Motion Capture to MIDI

by Joe Starling

0920088

Supervisor:

Prof. D. Marshall

Moderator:

Dr. Y. Lai

School of Computer Science and Informatics, Cardiff University

2013

Abstract

This aim of the project is to successfully recognise gestures performed in 3D space with motion sensors and to then output a sound or event using MIDI, thus creating a type of virtual instrument. This was achieved using Hidden Markov Models and related algorithms such as the Forward-Backward and Baum-Welch. Appropriate experiments were carried out on variables to gain optimal performances, such as the number of states used for a model. Gestures are being correctly recognised after they are initially defined and trained by the user, and the program also offers a choice of MIDI instruments. This report is an extension of the related Interim Report, and the work and research previously carried out.

Acknowledgements

A special thanks to my supervisor David Marshall for his consistent and invaluable support throughout the project.

1. Introduction

The aim of this third year undergraduate project is to perform physical gestures and to then recognise and use those gestures to control sound; specifically, MIDI synthesisers. By learning multiple gestures and applying distinct sounds, notes, or effects to each one, a kind of “virtual instrument” can be created, where no physical medium exists other than the computer processing the data, and at present, a hand-held sensor.

1.1 Approach

The work done during the initial report was largely research based, as it was essential to gain a good understanding of the key technologies and methods such as Hidden Markov Models (HMMs) and the motion capture device itself. Since then the development of the project has been almost entirely practical. Programming and finding the best ways to implement the chosen methods and testing to see whether they were producing correct results have been the priority. Once the program was working successfully it was necessary to experiment with certain values to ensure optimal performance and correct recognition. This will all be described in detail later on.

The scope of the project is not large enough to produce a finished 'product', but it is assumed that the end result will be a fully working system which proves that the concept is reasonable and that the methods are adequate. It will then be in a good position for future expansion.

Upon completion the program will have a simple user interface that is linear and easy to understand, and displayed on the console. I intend for the user to be able to name and record their own gesture of varying complexity, and to assign each gesture its own instrument sound (selecting a choice from a provided shortlist). The gesture(s) can then be performed at the press of a button, and the program will recognise it and play the accompanying sound.

This report will discuss the key factors in more detail, namely HMMs, MIDI, and the MotionStar sensor, as well as describe the functionality of essential aspects of the code itself and justify some of the decisions made using appropriate experiments. It will also

include a detailed and critical evaluation of the performance of the system and the approach taken to complete it.

2. Background

The technology and theory involved in completing this project have many other uses other than the goals of this project and can be applied to a variety of interesting fields such as voice recognition [1], medical rehabilitation [2], and various other ways of generating music (hand drawn gestures on iPads [3], Oral recognition[4]). These and other applications were covered in detail in the initial report.

One of the major selling points for these products is ease of use – they require very little knowledge of how the systems or synthesisers themselves operate and can very easily produce the desired result, which is entirely different from learning to play a real piano, for example. Musical gesture recognition systems could also be designed to trigger certain musical events, sound effects, pitch bends or countless other things just by making a simple hand gesture.

There is growing global demand for technologies such as these, as we have seen with the popularity of the Kinect (Microsoft) and Wii (Nintendo) technologies for gaming purposes, and I believe there is a potential gap in the market for a complete and commercialised product similar to the one in this project for making music. A product such as this would be extremely versatile, allowing people who would not normally be able to create music (mentally impaired, physically disabled) to do so, but it could also be extended to be a useful and exciting tool for a professional performing musician. It is predicted that gesture recognition and motion capture technology will soon be major factors in everyday life, and that by 2018 the largest companies in the industry could increase their worth by up to $7.15 Billion [5].

In the following sub sections I will be discussing the most important aspects of this project; introducing the motion capture hardware and system design, as well as essential algorithms, Java packages, and mathematical models.

2.1 MotionStar

The MotionStar system is the device used to detect the location of the sensor(s). It is capable of handling up to 20 such sensors, but for the purpose of this project I will use only 1 for simplicity. For more advanced limb gestures in the future an additional 1 or 2 sensors could easily be handled by the program by making some adjustments. They are designed to be attached to clothing/objects (a relevant example would be a drum stick) to track movement, but the fact that they are wired makes this somewhat cumbersome and can limit mobility, so they will be hand-held during this project.

