduced, which speeds up the subsequent processing steps and keeps memory demands low.

The threshold value ϵ was empirically determined at $\epsilon = 0.055$ ($\epsilon^2 = 0.003$), which reduces the number of input samples by 14.9 percent on average (see table 4.1, also see appendix A.1.1 for the collected data). The final implementation in C uses the squared Euclidean distance, therefore the corresponding filter constant, in `ABGVectorUtils.h` is defined as:

```
#define kABGThresholdFilter 0.003
```

Table 4.1: Tested Threshold Filter Values.

Filter Value ϵ^2	Average Reduction of Input Vectors	Recognition Results Percent
0.01	43.3%	42%
0.005	27.5%	49%
0.003	**14.9%**	**62%**
0.0	0%	61%

From a theoretical perspective this pre-processing step is not strictly necessary, however, for the practical implementation, especially on a mobile device, it provides a valuable performance increase.

4.3.2 High-Pass Filter

The second pre-processing step, which was tested, is a high-pass filter. The high-pass filter was implemented and tried, but the results of the experiment in section 4.4.3 show that it does not improve the overall gesture recognition results. As a consequence of this, the filter is not used in the final implementation.

A high-pass filter attenuates low frequencies, but passes high frequencies. In the case of accelerometer data, this means that the acceleration caused by gravity is attenuated. Acceleration caused by sudden motion of the device, on the other hand, passes the filter. A high-pass filter operates in the frequency domain, but the accelerometer values are obtained as a function of time. Therefore, it would be necessary to apply a Fourier transform[6] function first. However, there is a simplified algorithm which approximates the function of a real high-pass filter well enough, without the need for a Fourier transform. The code was adapted from the class notes of the Stanford University iPhone course CS139P[7].

[6]e.g. see `http://en.wikipedia.org/wiki/Fourier_transform` (last visit 01.09.2009)

[7]CS193P http://www.stanford.edu/class/cs193p/cgi-bin/index.php (last visit 15.07.2009)

Listing 4.3: Retrieving acceleration values

```
#define kHighPassFilterFactor 0.1

ABGVector ABGHighPassFilter(ABGVector v) {
        static ABGVector lastV={0,0,0};

        lastV.x = (v.x * kHighPassFilterFactor) + (lastV.x * (1.0 -
            kHighPassFilterFactor));
        lastV.y = (v.y * kHighPassFilterFactor) + (lastV.y * (1.0 -
            kHighPassFilterFactor));
        lastV.z = (v.z * kHighPassFilterFactor) + (lastV.z * (1.0 -
            kHighPassFilterFactor));

        v.x = v.x - lastV.x;
        v.y = v.y - lastV.y;
        v.z = v.z - lastV.z;

        return v;
}
```

4.4 Vector Quantiser

The Lloyd's algorithm is the vector quantiser training algorithm, utilised in this project. It is described in section 3.1 of the theory chapter. The purpose of this algorithm is to find an optimised codebook for a given set of input vectors. The vector quantiser training algorithm is only of relevance during the training phase of a gesture. In order to train the vector quantiser with the Lloyd's algorithm, the vector data of all training sequences for one gesture is considered. This way, an optimised codebook for each individual gesture is created.

Once the gesture is learned, and available in the gesture pool, the Lloyd's training algorithm is no longer involved, because the codebook is not re-estimated for the recognition of a gesture. When in recognition mode, the vector quantiser serves in its basic mapping function. It maps every incoming data vector onto one prototype vector of the gesture's codebook, according to the nearest-neighbour rule (see section 3.1.1). Because each gesture has its own codebook, each input data sample goes through the vector quantisation process for as many times as there are trained gestures in the recognition pool.

The training algorithm has the following two prerequisites: Firstly, the number of prototype vectors, the so-called codebook size N, needs to be defined. Secondly, an initial set of N prototype vectors, the so-called initial codebook Y^0 needs to be chosen. In these sections various experiments are conducted in order to seek suitable values for both.

In the first series of experiments several methods for choosing the initial prototype vectors are tried. Thereafter, various codebook sizes are investigated. To gain indications for suitable codebook sizes, comparable studies are consulted. In Kela et al. (2006), a codebook size of 8 is used, but their study is confined to 2-dimensional gestures only. Schlömer et al. (2008) empirically determined in their study, 14 to be a good value for 3-dimensional data. Therefore, the codebook size of 14 serves as starting point for the first series of experiments.

Recognition results are based on left-to-right hidden Markov models with 8 states and step size 2 (see section 4.5.1 for details on HMM configurations and topologies). Each test gesture is trained based on the sample data from 6 training sequences. Training as well as recognition trials are conducted by one and the same person.

4.4.1 Random Codebook Initialisation

The canonical approach for choosing the initial codebook for the Lloyd's algorithm, is to select random vectors from the sample set of input vectors. This approach is tried, and 14 vectors are selected from the entire set of training vectors. The standard C random generator function `rand()` is employed to pick the prototype vectors.

In figure 4.3 the sample vectors and codebook of the circle gestures are visualised. Thin lines represent the sample vectors and the thicker lines represent the 14 prototype vectors, which were determined by vector quantiser training algorithm. Further visualisations of the remaining test gestures can be found in appendix A.2.1.

From looking at the visualisations, the random initialisation method resulted in a quite a good distribution of prototype vectors within the sample vectors. The mean squared quantisation error is 0.0443 and the average recognition rate is 42 percent (see appendix A.2.1).

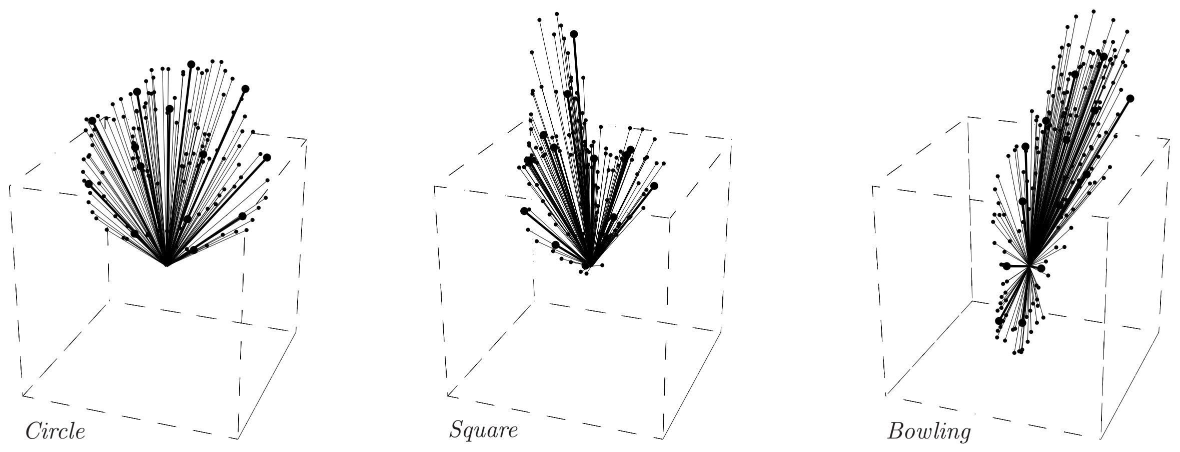


Figure 4.3: Visualisation of sample vectors (thin lines) and prototype vectors (bold lines). Random codebook initialisation with 14 prototype vectors.

4.4.2 Spherical Codebook Initialisation

In Schlömer et al. (2008), they describe a method for the choice of the initial codebook, which uses the points on a sphere as initial prototype vectors. The centre point of the sphere is located at the coordinate system's point of origin. The radius is dynamically determined: The mean length of all sample vectors is calculated and serves as the sphere's radius. On the surface of this sphere, 14 points equally distributed points are selected and used as the initial codebook. For this experiment, the 14 points are manually selected and hard coded, but for the further trials with different codebook sizes, the points are selected by an algorithm (see section 4.4.5).

This method is tried and the result for the mean squared quantisation error is 0.0759. An average recognition rate of 64 percent is achieved.

This method achieves better recognition results than the random initialisation. Although, visualisations of the sample vectors and the prototype vectors show various unused, freestanding prototype vectors (see figure 4.4, see appendix A.2.2 for visualisations of all test gestures).

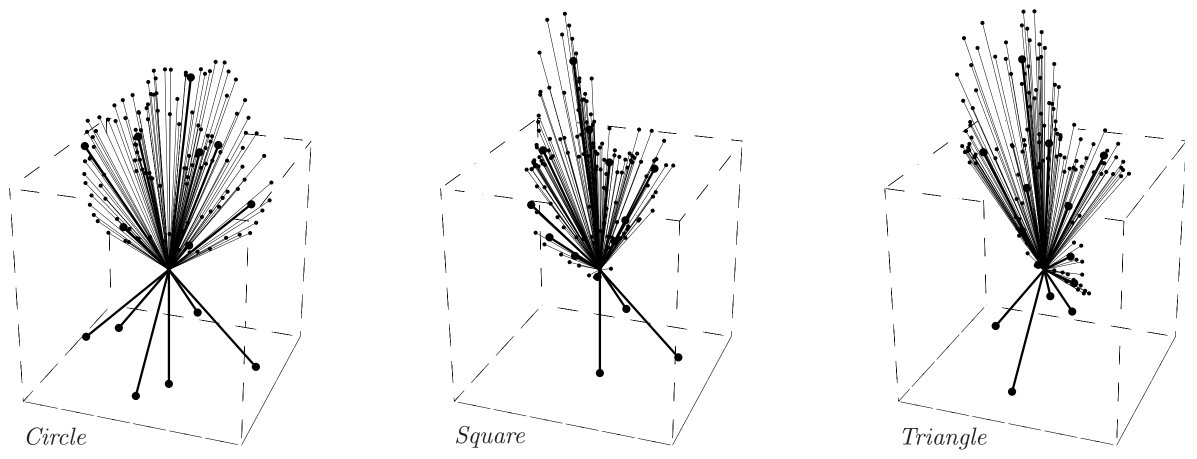


Figure 4.4: Visualisation of sample vectors (thin lines) and prototype vectors (bold lines). Spherical codebook initialisation with 14 prototype vectors.

4.4.3 Spherical Initialisation with High-Pass Filter

As remarked before, some prototype vectors in figure 4.4 stand out from the sample vectors and are clearly not involved in the quantisation process. It seems that the initial sphere is slightly off and does not coincide very well with the sample vectors. This is due to the influence of gravity on the accelerometer measurements (see section 4.2.1), which causes all sample vectors to appear to be shifted upwards.

In order to reduce the gravity component and to isolate out sudden movements, a high-pass filter was added as a pre-processing step (see section 4.3.2), before the vector quantiser. The aim of this pre-processing step is to better make the initial codebook fit the sample vectors, which in turn should reduce the quantisation error and therefore possibly lead to better recognition results.

With the high-pass filter hooked up in the preprocessing component, the spherical initialisation method is repeated as described in section 4.4.2.

Compared with 4.4.2, the quantisation error could indeed be reduced; the mean squared error is 0.0429. And as figure 4.5 shows, the prototype vectors are evenly distributed within the sample vectors. However, the recognition rate is with 44 percent (see appendix A.2.3) worse than in trial without the high-pass filter (section 4.4.2). See appendix A.2.3 for the full list of tables and visualisation figures.

It is therefore assumed that the high-pass filter makes the recognition process prone to differences in timing when drawing the gestures. When the high-pass filter is active, the input vectors tend towards the point of origin, whenever the device is not in motion. Therefore, depending on the speed at which a gesture is drawn, the resulting input vectors show differing patterns.

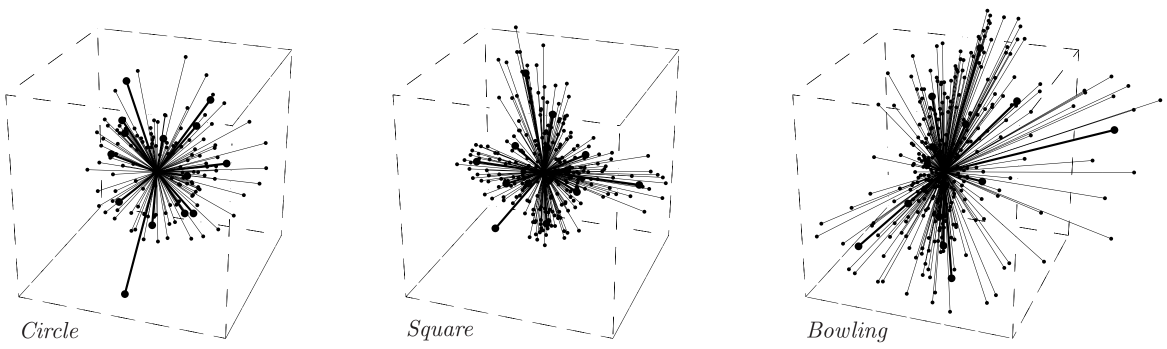


Figure 4.5: Visualisation of sample vectors (thin lines) and prototype vectors (bold lines). Spherical codebook initialisation with high-pass filter. 14 prototype vectors.