The device includes two Extended Range Transmitters (ERTs): 12 x 12 inch cubes placed on stands which generate a magnetic field in order for the hand held sensor, or 'bird', to relay its position to the Extended Range Controller (ERC), the computer controlled power amplifier connected to the ERTs [6]. Ideally the ERTs should be positioned away from metallic objects, the sensors should not be attached to metallic objects, and any CRT screens should be at least 4 feet away, as the readings and the screens may be disrupted by the magnetic field. The position of the sensor (x, y, z, angles) is shown on an attached monitor. In order to communicate with the system and control its operation the host computer sends specific byte messages to which the MotionStar system responds with a corresponding message of success, failure, or a relevant error. Examples are shown in the table below of sent messages and expected responses, expressed as RSP (response from server) or MSG (message).

NAME	VALUE	Data Field	DESCRIPTION
MSG_WAKE_UP	10	N	This is the first command that a client sends to the server. This command is used to synchronize communication between client and server.
RSP_WAKE_UP	20	N	This is the response to the Wake Up call.
MSG_SHUT_DOWN	11	N	This message will terminate communication between the client and server.
RSP_SHUT_DOWN	21	N	This is the response to the Shut Down command

Figure 2.1 – Simple examples of messages sent and received to and from the MotionStar device. [6]

The MotionStar device is instructed to continuously send data packets, which include the position of the sensor and other data, and it does this via the local network. My program will then retrieve these packets from the datastream. Figure 2.2 shows how my system is set up, however my system has an additional ERT, or 'XMTR' as shown in the diagram. It also demonstrates where the optional additional sensors are attached.