4.4.4 Spherical Initialisation with Offset

Because the high-pass filter does not yield improved results compared to the normal spherical initialisation, an other way is tried to further decrease the quantisation error and provide a better coincidence of the prototype vectors with the sample data. In this trial, the centre point of the codebook sphere is shifted to the centre point of the input vectors.

The quantisation error is at 0.0484 relatively small. Judging from the visualisation in figure 4.6, the prototype vectors are representing the sample data well. However, the recognition results at an average of 42 percent are not good enough. Therefore this initialisation method is not further prosecuted.

4.4.5 Testing Various Codebook Sizes

From the the tested methods for codebook initialisation in previous sections, the method that uses points on a sphere as described in section 4.4.2, provided the best results. Based on this method, further trials

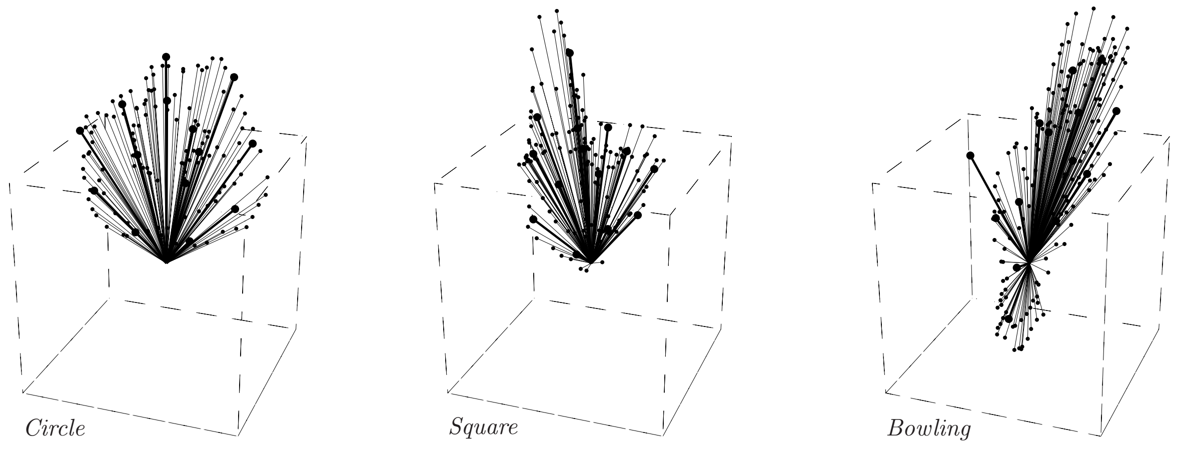


Figure 4.6: Visualisation of sample vectors (thin lines) and prototype vectors (bold lines). Spherical codebook initialisation with offset. 14 prototype vectors.

with varying codebook sizes are conducted in this section in order to investigate the effect of the codebook size on the recognition rate.

To be able to try spherical initialisation with various codebook sizes, an algorithm is sought, which can calculate N evenly distributed points on a sphere. It turns out that there is not a single definition for what *evenly distributed* means in this context. This is discussed in detail by Dave Rusin[8]. However, the definition relevant for this project can be described as: Given a number of points on a Sphere, the minimum distance between any two points should be as large as possible. All points should be equally distant from their closest neighbour.

The Platonic solids fulfil this criteria, but there are only five of them, therefore a more general solution is needed. One way to organise the points on the sphere is to simulate them repelling themselves with equal force, until their position settles. This would provide an optimal solution, but it is computationally expensive. Therefore, an approximation algorithm was implemented. The utilised algorithm is based on the work by Kuijlaars et al. (1998), which was further improved by the computer graphics community[9]. The pseudo code, which is presented below, is obtained from the respective entry[10] of the Computer Graphics Algorithms Wiki[11]. This pseudo code served as the basis for the implementation of the C-method `initCodeBookSphere()`, which can be found in file `ABGQuantizer.c` of ABGLib.

Listing 4.4: N evenly distributed points on a Sphere (Pseudo Code)

```
dlong := pi*(3-sqrt(5))
long := 0
dz := 2.0/N
z := 1 - dz/2
for k := 0 .. N-1
    r := sqrt(1-z*z)
```

[8]`http://www.math.niu.edu/~rusin/known-math/95/sphere.faq` (last visit 28.08.2009)

[9]Exact authorship is not known

[10]`http://cgafaq.info/wiki/Evenly_distributed_points_on_sphere` (last visit 01.09.2009)

[11]`http://cgafaq.info/wiki/` (last visit 01.09.2009)

```
pt[k] := (cos(long)*r, sin(long)*r, z)
z := z − dz
long := long + dlong
```

Equipped with this algorithm, spherical initialisation with the codebook sizes 8, 14, 20, 28 and 36 is tried. The results of these test series are summarised in table 4.2. Detailed results can be found in appendix A.2.5.

Table 4.2: Recognition Results and Quantisation Error for Codebook Sizes 8, 14, 20, 28, 36

Codebook Size	Quantisation MSE	Recognition Results		
		Recognised	Unrecognised	Misclassified
8	0.099056	62%	11%	27%
14	0.078692	61%	2%	37%
20	0.060433	66%	33%	1%
28	0.053147	61%	39%	0%
36	0.047476	56%	44%	0%

The gesture recognition results show that codebook size does not seem to have significant influence on the average recognition rate, which lies between 56% – 66% for all tried codebook sizes. However, the results demonstrate that a smaller codebook size causes more misclassifications. Misclassifications should be avoided because they can unlikely be eliminated by a higher number of training sequences. The number of unrecognised tries (see column unrecognised) are likely to be reduced with a higher number of training sequences. Based on this thoughts and the recognition results, the codebook size of 28 is chosen for the final implementation.

4.4.6 Vector Quantiser Post-Processing Step

The vector quantiser maps each input vector of the sample set onto a prototype vector from the codebook. Therefore, after the vector quantisation process, the length of the resulting list of prototype vector indices is equal to the length of the sample set. At this point a small post-processing step is added to the vector quantiser. Consecutive sample vectors are often mapped onto the same prototype vector. In an additional processing step, elements are removed from the resulting list, if they are equal to the preceding element. Therefore, in practice, the final list of prototype vector indices is shorter than the original list of input vectors. This step is applied, because shorter observation sequences speed up the HMM training and recognition process. Also, equal elements can be safely removed, as they do not contain further information about the gesture geometry. This processing step is recommend in Schlömer et al. (2008).

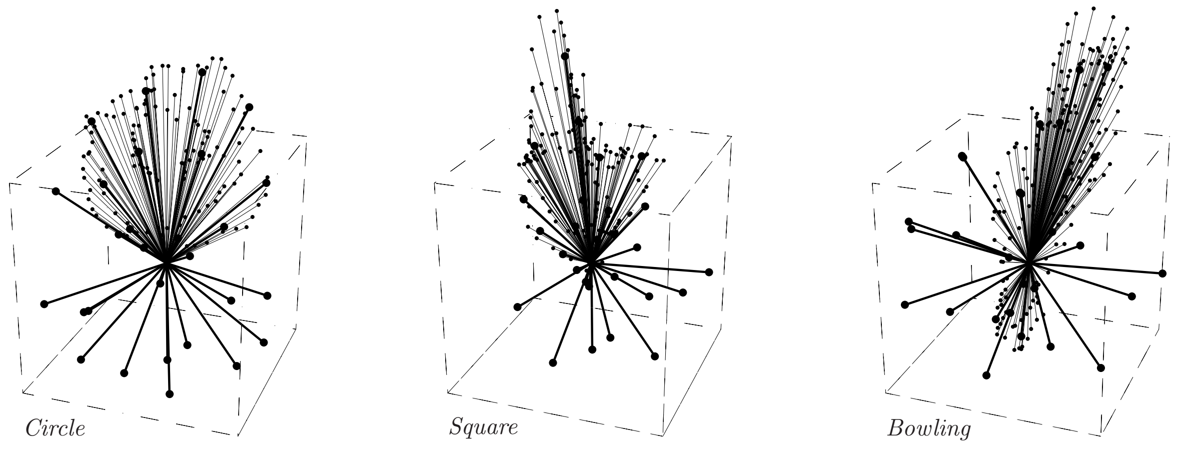


Figure 4.7: Visualisation of sample vectors (thin lines) and prototype vectors (bold lines). Spherical codebook initialisation with 28 prototype vectors.

4.5 Hidden Markov Models

Although various software libraries for hidden Markov models do already exist, the necessary HMM algorithms were implemented from scratch for this thesis project. Reasons for this are that most libraries are rather bloated and are therefore not ideal for use on a mobile device, where optimisation is important. Often, they contain extensions for specific applications, such as the processing of genome data for example. A second, and more important reason is that the standard HMM training algorithm can only take one training sequence to train the HMM. For a robust classification of gestures, hidden Markov models needs to be trained with multiple training examples as described in section 4.5.2. Although the changes in the HMM training algorithm for multiple training sequences are straight forward, when trying to modify an existing software implementation, these changes go very deep. Also, only libraries written in the programming languages C or Objective-C, and to a limited extent also C++, come into consideration, because only these languages go together with the iPhone SDK.

For these reasons, a fresh implementation is preferred. However, the code base of existing implementations has been consulted as a reference and had major influence on the present form of the HMM implementation in ABGLib. The consulted code libraries are the C library ghmm[12] and the wiigee[13] Java library.

To verify the Objective-C implementation, developed in this thesis project, example observation sequences are run against the wiigee Java library. The wiigee library also supports the training with multiple observation sequences but does not provide a solution to prevent numerical underflow (see section 4.5.3).

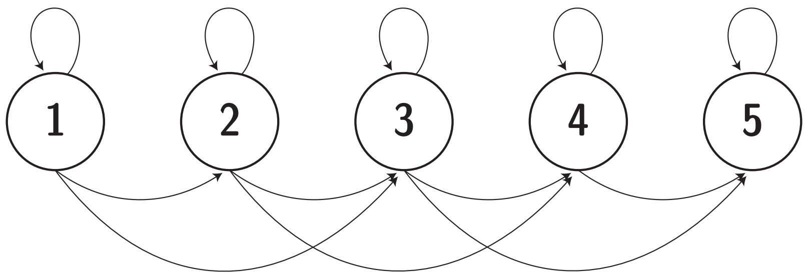


Figure 4.8: Left-to-right HMM with 5 states and step size 2

4.5.1 Configurations of Hidden Markov Models

In the theory section on hidden Markov Models (section 3.2), a fully-connected HMMs has been assumed (see figure 3.2). A fully-connected model, also called ergodic model, allows arbitrary state transitions between any states. Each state can be reached from any other state in one step. Therefore, all components of the state transition matrix A have to be greater than zero.

$$a_{ij} > 0, \quad \forall_{i,j}$$

For certain applications, such as the analysis of chronological sequences (and the analysis of gesture data is such a task), the literature recommends the use of so-called left-to-right HMMs (Jurafsky and Martin, 2008; Fink, 2007; Rabiner, 1989). A left-to-right model represents an HMM with a further constraint model topology. In contrast to ergodic models, which allow arbitrary state transitions, a left-to-right model only allows state changes from a lower-index state to a higher-index state. Jumping back to previous states is not allowed in a left-to-right model. Therefore, the state transition probability of 'backward-connections' are set to zero. In graphical representations such connections are excluded (see figure 4.8). Left-to-right models are often further constrained to make sure that large changes in state indices do not occur. A maximum step size Δ is introduced to limit state transitions to indices, which are at most Δ higher than the current state. The resulting constraints for the state transition probabilities of a left-to-right HMMs with step size Δ can be written as:

$$\begin{aligned} &a_{ij} = 0, \quad i > j \\ &\text{and} \\ &a_{ij} = 0, \quad j > (i + \Delta) \end{aligned}$$

To ensure that all states of a left-to-right HMM can actually be reached, the model must be in state 1 at the beginning. As a consequence of this, the inital probability for state 1 must be 1. All other start probabilities are therefore 0.

$$\begin{aligned} &\pi_1 = 1 \\ &\pi_{i,i \neq 1} = 0 \end{aligned}$$

[12] `http://ghmm.sourceforge.net/` (last visit 06.08.2009)
[13] `http://www.wiigee.com/` (last visit 06.08.2009)

Based on the recommendation found in the literature (Jurafsky and Martin, 2008; Fink, 2007; Rabiner, 1989), left-to-right models are used in this project. As a starting point, 8 hidden states and a step size $\Delta = 2$ are used as seen in Schlömer et al. (2008). In Mäntylä (2001) it has been reported that the number of states does not have significant effect on gesture recognition results. Nevertheless, an experiment is conducted in order to try 5 and 12 hidden states as well. The recognition results do not provide significant evidence in favour of a certain number of states (see appendix A.3). Therefore the 8 states that have been chosen initially, are kept for the final implementation in ABGLib.

A dump of the state transition matrix A can be found in section 4.5.4 (page 39). It shows the initial state probabilities of a left-to-right HMM with 8 states and step size 2, before training. The left-to-right character of the model can be comprehended by looking distribution of the zero, respectively non-zero, state-transition probabilities.

4.5.2 Training with Multiple Training Sequences

According to the theory of hidden Markov models, an HMM can only be trained with one training sequence. However, the recognition rate of a real-world gesture can be greatly improved by using several training sequences. By repeating a gesture in various different ways, ideally also by different people, it is possible to set out a certain range that describes the actual gesture.

As seen in Rabiner (1989), the Baum-Welch algorithm can be modified to operate on multiple training sequences. The set of Q training sequences is defined as $\mathrm{O} = \{O^{(1)}, O^{(2)} \ldots O^{(q)}\}$, where the qth observation sequence is:

$$O^{(q)} = \{O_1^{(q)}, O_2^{(q)}, \ldots O_{Tq}^{(q)}\}$$

The modified equations to calculate the improved model parameters are:

$$\hat{a}_{ij} = \frac{\sum_{q=1}^{Q} \frac{1}{P(O^{(q)}|\lambda)} \sum_{t=1}^{T_q-1} \alpha_t^q(i) a_{ij} b_j(O_{t+1}^q) \beta_{t=1}^q(j)}{\sum_{q=1}^{Q} \frac{1}{P(O^{(q)}|\lambda)} \sum_{t=1}^{T_q-1} \alpha_t^q(i) \beta_t^q(i)}$$

and

$$\hat{b}_j(o_k) = \frac{\sum_{q=1}^{Q} \frac{1}{P(O^{(q)}|\lambda)} \sum_{t:O_t^q=o_k} \alpha_t^q(j) \beta_t^q(j)}{\sum_{q=1}^{Q} \frac{1}{P(O^{(q)}|\lambda)} \sum_{t=1}^{T_q} \alpha_t^q(i) \beta_t^q(i)}$$

The start probabilities π_i do not need to be re-estimated. Since a left-to-right HMM is used, they remain fixed – with the first state having the maximum probability $\pi_1 = 1$.

4.5.3 HMMs and Digital Computing Systems

At first sight, the calculation of probability values for hidden Markov models with real world computing systems, seems to be a trivial problem. Therefore, the initial implementation of the HMM algorithms in *ABGLib* just followed the mathematical formulas as presented in sections 3.2 and 4.5.2. Although this initial HMM implementation basically worked, even in some practical tests with real gesture data, problems quickly arose, especially with longer data sequences and multiple training sequences.

The probability values involved in the computation lie all in the interval $[0 \ldots 1]$. Problems occur especially in longer computational procedures, when extremely small values lying close to zero, need to be represented. Such vanishingly small probability values can appear relatively quickly in computations for Markov models. For example, the simplified case of a Markov chain model shall be considered. The probability that a sample sequence O is generated by a given model λ can be computed as:

$$P(O|\lambda) = \prod_{t=1}^{T} a_{S_{t-1},S_t} \qquad\qquad S_t = O_t$$

The above formula represents the multiplication of all state transition probabilities involved in the generation of observation O. When the length of the sequence T becomes increasingly larger, very small numerical values, which can hardly be represented in today's digital computers occur. The smallest number (closest to zero), that can be stored in a double precision floating point variable is $2.2 \cdot 10^{-308}$, if the value becomes even smaller, a so-called underflow occurs, and the variable's content is set to zero.

This may appear as negligible inaccuracy as $2.2 \cdot 10^{-308}$ indeed is already a very small value. However, when working with full blown hidden Markov models and real world data, such small values occur surprisingly often and they are of significant relevance for the calculation of the formulas. Taking them as exactly zero leads to fatal errors. For example, the calculation of the forward variables α_i (see equation (3.2) in section 3.2.4), is basically a summation over a large number of very small values. If these values are taken as equal to zero, the result of the summation would still be zero. In contrast, when the actual very small values were considered, they summed up to a considerable value, which would usually again be within the machine's precision range.

To overcome this problem, the fist attempt was simply to change the floating point format from double precision to quadruple precision, a 16-byte (128-bit) floating point number. Indeed, this change solved the problem for the considered sample data that had suffered before from the underflow problem. Unfortunately, the quadruple precision format (`long double` in C) is not available on the mobile phone processor, but only on the development computer, where the data samples were processed off-line. Therefore, this easy modification does not provide a solution for the application on the phone.

One method for handling extremely small probability values consists in scaling these appropriately. The idea behind this solution is to multiply the probability values with a sufficient large scaling factor, then to perform the critical calculations and finally to restore the original scale. Although this method should theoretically work fine and is also documented in detail in Rabiner (1989), the actual implementation

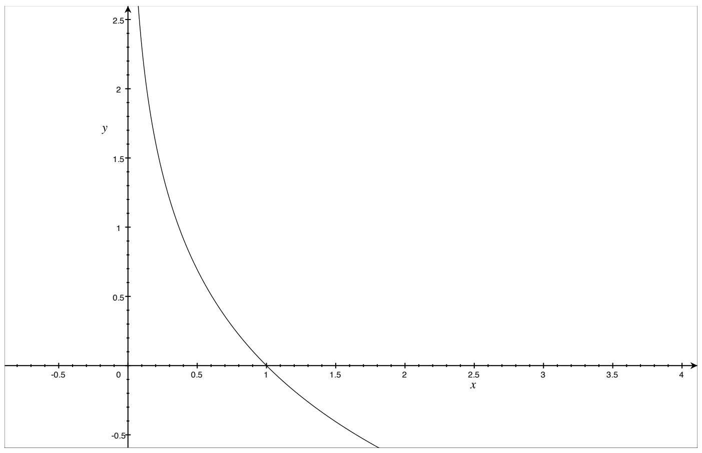


Figure 4.9: Negative Natural Logarithm $y = -ln(x)$

provided difficulties. The scaling factor needs to estimated dynamically for each observation at time t individually and then cancelled out again for computations on the same time t. Also, the method can only be applied locally, which makes it cumbersome with longer computations.

For the reasons mentioned above, another solution has been looked into and has been finally implemented. A method called *logarithmic probability representation*, found in Fink (2007), is applied. This method does not use real probability values in any computations. Instead, all probability values are represented on a negative logarithmic scale (see figure 4.9). To convert a probability value p from the normal linear scale to the negative logarithmic scale, the following transformation is applied:

$$\tilde{p} = -log_e\, p$$

By this transformation, the original range of values $[0 \ldots 1]$ for probabilities is mapped to the entire non-negative floating point number that can be represented with contemporary floating point formats. Although the resolution of this negative-logarithmic representation is confined by the same floating-point range, the accuracy is sufficient for the practical application of the method.

The negative logarithmic scaled probability value $\tilde{p}$ can be converted back to into the linear domain:

$$p = e^{-\tilde{p}}$$

Being in the logarithmic domain, the operations for the HMM formulas need to be converted as well. All multiplications in the formulas can now be written as simple additions, divisions become subtractions. However, doing additions when using the negative-logarithmic representation, becomes more complex. The values need to be converted first to the linear domain, then added, and finally transformed to the logarithmic domain again. The following formula illustrates the logarithmic addition of value $\tilde{p_1}$ and $\tilde{p_2}$:

$$\tilde{p_1} \ +_{(log)} \ \tilde{p_1} \ = -ln(e^{-\tilde{p_1}} + e^{-\tilde{p_2}})$$

This is quite a costly operation, especially for mobile phone processors. By applying the so-called Kingsbury-Rayner formula (Kingsbury and Rayner, 1971), one exponentiation can be saved:

$$\tilde{p_1} \ +_{(log)} \ \tilde{p_1} \ = \tilde{p_1} - ln(1 + e^{-(\tilde{p_2} - \tilde{p_1})})$$

There is still one exponentiation and one logarithm, which needs to be computed. To further improve performance, in the implemented C-program code the function `log1p(x)` is used, which is an optimised function for computing $ln(1 + x)$.

Now, the problem of representing and manipulating numerically vanishing quantities with digital computers can be solved by using the negative logarithmic scale. However, there is one small probelem left. What if 'real' zeros appear, e.g. to denote non-existing connections of a left-to-right HMM (see section 4.5.1). The negative natural logarithm of zero is positive infinity (see figure 4.9), which is again not defined in the numerical range of a digital computer.

The keyword for the solution to this problem is flooring. Flooring means defining a minimal value p_min and making sure that the probability values, such as state transition or emission probabilities, never fall below this minimal value. In the current implementation, the minimal value, the so-called floor, is set to 620 on the logarithmic scale, which maps to $5.46 \cdot 10^{-270}$ on the linear scale.

4.5.4 Training of an HMM

Prior to the training of a hidden Markov model, a fresh HMM needs to be instantiated. According to the literature (e.g. Rabiner, 1989), the various HMM parameters, such as the state transition matrix A and the emission probability matrix B, should be initialised as neutral as possible. The recommended method for choosing neutral initial values for A and B, is to use even probability distributions, which obey the stochastic constraints of the respective model configuration (Rabiner, 1989). A random initialisation in contrast, could bias the model in a negative way and prolong the training phase.

Following the recommendations, a freshly initialised state transition probability matrix of a 8-state HMM with step size 2 looks like this:

$$A = \begin{bmatrix} 0.3333 & 0.3333 & 0.3333 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.3333 & 0.3333 & 0.3333 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.3333 & 0.3333 & 0.3333 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.3333 & 0.3333 & 0.3333 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.3333 & 0.3333 & 0.3333 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.3333 & 0.3333 & 0.3333 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.5000 & 0.5000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

The emission probability matrix of an HMM with $N = 8$ states and observation alphabet size of $K = 20$ symbols is initialised as:

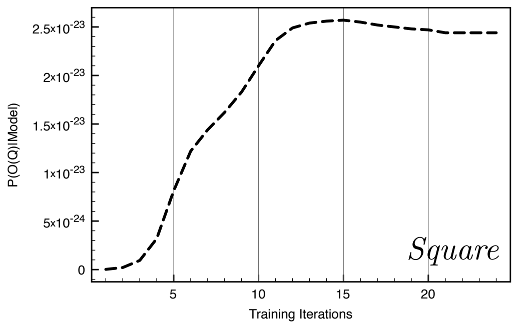
$$B = \{b_{jk}\}$$
$$b_{jk} = \frac{1}{K} = \frac{1}{20} = 0.05 \qquad 1 \leq j \leq N, \quad 1 \leq k \leq K$$

The start probabilities for a left-to-right HMM are given as $\pi_0 = 1$ and remain fix during the training phase.

When a new HMM for a gesture has been instantiated, it is trained with the Baum-Welch algorithm, which optimises the model for the given training sequences. The Baum-Welch training algorithm, and also its extended version for multiple training sequences, generally converges quite well towards a maximum likelihood for $P(O|\lambda)$ (see figure 4.10 and appendix A.3.2). According to the theory (see section 3.2.6), the training algorithm produces an improved HMM ($\hat{\lambda}$), after each training iteration:

$$P(O|\hat{\lambda}) \geq P(O|\lambda) \tag{4.1}$$

It has been observed that the relative increase of the likelihood can be very small during the first couple of training iterations. Therefore, a lower bound value for the likelihood-increase can not easily be used as a termination condition for the algorithm. Otherwise, there would be a risk that the training stops, before the major increase could have taken effect. Based on the obtained data samples and their training convergence, it has been decided to simply stop after a fixed number of training iterations. The number of training iterations was determined at 16. Figure 4.10 shows the training graphs for the gestures Square and Triangle. The full list of training graphs and the corresponding data can be found in appendix A.3.2.