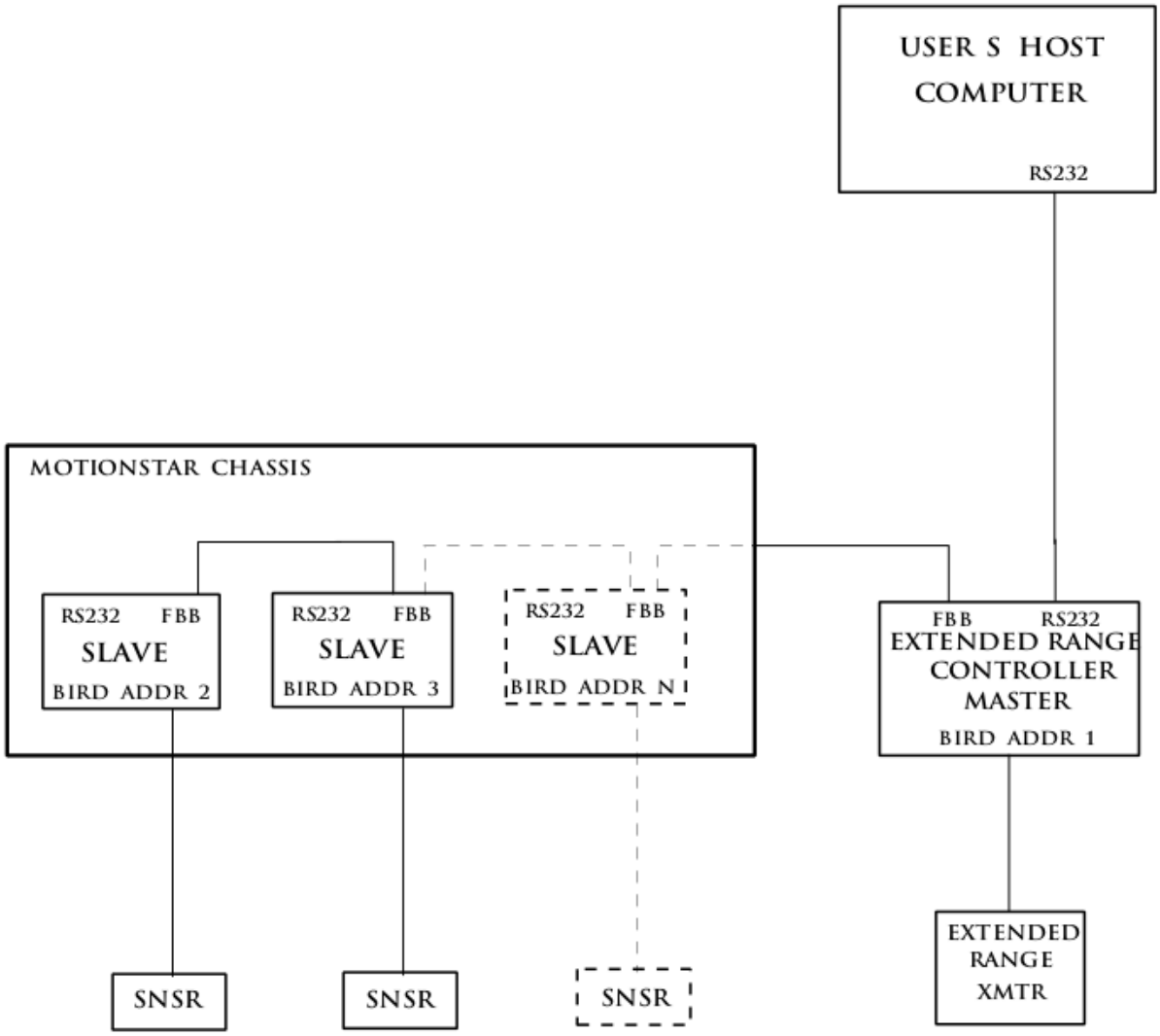

Figure 2.2 – Demonstrates the setup of the system. Host Computer/ERC communicate via the network. [6]

The sensor is attached to the main chassis of the system, which is in turn connected to the ERC. The ERC supplies and controls the ERTs with the necessary current required to create the magnetic field. As the position of the sensor is calculated within this magnetic field, its relative coordinates are displayed on a monitor for convenience, not shown in this diagram. The host computer, in this case my laptop, communicates with the ERC by

sending and receiving data over the small local network.

2.2 Hidden Markov Models (HMMs)

The main problem associated with this project is the automatic recognition of gestures. To achieve this, Hidden Markov Models (HMM) were the chosen method, as discussed during the interim report.

HMMs were chosen because a human action, such as a gesture, is inherently stochastic i.e. if a person, or a number of people, repeat the same gesture and the movement of that gesture can be measured accurately, then each measurement will be different. It is almost impossible, for example, to perform a free-hand gesture multiple times and to have the exact same measurements (location, movement, velocity, angle) each time. The measurements will be different, but they represent the same gesture. Figure 2.3 demonstrates a possible example of a circular gesture being repeated and recorded 10 times and how they obviously represent the same gesture, but each time, the measurements are different [7].

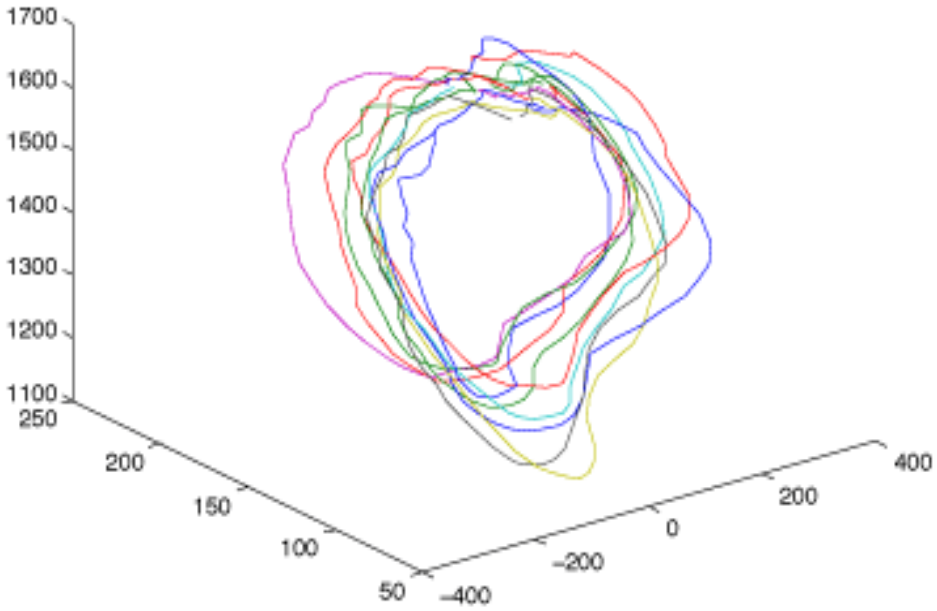

Figure 2.3 – Each coloured line represents a separate attempt at performing a circular gesture. [7]