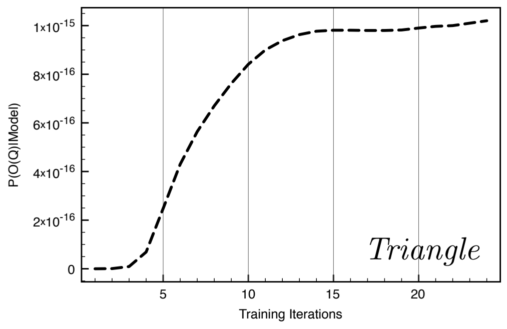


Figure 4.10: Convergence of Training Algorithm for Gestures Square (left) and Triangle (right)

As expressed in the above equation (4.1), the likelihood $P(O|\hat{\lambda})$ of the new model $\hat{\lambda}$ should always be greater, or at least stay the same, compared with the previous model λ. However, the practical test with the current training data shows that the likelihood sometimes slightly decreases. It is assumed that this is caused due to the limited accuracy when working with real-world computation machines.

4.6 Results and Evaluation

Based on the results of the experiments described in the previous sections of this chapter, the final values and configurations of the elements in the recognition and training system, which has been developed, can be summarised as follows.

The 3-dimensional acceleration vectors are read from the device sensor at 60 times per second. Acceleration vectors, which are less distant than 0.003 from the preceding input vector, are filtered out by the threshold filter. The vector quantiser maps the remaining input vectors onto a codebook of 28 prototype vectors. The codebook is initialised with the vectors of 28 points on a sphere, centred at the point of origin. The quantised sequence of symbols serves as an observation of the hidden Markov models. The hidden Markov Models are configured as left-to-right models with 8 hidden states and a step size of 2.

4.6.1 Recognition Rate

Given the setup as described above, the recognition results of the system are evaluated. The results are based on tests with 20 trials per each gesture. The corresponding data tables for these tests can be found in appendix A.4. Figure 4.11 shows the average recognition rate for the five test gestures, depending on the number of training sequences, which have been used to train the HMMs. A recognition rate of over 90 percent has been achieved with 10 training sequences.

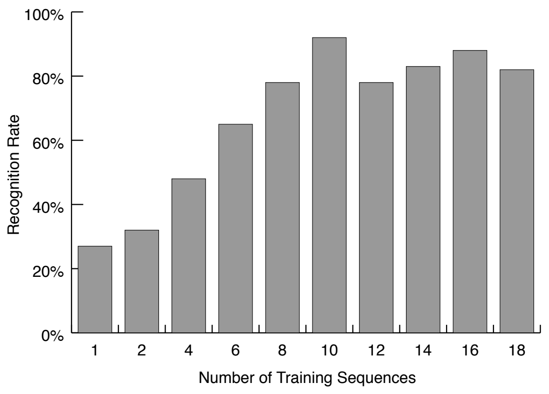


Figure 4.11: Average Recognition Results vs. Number of Training Sequences

Figure 4.12 shows the results for each of the five test gesture individually. The recognition results vary between 85 to 95 percent, depending on the gesture.

4.6.2 Speed of the System

The recognition of the gestures happens in quasi real-time. No delay between the end of the gesture drawing and the delivery of the recognition result is noticed. However, because the likelihood of the

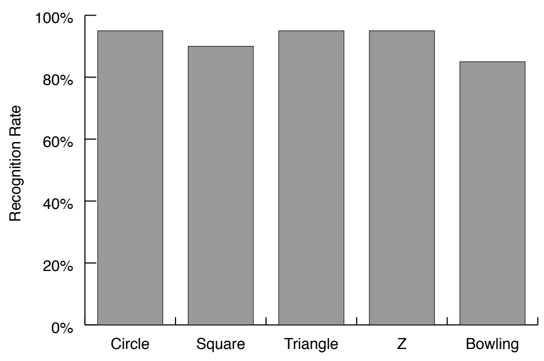


Figure 4.12: Recognition Results for Test Gestures (each trained with 10 repetitions)

input sequence needs to be calculated for all HMMs, the time it takes to find the best match depends on the number of gestures, which are available in the recognition pool. It also depends on the length and complexity of the gestures. With a recognition pool of the five test gestures defined in section 4.1, it takes 25 ms, on average, to determine the recognition result. This value measured on the iPhone 3G, which has been used for this project. On models with a faster processor, such as the iPhone 3G S, this value is expected to be smaller.

The time it takes for the training of a gesture hidden Markov model, on the other hand, is far from real-time. For example, when training the test gesture Square with 12 training sequences, it takes 8 minutes[14] for the training algorithm to complete.

4.6.3 False Positives

As described earlier in this paper, the likelihood that an observation sequence O was generated by the hidden Markov model λ, can be evaluated for any O, no matter how long O is. Therefore, it is possible to determine the most likely gesture HMM at any time while the gesture is drawn. It has been observed that short observation sequences can achieve relatively high probabilities. For example, when just drawing half a circle, it is likely that the gesture is recognised as a circle. By drawing a single stroke upwards, this may be recognised as the square gesture.

Because of this property of hidden Markov models, false positives can occur in the gesture recognition process. Also, this imposes some restrictions on how the gesture alphabet should be chosen. No gesture in the gesture alphabet should be contained within another gesture. For example it is not possible to differentiate between a circle and a semi-circle, by only using the techniques as presented in this paper.

A way to avoid this restriction was developed in the context of speech recognition. Instead of trying to model entire gestures (or words or phonemes in the context of speech recognition) with a single HMM, they are split up into smaller subparts. In HMM-based speech recognition so-called phones denote the

[14]This value has been determined on an iPhone 3G, it is expected be lower on faster iPhone models

smallest recognition entity. They represent a single speech sound and they are shorter than phonemes. These phones are then recognised within the speech signal, resulting in a sequence of probable phones. These phone probability sequences are thereafter processed by further hidden Markov models or markov chain models in order to recognise entire words and phrases. This technique could be adopted in the context of gesture recognition, by modelling small sub-gesture parts with HMMs. However, this approach could not be explored within the scope of this paper.

Chapter 5

ABGLib Objective-C Library and ABGDemo iPhone App

This project yields the Objective-C library ABGLib. It provides methods for accelerometer-based gesture recognition with the iPhone and it depends on Apple's Cocoa Touch framework[1]. The library is designed to facilitate easy integration with any iPhone application. The iPhone application ABGDemo, which also results from this paper, demonstrates the features available in ABGLib.

The demonstration application allows the training of arbitrary gestures, which are thereafter available to be recalled. When a gesture is recognised, the gesture name is displayed on the screen. In case the captured gesture movement does not match a previously trained gesture model, the the text "unknown gesture" is displayed.

In the following, a short walk-through through the basic functionality provided by ABGLib is given. An introduction to the Objective-C programming language and Apple's Cocoa framework can not be provided within the scope of this paper[2]. However, it should still be possible to follow the explanations below, as the names of the commands in ABGLib give an indication of their function.

In order to be concise with most existing program code, the code written for ABGLib and ABGDemo follows the American English spelling rules. Code comments, however, are kept in British English. ABGLib is written in a mixture of C and Objective-C code, but the C parts are only used internally. The library exposes a comfortable, high-level interface, which is entirely in Objective-C.

[1] `http://developer.apple.com/technology/cocoa.html` (last visit 04.09.2009)

[2] A short introduction is can be found in: `http://developer.apple.com/mac/library/referencelibrary/GettingStarted/Learning_Objective-C_A_Primer/index.html` (last visit: 04.09.2009)

5.1 Recognising Gestures

The fist step is to instantiate a singleton object of the class ABGController. The controller object is the central element of the training and recognition system. When it is called for the first time in the client application, basic initialisation tasks, such as the registration for the retrieval of accelerometer data (see section 4.2.2) and allocation of a buffer for the incoming data, are executed.

```
[[ABGController sharedController];
```

Provided there is already a file containing the relevant HMM gesture data, the following command unpacks the archive and loads the gesture models so that they are available for recognition immediately.

```
NSMutableArray *gestures = [[ABGController sharedController] unarchiveGestures];
```

By default, the system is in recognition mode and the capturing of accelerometer data for the classification is initiated with:

```
[[ABGController sharedController] motionStart];
```

Now, ABGController is capturing the accelerometer data, and the user can move the device and draw a gesture. When the user is done, the client application needs to send the message `motionStop` to the controller.

```
[[ABGController sharedController] motionStop];
```

The demonstration application ABGDemo sends `motionStart` when the screen of the device is touched. Then the user performs the gesture movement and then releases the finger from the screen. Upon the release of the finger, ABGDemo sends the `motionStop` message.

The recognition of the gestures happens automatically after `motionStop` has been received by the controller. The result of the recognition process is broadcasted in the form of a so-called NSNotification. NSNotification is a notification service provided by the Cocoa framework. The objects of the client application, who are interested in the recognition results can register to receive these notifications:

```
[[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(displayGestureRecognized:)
        name:ABGGestureRecognizedNotification object:nil];

[[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(displayGestureNotRecognized:)
        name:ABGGestureNotRecognizedNotification object:nil];
```

The use of notifications has the advantage that ABGController does not need to know anything about the objects, who should receive the recognition results, which in turn adheres to reusability and independence of ABGLib.

In the listing above, it can be seen that the controller sends out a different notification, depending on whether or not the captured input data matches one of the gestures. The notification named `ABGGestureRecognizedNotification` is sent in case a gesture was recognised, `ABGGestureNotRecognizedNotification` is sent when nothing was recognised. Therefore, a client application which does not want to be bothered with unrecognised gestures may just register to the former notification.

Upon the reception of a notification, the associated selector (see listing above) is invoked. The selector refers to a method, which is implemented by the object, which receives the notifications. In case the gesture was successfully recognised, the notification object contains the name of the gesture.

```
- (void)displayGestureRecognized:(NSNotification*)note {
        NSString *gestureName = [[note userInfo] objectForKey:@"gestureName"];
        NSLog(@"Gesture %@ was recognised.", gestureName);
}
- (void)displayGestureNotRecognized:(NSNotification*)note {
        NSLog(@"-- unkown gesture  --");
}
```

5.2 Training of New Gestures

In case a gesture archive with pre-trained gestures does not yet exist, or a new gesture should be added to the gesture pool, a fresh, untrained gesture object needs to be created. A new ABGGesture object is allocated and initialised with the name "New Gesture" and added to the arrays of gestures as follows:

```
ABGGesture *gesture = [[ABGGesture alloc] initWithName:@"New Gesture"];
[gestures addObject:gesture];
```

Then, the controller needs to be set to training mode:

```
[[ABGController sharedController] setInTrainingMode:YES];
```

When the ABGController is in training mode, training sequences for the new gesture can be captured in the same way as seen in the part on recognition above:

```
[[ABGController sharedController] motionStart];
```

Now the user can perform the gesture in order to generate one training sequence. When done, the client application needs to message `motionStop` to terminate the training sequence.

```
[[ABGController sharedController] motionStop];
```

All `motionStart` and `motionStop` events are also sent out as notifications. A training controller object, which needs to be implemented by the client application, can register to receive the `motionStop` events, the corresponding notification is named `ABGMotionEndedNotification`:

```
[[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(newTrainingSequenceArrived:)
        name:ABGMotionEndedNotification object:[ABGController sharedController]];
```

The arrival of a `ABGMotionEndedNotification` indicates that a new training sequence has been recorded. It needs to be assigned to the respective gesture object:

```
- (void)newTrainingSequenceArrived:(NSNotification*)note {
        [gesture addTrainingSequence];
}
```

In the same way, more training sequences can be recorded and added to the gesture object. When in enough training sequences have been collected, the training of the gesture model can be initiated:

```
[gesture learnFromTrainingSequences];
```

When the training process has finished, the gesture is available for recognition and the ABGController needs to be switched back into recognition mode:

```
[[ABGController sharedController] setInTrainingMode:NO];
```

At this point, it is recommended to save the trained gesture models permanently to disk. This ensures that they are still available, when the application is closed and reopened.

```
[[ABGController sharedController] archiveGestures];
```

Chapter 6

Conclusion and Prospects

Accelerometer-based gesture recognition has been studied as an alternative interaction modality, providing new possibilities to interact with mobile devices. Sensor-based gesture control has some advantages compared with more traditional modalities. Gestures require no eye focus on the interface, and they are silent. For certain tasks, a hand gesture may feel more natural than pressing a button on a keyboard, for example.

A gesture recognition application has been implemented for the iPhone and its core features have been consolidated into a reusable Objective-C library. The iPhone's built-in accelerometer is used to capture sensor data generated by hand movements. Discrete hidden Markov models, which work in combination with a vector quantiser, lie at the heart of the recognition system. Various tests have been conducted in order to optimise certain application specific values, such as the codebook size and the initial prototype vectors. The final system reaches recognition results between 80 to 95 percent, which is promising but leaves room for further optimisation.

Gestures can be freely trained and are thereafter available for recognition. The training with 10 repetitions per gesture yielded an average recognition rate of over 90 percent.

In order to achieve better recognition results, it is suggested for future works to examine the following techniques. The vector quantiser codebook has significant influence on the recognition quality. As an alternative to the Lloyds algorithm, employed in this project, the input vectors could be analysed by self-organising maps (Kohonen, 1982) in order to select suitable prototype vectors for the vector quantisation process.

The occurrence of false positives, as described in section 4.6.3, should be reduced. Instead of modelling an entire gesture with one hidden Markov model, it is suggested to try splitting up each gesture into unique smaller parts. These gesture parts could then be recognised individually, and the resulting sequence could eventually be classified into the full gestures.

A further approach, which is suggested, is the use of continuous hidden Markov models, rather than discrete hidden Markov models, which are currently utilised. Most of today's speech recognition systems are based on continuous hidden Markov models (Fink, 2007). They are able to operate directly on multi-dimensional vector data. Therefore, the vector quantisation step could be saved, which hopefully results in higher quality recognition results. On the other hand, continuous hidden Markov models are computationally more complex, which may move them out of reach for the real-time use with today's mobile devices.

Besides the improvements of the recognition technique itself, further, more fundamental amendments are considered. Firstly, in the current implementation the user needs to touch the screen of the device, while he or she is acting out the gesture. The necessity to touch the screen in order to indicate the beginning and end of a gesture should be eliminated. A simple solution for so-called auto-segmentation would be the automated detection of the phases, the device is in motion.

Secondly, the system, which has been developed in this project is user-dependent. This means, gestures are trained and recalled by one person only. A user-independent recognition system in contrast, should provide reliable recognition results for a wide range of people interacting with the system. In order to train gestures for a user-independent systems, training sequences need to be captured from various people. As a continuation of this project, it is suggested to collect training sequences in a distributed way through several devices and people. Training sequences could be collected on a central server, and also the training algorithms could run on a remote server. Thereafter, the updated gesture models could be sent back to the client devices.

Bibliography

Bergamasco, M., Frisoli, A., and Avizzano, C. (2007). Exoskeletons as Man-Machine interface systems for teleoperation and interaction in virtual environments. In *Advances in Telerobotics*, pages 61–76.

Corradini, A. (2001). Dynamic time warping for Off-Line recognition of a small gesture vocabulary. In *Proceedings of the IEEE ICCV Workshop on Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems (RATFG-RTS'01)*, page 82. IEEE Computer Society.

Dempster, A., Laird, N., and Rubin, D. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):38, 1.

Elmezain, M., Al-Hamadi, A., Appenrodt, J., and Michaelis, B. (2008). A hidden markov model-based continuous gesture recognition system for hand motion trajectory. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4.

Eshbach, O. W., Tapley, B. D., and Poston, T. R. (1990). *Eshbach's handbook of engineering fundamentals.* Wiley-IEEE.

Fink, G. A. (2007). *Markov Models for Pattern Recognition.* Springer.

Haykin, S. S. (1999). *Neural networks.* Prentice Hall.

Hofmann, F., Heyer, P., and Hommel, G. (1998). Velocity profile based recognition of dynamic gestures with discrete hidden markov models. In *Gesture and Sign Language in Human-Computer Interaction*, pages 81–95.

Jurafsky, D. and Martin, J. H. (2008). *Speech and language processing.* Prentice Hall.

Kalawsky, R. S. (1993). *The science of virtual reality and virtual environments.* Addison-Wesley.

Kela, J., Korpipää, P., Mäntyjärvi, J., Kallio, S., Savino, G., Jozzo, L., and Marca, D. (2006). Accelerometer-based gesture control for a design environment. *Personal Ubiquitous Comput.*, 10(5):285–299.

Kingsbury, N. and Rayner, P. (1971). Digital filtering using logarithmic arithmetic. *Electronics Letters*, 7(2):56–58.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69.

Kuijlaars, A. B. J., Saff, E. B., and R, I. (1998). Asymptotics for minimal discrete energy on the sphere. *TRANS. AMER. MATH. SOC*, 350:523—538.

Mäntylä, V. M. (2001). *Discrete Hidden Markov Models with Application to Isolated User-Dependent Hand Gesture Recognition*. VTT Publications 449.

Prekopcsák, Z. (2008). Accelerometer based real-time gesture recognition. In *Proceedings of the 12th International Student Conference on Electrical Engineering*, Prague, Czech Republic.

Prekopcsák, Z., Halácsy, P., and Gáspár-Papanek, C. (2008). Design and development of an everyday hand gesture interface. In *Proceedings of the 10th international conference on Human computer interaction with mobile devices and services*, pages 479–480, Amsterdam, The Netherlands. ACM.

Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, volume 77, pages 257—286.

Saffer, D. (2008). *Designing Gestural Interfaces: Touchscreens and Interactive Devices*. O'Reilly Media, Inc., illustrated edition edition.

Schlömer, T., Poppinga, B., Henze, N., and Boll, S. (2008). Gesture recognition with a wii controller. In *Proceedings of the 2nd international conference on Tangible and embedded interaction*, pages 11–14, Bonn, Germany. ACM.

Seniuk, A. and Blostein, D. (2009). Pen acoustic emissions for text and gesture recognition. In *10th International Conference on Document Analysis and Recognition*, Barcelona, Spain.

Tuulari, E. and Ylisaukko-oja, A. (2002). SoapBox: a platform for ubiquitous computing research and applications. In *Proceedings of the First International Conference on Pervasive Computing*, pages 125–138. Springer-Verlag.

Vogler, C. and Metaxas, D. (2001). A framework for recognizing the simultaneous aspects of american sign language. *Comput. Vis. Image Underst.*, 81(3):358–384.

Yang, M. and Ahuja, N. (2001). *Face detection and gesture recognition for human-computer interaction*. Springer.

Yang, M. H., Ahuja, N., and Tabb, M. (2002). Extraction of 2D motion trajectories and its application to hand gesture recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(8):1061–1074.

Appendix A

Figures and Tables

A.1 Data Pre-processing

A.1.1 Threshold Filter

Table A.1: Reduction of Input Vectors with Threshold Filter Value 0.01

		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Average	Reduction
Circle	unfiltered	80	83	89	58	48	60	69.7	
	filtered	28	30	30	41	33	44	34.3	50.7%
Square	unfiltered	81	75	111	91	100	90	91.3	
	filtered	55	55	60	53	48	45	52.7	42.3%
Triangle	unfiltered	61	78	82	93	81	87	80.3	
	filtered	27	39	34	31	38	43	35.3	56.0%
Z	unfiltered	65	61	66	77	77	76	70.3	
	filtered	47	47	47	48	41	35	44.2	37.2%
Bowling	unfiltered	117	92	95	106	128	115	108.8	
	filtered	68	71	71	70	88	73	73.5	32.5%
					Average Reduction of Input Vectors				43.7%

Table A.2: Reduction of Input Vectors with Threshold Filter Value 0.005

		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Average	Reduction
Circle	unfiltered	74	80	99	71	84	97	84.2	
	filtered	51	62	51	52	50	47	52.2	38.0%
Square	unfiltered	77	96	95	79	87	120	92.3	
	filtered	64	86	75	68	72	72	72.8	21.1%
Triangle	unfiltered	68	79	84	86	79	80	79.3	
	filtered	55	56	56	62	60	58	57.8	27.1%
Z	unfiltered	66	86	74	77	76	77	76.0	
	filtered	58	59	56	52	53	57	55.8	26.5%
Bowling	unfiltered	94	120	113	115	130	125	116.2	
	filtered	85	98	83	93	86	81	87.7	24.5%
					Average Reduction of Input Vectors				27.5%

Table A.3: Reduction of Input Vectors with Threshold Filter Value 0.003

		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Average	Reduction
Circle	unfiltered	69	68	77	71	77	73	72.5	
	filtered	65	68	60	60	63	61	62.8	13.3%
Square	unfiltered	73	82	97	101	79	113	90.8	
	filtered	67	74	82	92	74	88	79.5	12.5%
Triangle	unfiltered	73	78	96	67	92	77	80.5	
	filtered	57	62	74	59	77	62	65.2	19.0%
Z	unfiltered	74	83	77	80	72	67	75.5	
	filtered	66	69	69	60	65	61	65.0	13.9%
Bowling	unfiltered	121	132	118	121	126	115	122.2	
	filtered	107	99	99	107	108	98	103.0	15.7%
					Average Reduction of Input Vectors				14.9%

Table A.4: Recognition Results with Threshold Filter Value 0.01

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	10	0	10	20	50%
Square	8	0	12	20	40%
Triangle	8	0	12	20	40%
Z	10	0	10	20	50%
Bowling	6	0	14	20	30%
Average					42%

Table A.5: Recognition Results with Threshold Filter Value 0.005

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	12	0	8	20	60%
Square	11	1	8	20	55%
Triangle	10	2	8	20	50%
Z	11	0	9	20	55%
Bowling	5	0	15	20	25%
Average					49%

Table A.6: Recognition Results with Threshold Filter Value 0.003

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	14	0	6	20	70%
Square	15	1	4	20	75%
Triangle	12	0	8	20	60%
Z	13	1	6	20	65%
Bowling	8	0	12	20	40%
Average					62%

Table A.7: Recognition Results without Threshold Filter (Filter Value 0.0)

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	14	0	6	20	70%
Square	15	0	5	20	75%
Triangle	11	0	9	20	55%
Z	13	1	6	20	65%
Bowling	8	0	12	20	40%
Average					61%

Table A.8: Summary Tested Threshold Filter Values.

Filter Value ϵ^2	Average Reduction of Input Vectors	Recognition Results Percent
0.01	43.3%	42%
0.005	27.5%	49%
0.003	14.9%	62%
0.0	0%	61%

A.2 Vector Quantiser

A.2.1 Random Codebook Initialisation

Table A.9: Quantisation Error for Random Codebook Initialisation.

Gesture	Quantisation MSE
Circle	0.062362
Square	0.037210
Triangle	0.036087
Z	0.040684
Bowling	0.045375
Average	0.044343

Table A.10: Recognition Results for Random Codebook Initialisation

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	6	0	14	20	30%
Square	14	0	6	20	70%
Triangle	5	1	14	20	25%
Z	11	9	0	20	55%
Bowling	6	6	8	20	30%
Average					42%