HMMs were ideal for use within speech recognition for similar reasons, that is, the same word could be observed, but from an entirely different voice. This implies there are certain characteristics or properties to these words or gestures which are the same each time, despite being technically different [8]. The HMM overcomes this problem and essentially learns to recognise these characteristics.

In a regular Markov chain, the states are also the observables and the transition probabilities from one state to the other are the only parameters. However the idea behind a HMM is that we have an underlying process which is hidden from observation, and an observable process which is determined by the underlying process [9]. A HMM is a finite set of states, where each state has two parameters: a transition probability, and an output probability distribution. In a particular state and outcome, or observation can be generated according to the associated probability distribution. The output dependent on the state is visible, but the state itself is not [10], hence it is hidden.

A HMM can be formally defined as follows [8]:

The number of states in the model, N

The number of observation symbols in the alphabet, M

The transition probability matrix $A=\{a_{ij}\}$ where a_{ij} is the transition probability of taking the transition from state *i* to state *j,*

$$a_{ij}=p\{q_t+1|q_t=i\}, \textit{where } q_t \textit{ denotes the current state}$$

A should satisfy

$$a_{ij}\geq 0, \quad 1\leqslant i, \quad j\leq N$$

and (as it is a probabilistic function)

$$\sum_j a_{ij}=1$$

The output probability matrix $B=\{b_j(O_k)\}$ where O_k is the discrete observation symbol,

$$b_j(O_k)=p\{o_t=v_k|q_t=j\}$$

where v_k denotes the k^{th} observation symbol, and o_t is the current parameter vector.

B should satisfy

$$b_j(O_k) \geqslant 0, \quad 1 \leqslant j \leqslant N, \quad 1 \leqslant k \leqslant M$$

and

$$\sum_k b_j(O_k) = 1$$

If the initial state distribution is $\pi = \{\pi_i\}$ where $\pi_i = p\{q_1 = i\}$ we can write a HMM compactly as

$$\lambda = (A, B, \pi)$$

Figure 2.4 illustrates how the model works with 5 states. For this project, a state would represent the sensor occupying a specific area in space, so this would potentially be an example of a circular gesture.