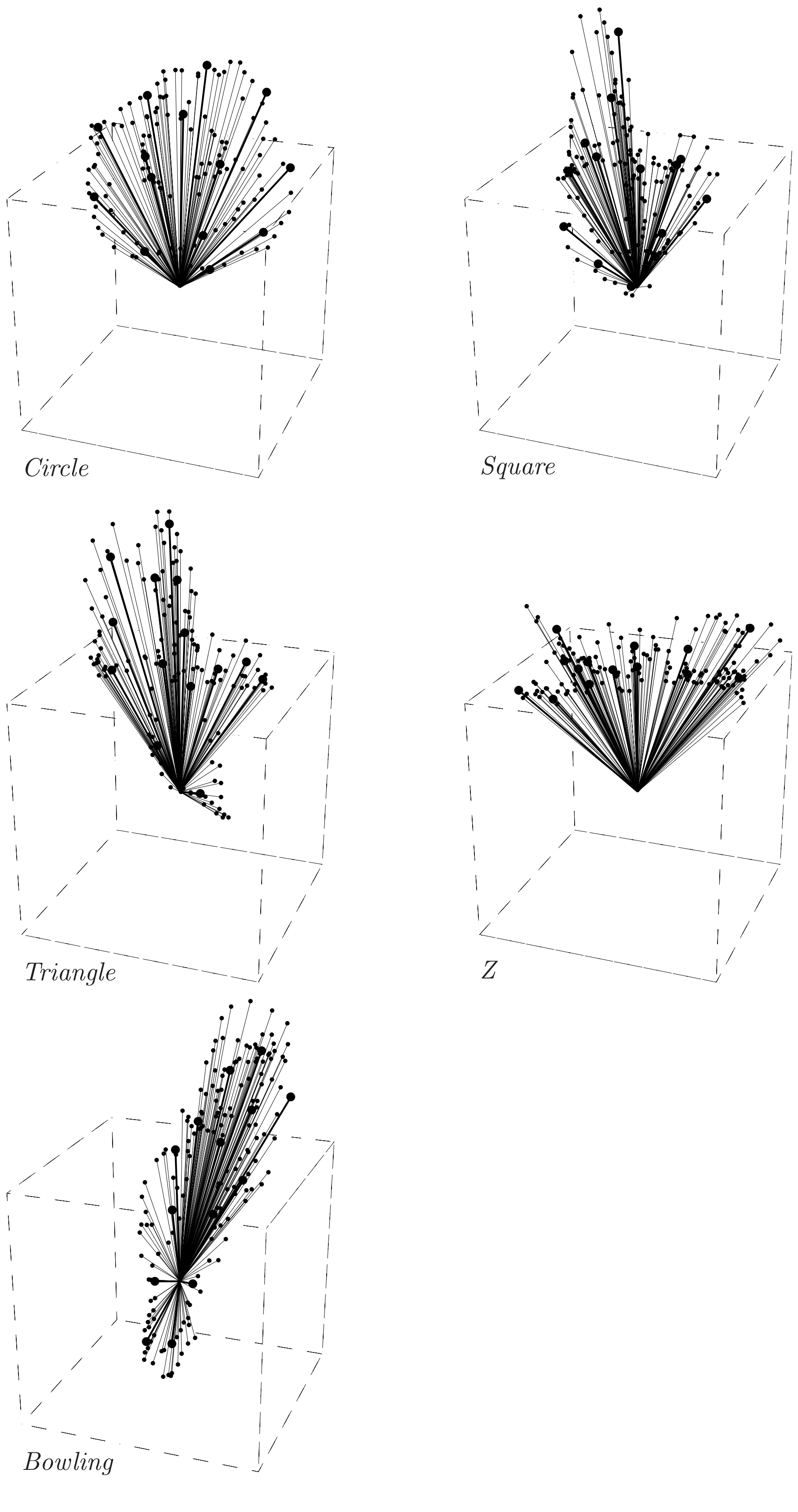


Figure A.1: Vector Visualisation. Random Codebook Initialisation

A.2.2 Spherical Codebook Initialisation

Table A.11: Quantisation Error for Spherical Codebook Initialisation

Gesture	Quantisation MSE
Circle	0.109650
Square	0.050763
Triangle	0.058436
Z	0.084859
Bowling	0.075767
Average	0.075895

Table A.12: Recognition Results for Spherical Codebook Initialisation

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	8	0	12	20	40%
Square	16	2	2	20	80%
Triangle	15	1	4	20	75%
Z	17	0	3	20	85%
Bowling	8	0	12	20	40%
Average					64%

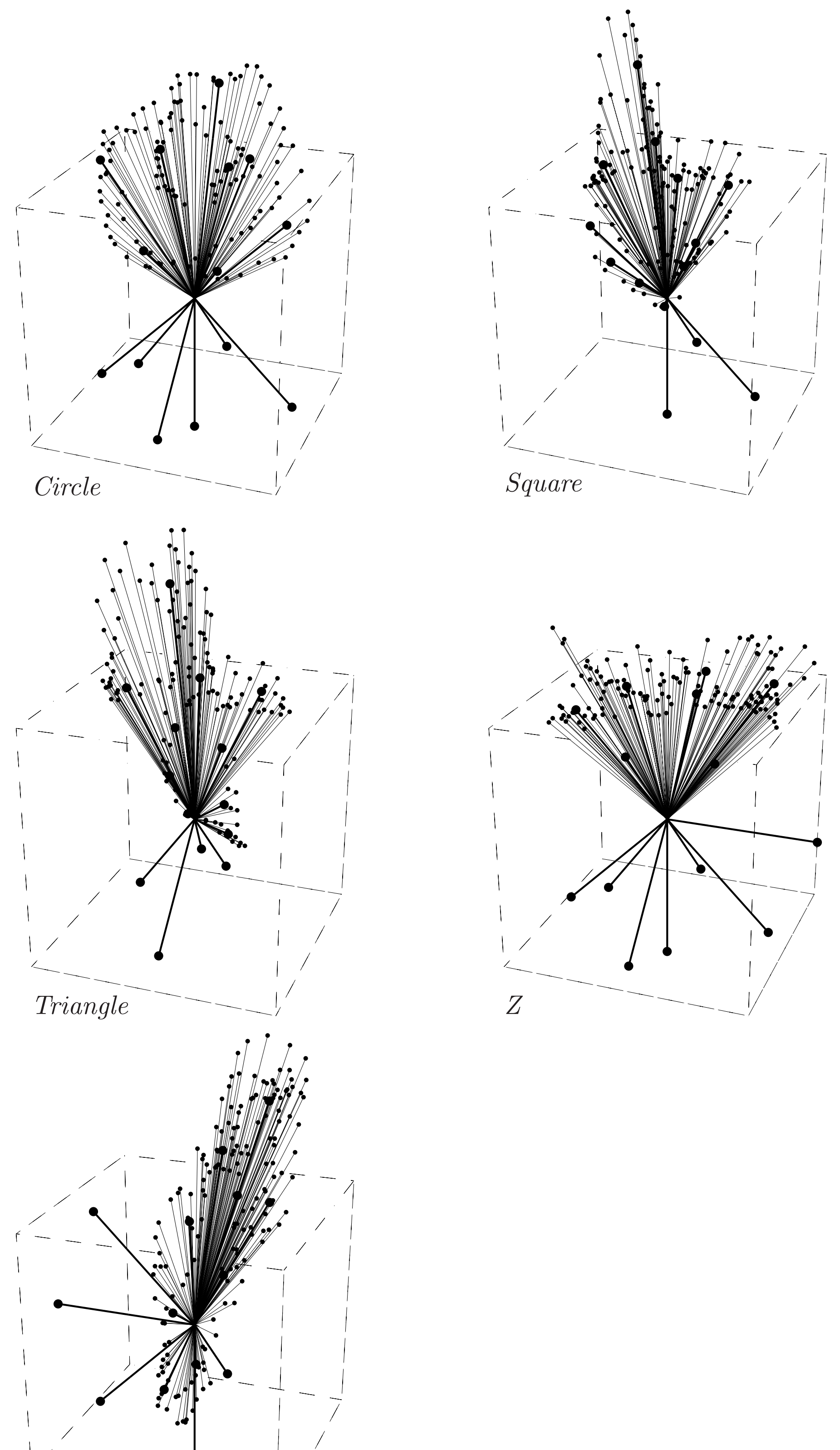


Figure A.2: Vector Visualisation. Spherical Codebook Initialisation

A.2.3 Spherical Codebook Initialisation with High-Pass Filter

Table A.13: Quantisation Error for Spherical Codebook Initialisation with High-Pass Filter

Gesture	Quantisation MSE
Circle	0.039947
Square	0.034614
Triangle	0.021848
Z	0.020398
Bowling	0.098015
Average	0.042964

Table A.14: Recognition Results for Spherical Codebook Initialisation with High-Pass Filter

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	8	4	8	20	40%
Square	11	4	5	20	55%
Triangle	6	3	11	20	30%
Z	14	0	6	20	70%
Bowling	5	4	11	20	25%
Average					44%

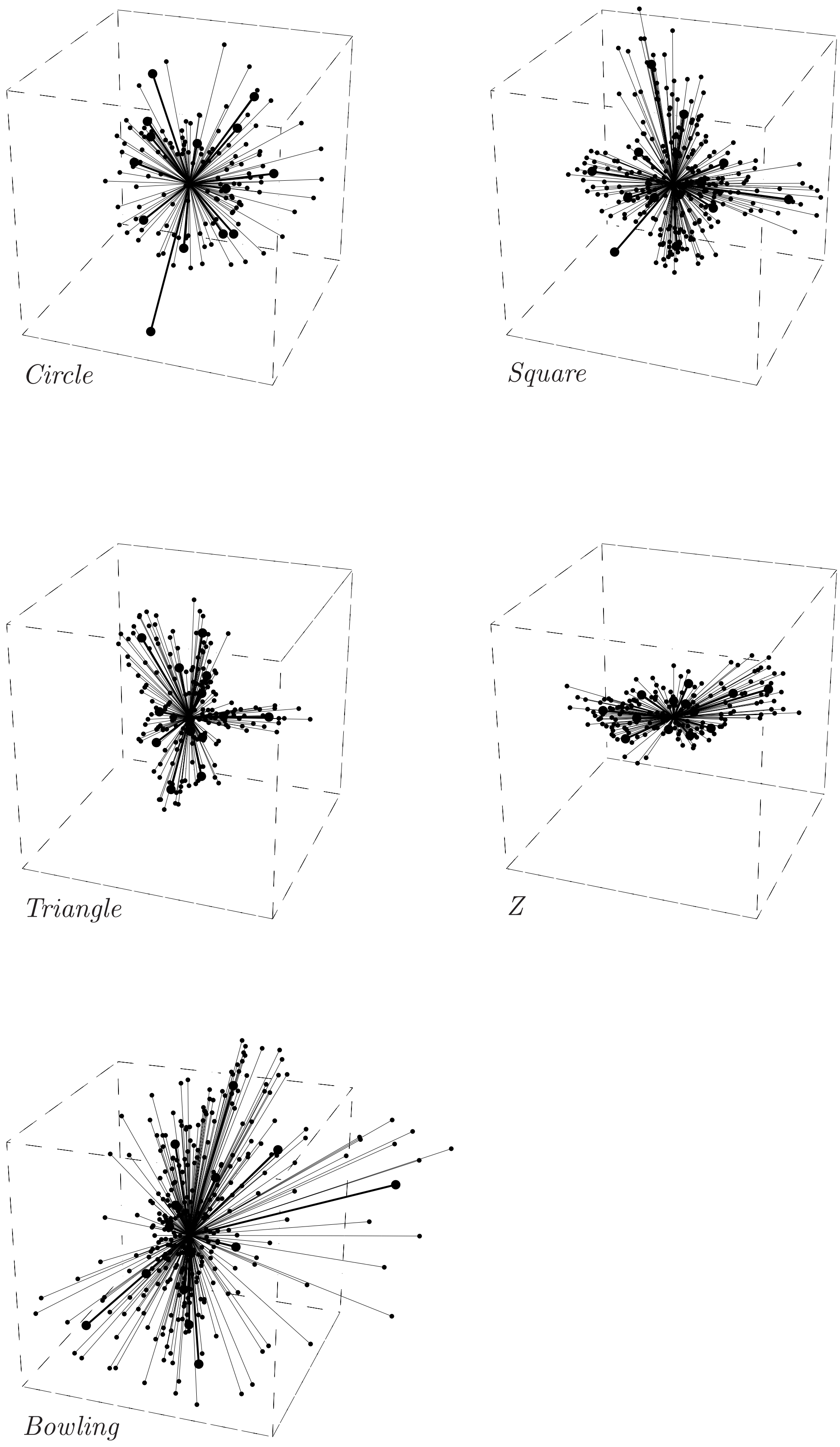


Figure A.3: Vector Visualisation. Spherical Codebook Initialisation with High-Pass Filter

A.2.4 Spherical Codebook Initialisation with Offset

Table A.15: Quantisation Error for Spherical Codebook Initialisation with Offset

Gesture	Quantisation MSE
Circle	0.065712
Square	0.040789
Triangle	0.034489
Z	0.046274
Bowling	0.054871
Average	0.048427

Table A.16: Recognition Results for Spherical Codebook Initialisation with Offset

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	2	6	12	20	10%
Square	12	6	2	20	60%
Triangle	5	2	13	20	25%
Z	16	0	4	20	80%
Bowling	7	2	11	20	35%
Average					42%