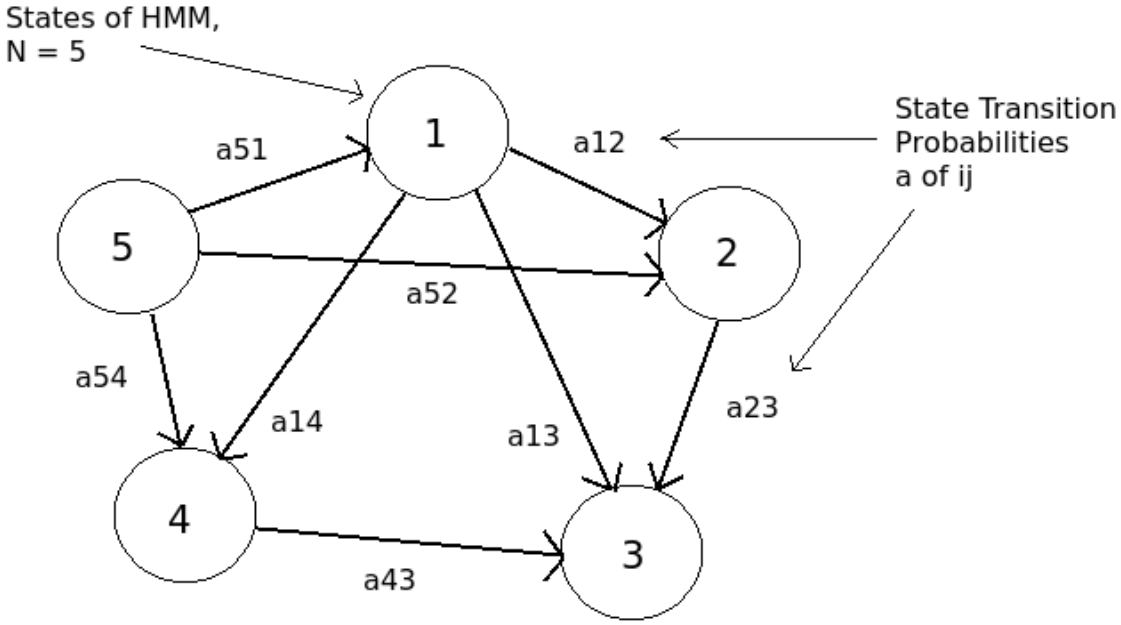

Figure 2.4 – Graphical illustration of a HMM . [11]

HMMs are associated with three 'problems', and each one has its own method of solving [1].

- Problem 1 – Given the observation sequence *O*, how do we efficiently compute $P(O|\lambda)$, the probability of the observation sequence, given the model?
- Problem 2 – Given the observation sequence *O*, how do we choose a corresponding state sequence *Q* which best explains the observations? (i.e. uncover the hidden part of the model)
- Problem 3 - How do we adjust the parameters of λ to maximise $P(O|\lambda)$?

I am most interested in the solutions to problem 1, which allows us to calculate the probability that an observation sequence 'belongs' to a particular HMM (i.e. perform a gesture and calculate the probability that it is the same gesture associated with a particular HMM), and problem 3, which allows us to 'train' a HMM for greater accuracy.

For problem 1, the most obvious way to solve it is by evaluating every possible state sequence combination of length T (the number of observations), however even for a small example with 5 states and 100 observations, this would result in ~10^{72} computations, which is obviously unreasonable. Rabiner [1] describes the much more efficient method, the Forward-Backward calculator, a Dynamic Programming algorithm which computes a combined 'smoothed' probability after completing 'forward' and 'backward' time passes. Strictly speaking, only the forward part of the algorithm is really used to solve the problem.

The forward variable $\alpha_{(t)}(i)$ can be defined as :

$$\alpha_{(t)}(i) = P(O_{(1)}, O_{(2)} \ldots O_{(t)}, q_{(t)} = S_{(i)}|\lambda)$$

The probability of the partial sequence of observations until time *t* and the state S_i at time *t* given the model λ. By computing this for all possible states and observations and iterating for incremental time *t* until terminal time *T,* the probability of a sequence given a model can be written as:

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

The backward variable, β, is computed similarly. For a more detailed description refer to [1, pages 262 - 263]. This will be the method I will use later in this project to judge whether a performed gesture matches a pre-existing gesture.

The solution to the 3rd problem is far more complex and comes in the form of the Baum-Welch algorithm. There is no optimal way of estimating the HMM parameters, but the Baum-Welch algorithm chooses parameters that provide a locally maximised $P(O|\lambda)$ [1]. By doing numerous iterations of the algorithm, likely transitions are reinforced until a local maximum value is reached for the transition and emission probabilities.

2.3 HMMs in Java

Clearly, calculating HMMs, modifying them, and evaluating recognition probabilities is a complex task, and would be incredibly time consuming to program from scratch. To aid the project I will be using an open source Java library called Jahmm [12], which has been specifically designed to create and evaluate HMMs and also implements various interesting and useful algorithms, including the Forward-Backward and Baum-Welch mentioned previously. The library was created in 2009 and last updated in 2011, however the theory involved is all correct and works well. There are very few relevant examples available, which contributed to the amount of time taken to understand the implementation, along with the general complexity of HMMs, as mentioned in the earlier report.

There are other limited implementations of HMMs in Java, however they are currently unavailable, or provide limited functionality – Jahmm was the best choice due to its exhaustive list of useful classes and methods, despite the lack of detailed instructions.

2.4 MIDI

The eventual output of the program will be a MIDI sound and the basics were covered in the initial report [13]. The main goal within this project was to simply output a note, or series of notes, using 'Note On' 'Note Off' commands. A more detailed description of MIDI and its capabilities can be found at [14].