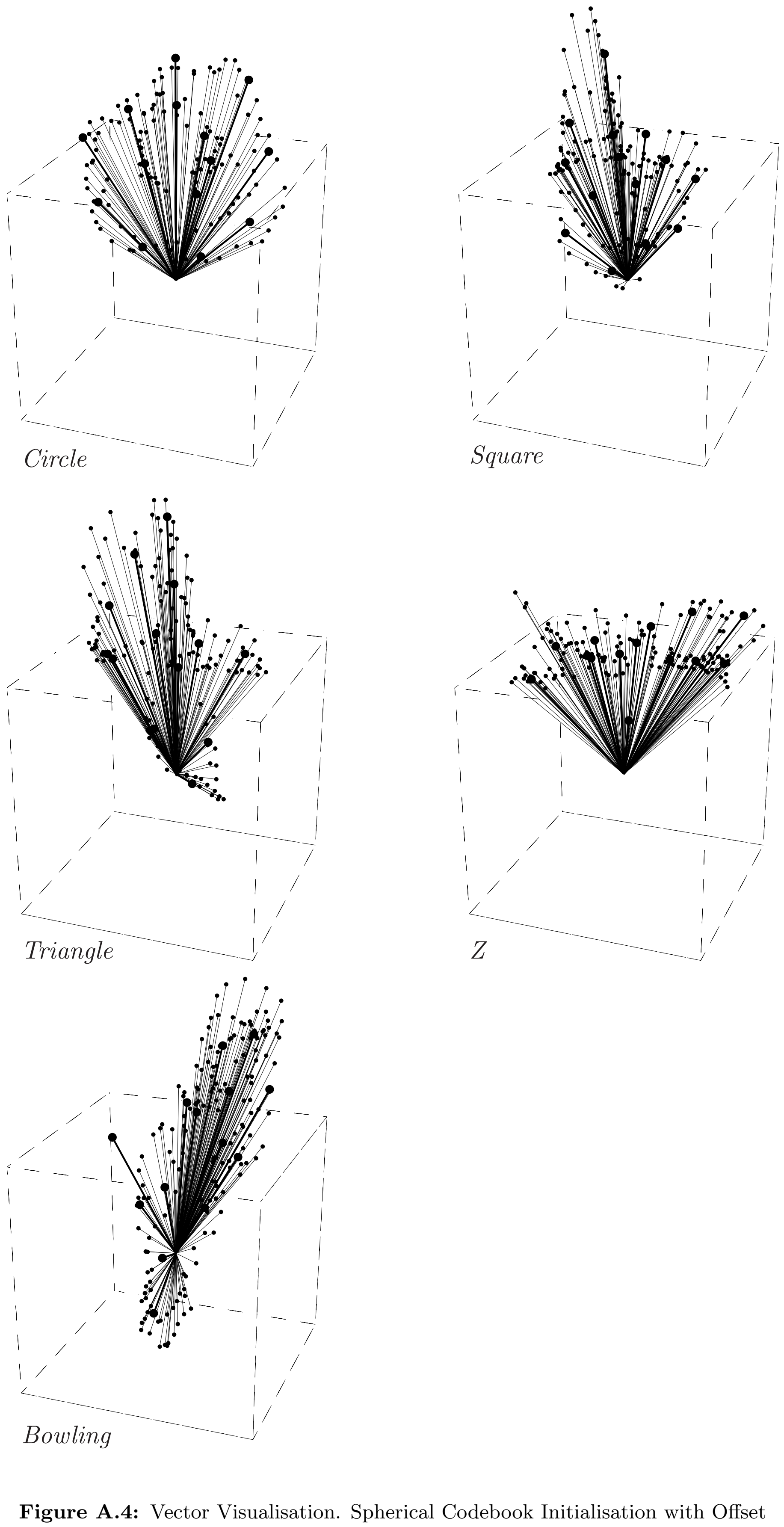


Figure A.4: Vector Visualisation. Spherical Codebook Initialisation with Offset

A.2.5 Codebook Size

Table A.17: Quantisation Error for Codebook Size 8

Gesture	Quantisation MSE
Circle	0.138621
Square	0.063813
Triangle	0.084635
Z	0.096249
Bowling	0.111965
Average	0.099056

Table A.18: Recognition Results for Codebook Size 8

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	7	12	1	20	35%
Square	8	12	0	20	40%
Triangle	18	1	1	20	90%
Z	20	0	0	20	100%
Bowling	9	2	9	20	45%
Average					62%

Table A.19: Quantisation Error for Codebook Size 14

Gesture	Quantisation MSE
Circle	0.097587
Square	0.057823
Triangle	0.058720
Z	0.096956
Bowling	0.082372
Average	0.078692

Table A.20: Recognition Results for Codebook Size 14

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	18	0	2	20	90%
Square	16	2	2	20	80%
Triangle	12	0	8	20	60%
Z	14	0	6	20	70%
Bowling	1	0	19	20	5%
Average					61%

Table A.21: Quantisation Error for Codebook Size 20

Gesture	Quantisation MSE
Circle	0.075953
Square	0.046028
Triangle	0.057496
Z	0.066434
Average	0.060433

Table A.22: Recognition Results for Codebook Size 20

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	7	12	1	20	35%
Square	8	12	0	20	40%
Triangle	18	1	1	20	90%
Z	20	0	0	20	100%
Bowling	9	2	9	20	45%
Average					62%

Table A.23: Quantisation Error for Codebook Size 28

Gesture	Quantisation MSE
Circle	0.062653
Square	0.046091
Triangle	0.037759
Z	0.062663
Bowling	0.056571
Average	0.053147

Table A.24: Recognition Results for Codebook Size 28

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	18	0	2	20	90%
Square	11	0	9	20	55%
Triangle	15	0	5	20	75%
Z	17	0	3	20	85%
Bowling	0	0	20	20	0%
Average					61%

Table A.25: Quantisation Error for Codebook Size 36

Gesture	Quantisation MSE
Circle	0.059399
Square	0.030851
Triangle	0.044497
Z	0.060778
Bowling	0.041856
Average	0.047476

Table A.26: Recognition Results for Codebook Size 36

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	18	0	2	20	90%
Square	13	0	7	20	65%
Triangle	11	0	9	20	55%
Z	14	0	6	20	70%
Bowling	0	0	20	20	0%
Average					56%

A.3 Hidden Markov Models

A.3.1 Configurations with 5, 8 and 12 Hidden States

Table A.27: Recognition Results for 5 Hidden States

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	12	0	8	20	60%
Square	14	0	6	20	70%
Triangle	11	0	9	20	55%
Z	13	1	6	20	65%
Bowling	8	0	12	20	40%
Average					58%

Table A.28: Recognition Results for 8 Hidden States

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	13	0	7	20	65%
Square	14	0	6	20	70%
Triangle	12	0	8	20	60%
Z	12	1	7	20	60%
Bowling	7	0	13	20	35%
Average					58%

Table A.29: Recognition Results for 12 Hidden States

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	14	0	6	20	70%
Square	13	1	6	20	65%
Triangle	10	0	10	20	50%
Z	11	1	8	20	55%
Bowling	9	0	11	20	45%
Average					57%

A.3.2 Training Algorithm Convergence, Graphs

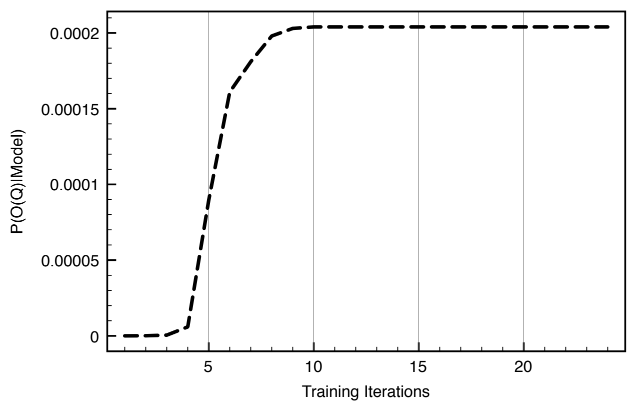


Figure A.5: Training Algorithm Convergence. Circle

Table A.30: Training Algorithm Convergence. Circle

Iteration	$P(O^{(Q)}\|Model)$	Delta	SD
1	3.87×10^{-9}	3.87×10^{-9}	7.59×10^{-18}
2	7.83×10^{-8}	7.44×10^{-8}	3.36×10^{-15}
3	4.82×10^{-7}	4.04×10^{-7}	1.29×10^{-13}
4	5.92×10^{-6}	5.43×10^{-6}	1.96×10^{-11}
5	8.96×10^{-5}	8.37×10^{-5}	4.63×10^{-9}
6	1.61×10^{-4}	7.09×10^{-5}	1.56×10^{-8}
7	1.81×10^{-4}	2.06×10^{-5}	2.09×10^{-8}
8	1.98×10^{-4}	1.65×10^{-5}	2.56×10^{-8}
9	2.03×10^{-4}	5.77×10^{-6}	2.74×10^{-8}
10	2.04×10^{-4}	6.97×10^{-7}	2.76×10^{-8}
11	2.04×10^{-4}	-3.50×10^{-7}	2.75×10^{-8}
12	2.04×10^{-4}	9.53×10^{-15}	2.75×10^{-8}
13	2.04×10^{-4}	-2.36×10^{-17}	2.75×10^{-8}
14	2.04×10^{-4}	2.95×10^{-17}	2.75×10^{-8}
15	2.04×10^{-4}	-1.16×10^{-17}	2.75×10^{-8}
16	2.04×10^{-4}	-3.85×10^{-17}	2.75×10^{-8}
17	2.04×10^{-4}	-2.11×10^{-17}	2.75×10^{-8}
18	2.04×10^{-4}	3.49×10^{-17}	2.75×10^{-8}
19	2.04×10^{-4}	-3.90×10^{-17}	2.75×10^{-8}
20	2.04×10^{-4}	6.15×10^{-17}	2.75×10^{-8}
21	2.04×10^{-4}	-9.32×10^{-18}	2.75×10^{-8}
22	2.04×10^{-4}	6.78×10^{-19}	2.75×10^{-8}
23	2.04×10^{-4}	5.07×10^{-17}	2.75×10^{-8}
24	2.04×10^{-4}	-4.35×10^{-17}	2.75×10^{-8}

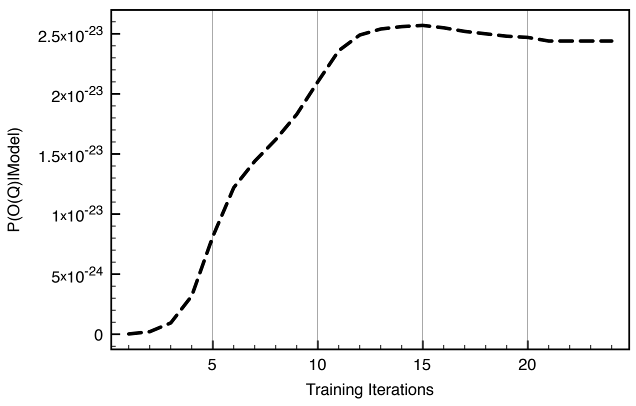


Figure A.6: Training Algorithm Convergence. Square

Table A.31: Training Algorithm Convergence. Square

Iteration	$P(O^{(Q)}\|Model)$	Delta	SD
1	2.32×10^{-26}	2.32×10^{-26}	5.37×10^{-52}
2	2.12×10^{-25}	1.89×10^{-25}	4.50×10^{-50}
3	9.41×10^{-25}	7.29×10^{-25}	8.86×10^{-49}
4	3.19×10^{-24}	2.24×10^{-24}	1.01×10^{-47}
5	8.12×10^{-24}	4.93×10^{-24}	6.59×10^{-47}
6	1.22×10^{-23}	4.09×10^{-24}	1.49×10^{-46}
7	1.44×10^{-23}	2.16×10^{-24}	2.07×10^{-46}
8	1.62×10^{-23}	1.79×10^{-24}	2.61×10^{-46}
9	1.83×10^{-23}	2.09×10^{-24}	3.33×10^{-46}
10	2.10×10^{-23}	2.79×10^{-24}	4.43×10^{-46}
11	2.36×10^{-23}	2.58×10^{-24}	5.58×10^{-46}
12	2.49×10^{-23}	1.30×10^{-24}	6.21×10^{-46}
13	2.54×10^{-23}	4.60×10^{-25}	6.44×10^{-46}
14	2.56×10^{-23}	2.54×10^{-25}	6.57×10^{-46}
15	2.57×10^{-23}	3.21×10^{-26}	6.59×10^{-46}
16	2.55×10^{-23}	-2.05×10^{-25}	6.48×10^{-46}
17	2.52×10^{-23}	-2.54×10^{-25}	6.35×10^{-46}
18	2.50×10^{-23}	-2.22×10^{-25}	6.24×10^{-46}
19	2.48×10^{-23}	-1.81×10^{-25}	6.15×10^{-46}
20	2.47×10^{-23}	-1.29×10^{-25}	6.09×10^{-46}
21	2.44×10^{-23}	-2.27×10^{-25}	5.98×10^{-46}
22	2.44×10^{-23}	-1.26×10^{-26}	5.97×10^{-46}
23	2.44×10^{-23}	-5.54×10^{-26}	5.94×10^{-46}
24	2.44×10^{-23}	-1.52×10^{-26}	5.94×10^{-46}

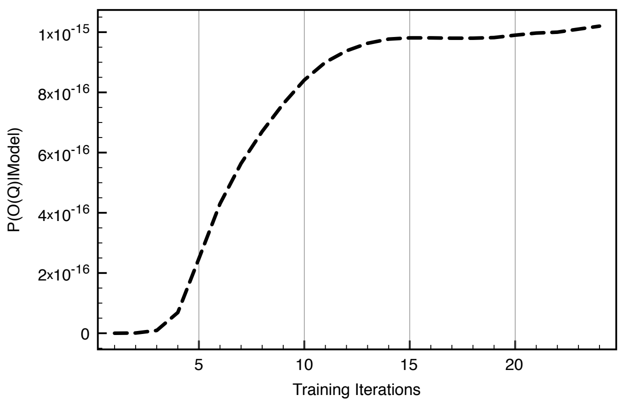


Figure A.7: Training Algorithm Convergence. Triangle.

Table A.32: Training Algorithm Convergence. Trianlge

Iteration	$P(O^{(Q)}\|Model)$	Delta	SD
1	8.35×10^{-21}	8.35×10^{-21}	2.53×10^{-43}
2	7.25×10^{-19}	7.16×10^{-19}	2.96×10^{-39}
3	9.16×10^{-18}	8.43×10^{-18}	5.61×10^{-37}
4	6.88×10^{-17}	5.97×10^{-17}	2.22×10^{-35}
5	2.48×10^{-16}	1.79×10^{-16}	1.96×10^{-34}
6	4.30×10^{-16}	1.82×10^{-16}	5.59×10^{-34}
7	5.64×10^{-16}	1.33×10^{-16}	1.00×10^{-33}
8	6.70×10^{-16}	1.07×10^{-16}	1.50×10^{-33}
9	7.62×10^{-16}	9.19×10^{-17}	2.06×10^{-33}
10	8.41×10^{-16}	7.91×10^{-17}	2.65×10^{-33}
11	9.00×10^{-16}	5.91×10^{-17}	3.16×10^{-33}
12	9.38×10^{-16}	3.82×10^{-17}	3.56×10^{-33}
13	9.63×10^{-16}	2.49×10^{-17}	3.85×10^{-33}
14	9.77×10^{-16}	1.32×10^{-17}	4.03×10^{-33}
15	9.81×10^{-16}	4.00×10^{-18}	4.13×10^{-33}
16	9.81×10^{-16}	6.97×10^{-19}	4.19×10^{-33}
17	9.80×10^{-16}	-9.29×10^{-19}	4.23×10^{-33}
18	9.80×10^{-16}	-5.02×10^{-19}	4.27×10^{-33}
19	9.82×10^{-16}	2.30×10^{-18}	4.33×10^{-33}
20	9.90×10^{-16}	7.60×10^{-18}	4.45×10^{-33}
21	9.97×10^{-16}	1.55×10^{-17}	4.66×10^{-33}
22	1.00×10^{-15}	2.35×10^{-17}	5.02×10^{-33}
23	1.01×10^{-15}	3.14×10^{-17}	5.56×10^{-33}
24	1.02×10^{-15}	3.94×10^{-17}	6.32×10^{-33}

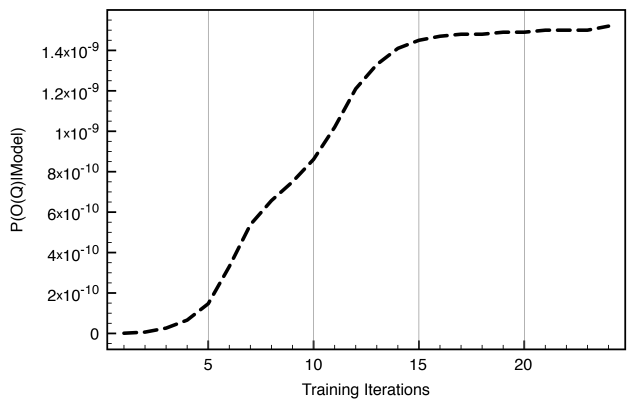


Figure A.8: Training Algorithm Convergence. Z

Table A.33: Training Algorithm Convergence. Z

Iteration	$P(O^{(Q)}\|Model)$	Delta	SD
1	1.02×10^{-12}	1.02×10^{-12}	3.19×10^{-25}
2	6.57×10^{-12}	5.55×10^{-12}	1.36×10^{-23}
3	2.63×10^{-11}	1.97×10^{-11}	2.15×10^{-22}
4	6.59×10^{-11}	3.96×10^{-11}	1.32×10^{-21}
5	1.47×10^{-10}	8.09×10^{-11}	6.21×10^{-21}
6	3.31×10^{-10}	1.84×10^{-10}	3.16×10^{-20}
7	5.38×10^{-10}	2.08×10^{-10}	8.75×10^{-20}
8	6.57×10^{-10}	1.19×10^{-10}	1.34×10^{-19}
9	7.50×10^{-10}	9.31×10^{-11}	1.77×10^{-19}
10	8.61×10^{-10}	1.11×10^{-10}	2.36×10^{-19}
11	1.02×10^{-9}	1.60×10^{-10}	3.38×10^{-19}
12	1.21×10^{-9}	1.84×10^{-10}	4.78×10^{-19}
13	1.33×10^{-9}	1.30×10^{-10}	5.95×10^{-19}
14	1.41×10^{-9}	7.41×10^{-11}	6.68×10^{-19}
15	1.45×10^{-9}	3.85×10^{-11}	7.07×10^{-19}
16	1.47×10^{-9}	1.77×10^{-11}	7.25×10^{-19}
17	1.48×10^{-9}	1.11×10^{-11}	7.36×10^{-19}
18	1.48×10^{-9}	5.67×10^{-12}	7.42×10^{-19}
19	1.49×10^{-9}	3.89×10^{-12}	7.46×10^{-19}
20	1.49×10^{-9}	7.11×10^{-12}	7.53×10^{-19}
21	1.50×10^{-9}	4.64×10^{-12}	7.58×10^{-19}
22	1.50×10^{-9}	1.89×10^{-12}	7.60×10^{-19}
23	1.50×10^{-9}	1.54×10^{-12}	7.61×10^{-19}
24	1.52×10^{-9}	1.47×10^{-11}	7.77×10^{-19}

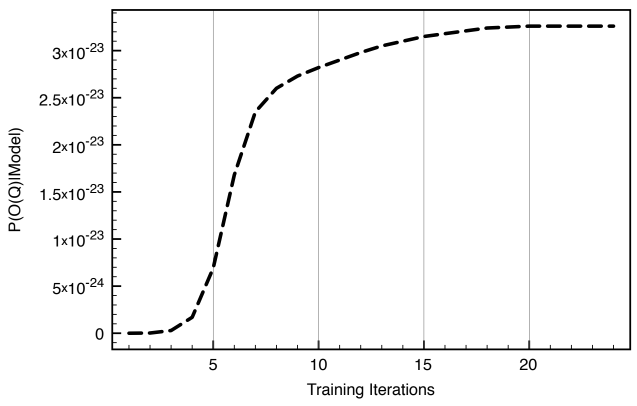


Figure A.9: Training Algorithm Convergence. Bowling

Table A.34: Training Algorithm Convergence. Bowling

Iteration	$P(O^{(Q)}\|Model)$	Delta	SD
1	5.60×10^{-29}	5.60×10^{-29}	1.64×10^{-57}
2	2.16×10^{-26}	2.15×10^{-26}	2.58×10^{-52}
3	2.86×10^{-25}	2.65×10^{-25}	4.57×10^{-50}
4	1.69×10^{-24}	1.40×10^{-24}	1.58×10^{-48}
5	6.92×10^{-24}	5.23×10^{-24}	2.63×10^{-47}
6	1.68×10^{-23}	9.91×10^{-24}	1.54×10^{-46}
7	2.35×10^{-23}	6.71×10^{-24}	3.00×10^{-46}
8	2.60×10^{-23}	2.49×10^{-24}	3.66×10^{-46}
9	2.73×10^{-23}	1.27×10^{-24}	4.02×10^{-46}
10	2.82×10^{-23}	8.89×10^{-25}	4.27×10^{-46}
11	2.90×10^{-23}	8.26×10^{-25}	4.51×10^{-46}
12	2.98×10^{-23}	7.81×10^{-25}	4.73×10^{-46}
13	3.05×10^{-23}	6.91×10^{-25}	4.92×10^{-46}
14	3.10×10^{-23}	5.57×10^{-25}	5.05×10^{-46}
15	3.15×10^{-23}	4.45×10^{-25}	5.15×10^{-46}
16	3.18×10^{-23}	3.48×10^{-25}	5.22×10^{-46}
17	3.21×10^{-23}	3.11×10^{-25}	5.29×10^{-46}
18	3.24×10^{-23}	2.33×10^{-25}	5.35×10^{-46}
19	3.25×10^{-23}	1.41×10^{-25}	5.38×10^{-46}
20	3.26×10^{-23}	7.57×10^{-26}	5.40×10^{-46}
21	3.26×10^{-23}	4.35×10^{-26}	5.42×10^{-46}
22	3.26×10^{-23}	-1.63×10^{-26}	5.41×10^{-46}
23	3.26×10^{-23}	-3.71×10^{-27}	5.41×10^{-46}
24	3.26×10^{-23}	-2.32×10^{-26}	5.40×10^{-46}

A.4 Recognition Results of the Final System

Table A.35: Recognition Results for 1 Training Sequence

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	18		2	20	90%
Square	5	11	4	20	25%
Triangle	0	1	19	20	0%
Z	4	5	11	20	20%
Bowling	0	0	20	20	0%
Average					27%

Table A.36: Recognition Results for 2 Training Sequences

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	18	0	2	20	90%
Square	1	6	13	20	5%
Triangle	6	6	8	20	30%
Z	7	7	6	20	35%
Bowling	0	0	20	20	0%
Average					32%

Table A.37: Recognition Results for 4 Training Sequences

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	14	0	6	20	70%
Square	14	0	6	20	70%
Triangle	20	0	0	20	100%
Z	0	0	20	20	0%
Bowling	0	0	20	20	0%
Average					48%

Table A.38: Recognition Results for 6 Training Sequences

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	17	0	3	20	85%
Square	18	0	2	20	90%
Triangle	14	0	6	20	70%
Z	15	0	5	20	75%
Bowling	1	0	19	20	5%
Average					65%

Table A.39: Recognition Results for 8 Training Sequences

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	20	0	0	20	100%
Square	18	0	2	20	90%
Triangle	19	0	1	20	95%
Z	17	0	3	20	85%
Bowling	4	0	16	20	20%
Average					78%

Table A.40: Recognition Results for 10 Training Sequences

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	19	0	1	20	95%
Square	18	0	2	20	90%
Triangle	19	0	1	20	95%
Z	19	0	1	20	95%
Bowling	17	0	3	20	85%
Average					92%

Table A.41: Recognition Results for 12 Training Sequences

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	19	0	1	20	95%
Square	12	7	1	20	60%
Triangle	15	5	0	20	75%
Z	18	1	1	20	90%
Bowling	14	0	6	20	70%
Average					78%

Table A.42: Recognition Results for 14 Training Sequences

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	20	0	0	20	100%
Square	14	2	4	20	70%
Triangle	14	5	1	20	70%
Z	19	0	1	20	95%
Bowling	16	1	3	20	80%
Average					83%

Table A.43: Recognition Results for 16 Training Sequences

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	20	0	0	20	100%
Square	17	0	3	20	85%
Triangle	14	0	6	20	70%
Z	20	0	0	20	100%
Bowling	17	0	3	20	85%
Average					88%

Table A.44: Recognition Results for 18 Training Sequences

Gesture	Hits	Misclassified	Unrecognised	Total Trials	Hits Percent
Circle	19	0	1	20	95%
Square	14	5	1	20	70%
Triangle	14	2	4	20	70%
Z	19	0	1	20	95%
Bowling	16	1	3	20	80%
Average					82%

Appendix B

Source Code

The source code of the gesture recognition library ABGLib and the demonstration iPhone program ABGDemo can be found on the enclosed CD.