3. Design

3.1 System Specification

As we know, the main aim of the project is to recognise human gestures and then to output a MIDI note. Gestures should be user-defined (name of the gesture, movement, and the resulting instrument sound) in order to give the user a greater degree of control over what is essentially a virtual instrument. For now there will be limitations such as the number of gestures the program can manage at a time, and the number of different instruments the user can choose from. These limitations are currently self enforced simply to reduce the overheads of this experimental program. For a diagram of the overall system refer to Figure 2.2, but in addition the photo below demonstrates the typical set up of the system as seen in the lab. The two ERTs can be clearly seen, as well as the main chassis and ERC. The wired sensor is also shown.

Figure 3.1 – The MotionStar hardware set up

Figure 3.2 – The sensor

The User Interface will be a simple console based output for the purposes of the project, sometimes requesting information to be typed as well as confirmation of events (yes/no options, prompts to begin recording data etc). The output will be informative and in real time, while also being very clear and easy to understand.

There are two main classes to the program, Record.java, and Recognise.java, each handling their respective tasks. Both of these classes utilise another class, MotionStar.java, which makes contacting the MotionStar system easier. MotionStar.java was written by Stephen Lawrence of Cardiff University, and only modified slightly by myself.

In order for the project to succeed it must accomplish the functional requirements. Table 3.1 gives an overview.

Functional Requirement	Action	How To Test
Save multiple gestures and HMMs	Generate and save multiple HMMs to file	Attempt to create more than one gesture
Recognise gestures correctly	Use appropriate algorithm to compare observations to HMMs	Purposefully perform incorrect gestures and observe results
Play MIDI sounds	Use Java's sound package to create MIDI	Create a test program
Play a sound on successful recognition	Couple recognition and MIDI playback	Perform correct gesture and observe result
Clear and concise console output	Only display important information	Ask peers to test the program
Simple to use	Clear and linear requests and commands	Ask peers to test the program
Recognise incorrect console input	Have appropriate error handling for illegal arguments	Purposefully input incorrect arguments

Table 3.1 – Functional Requirements

In addition, there are a few Optional Requirements which would be implemented if there is enough time after completing the Functional requirements. They are realistically achievable, but not essential within this project.

Optional Requirement	Action	How To Test
Recognise gestures continuously	Modify program to no longer need a prompt to perform a gesture	Run the program
Record musical sequences resulting from recognitions	Improve Java MIDI functionality	Save and load MIDI outputs
Alter live/constant playback of a song with MIDI effects	Play an imported song and alter the output with gesture recognition	Observe changes to the song

Table 3.2 – Optional Requirements

Recording Gestures

This class handles the saving of gestures, including a name or description of the gesture; the intended output instrument sound; and the movement itself, by training HMMs. A single HMM is generated for each direction of movement (x, y, z). The dedicated HMMs for each plane were created to hopefully increase the accuracy of recognition – the idea being that the program would recognise more detailed information, which would allow two or more similar, but distinct gestures to be recognised successfully. In order to train a HMM accurately, especially for a more complex gesture, a large amount of data is required and therefore a gesture must be performed multiple times, as similarly as possible. The program currently designates 10 recordings of a gesture movement before training, which may be necessary for some gestures, but it could equally be an unnecessary number of performances for a much simpler gesture which would be easier to learn. This and other restraints/variables will be discussed in more detail later.

Once a set of training data is recorded, it is clustered, and the best possible HMM is generated. Jahmm provides a useful functionality, 'HMMWriter', that allows the saving of HMMs to text files in a specific format which can then be read later on by a 'HMMReader'. A resulting saved gesture then has three small .txt files associated with it, as well as a designated line in another two files, saving its name and instrument sound.

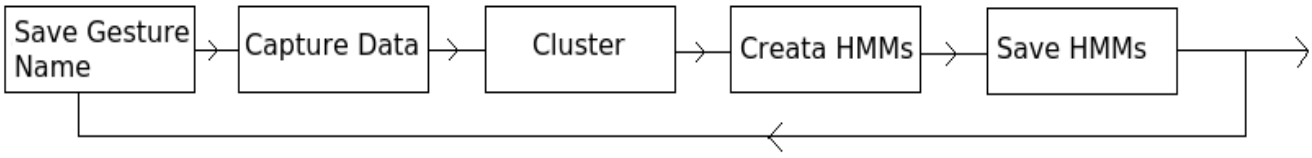

Figure 3.1 – Sequence of events for recording a gesture

Once a gesture has been trained and saved, the user can move onto the recognition stage, or begin the process again and train another gesture. Currently there can be up to 5 gestures saved due to restrictions in the Recognise program. This number was chosen arbitrarily and could easily be increased/reduced, however the implications of a large number of saved gestures are unclear, and the potential effects on successful recognition

rates will be discussed later.

Within this program there is appropriate and necessary error handling, especially concerning the user-defined instrument. The user must type the instrument name, but it must be a valid instrument for the Recognise class to be able to load. Similar handling exists at the end where the program asks the user a yes or no question (whether or not to create another gesture).

Recognising Gestures

The next step is recognising a saved gesture and playing the sound/instrument associated with it. To do this, the Recognise class records a gesture performance and compares its movement with all of the pre-existing HMMs, and either outputs a failure, or a successful match if the probability is high enough. The program currently prompts the user to perform 10 gestures one at a time, and also evaluates them one at a time. After the tenth gesture, the program will exit. To get the best results from this part of the program, various calculation results were displayed on to the console to judge how closely a gesture matched its HMMs according to the algorithms, and whether or not the gesture detected was indeed the one performed. For example, in the beginning it was difficult to know what number to set the default probability to, so I was able to compare the calculated probabilities and gauge roughly what to set it as, depending on the number returned for an actual gesture and the number returned for a random movement. Once a gesture is recognised correctly, the instrument name is read from the list file, and the MIDI channel is assigned the relevant instrument before being given the NoteOn command. The default output note is Middle C, however for the Bass and Synth instruments the note is lowered to better represent the instrument and get the best sounding output.

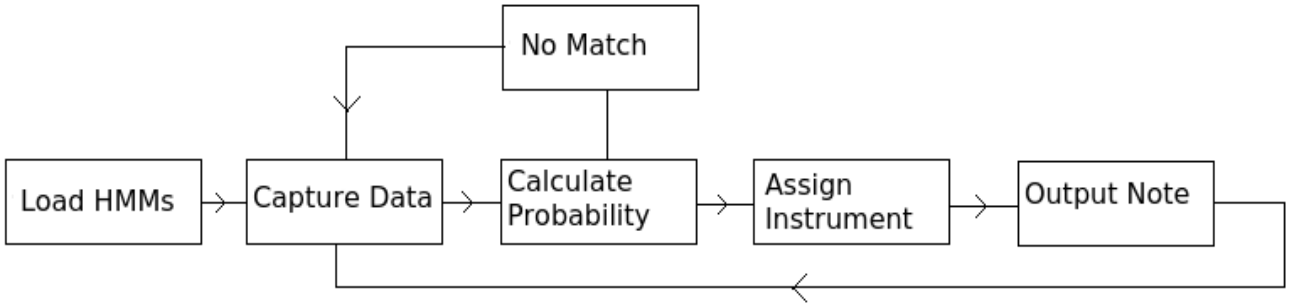

Figure 3.2 – Sequence of events for recognising a gesture

To communicate with the MotionStar sensor both of these classes use another class, MotionStar.java, in combination with some additional code they contain themselves. This has been done in the least obtrusive manner possible as the end user does not need to know about the details of the communication, but some simple messages are displayed on the console to inform the user when the program starts and stops collecting data. This is to ensure the user is not performing the desired gesture before or after the program is actually collecting, thus effecting the HMM and subsequent chances of recognition.

4. Implementation

This entire project has been completed in Java, produced using the Eclipse IDE. As the project was intended to be experimental and more of a proof of concept, there is no formal Javadoc included, but some comments are present. The Jahmm packages were used throughout the development. In this section I will describe the implementation of each step of the system in more detail.

4.1 Creating a gesture

4.1.1 Naming And Saving

The first step in creating a gesture within the program is to give it an appropriate name either relevant to the the motion or the sound. Relevant names help to distinguish gestures, however this is entirely the user's choice - they are presented with a blank input field. Selecting an instrument sound is similar, however one of a possible five instruments must be typed correctly, and no other input will be accepted.

```
Succesfully connected to MotionStar
Welcome!
You may record up to 5 different gestures.
Please state the name of the gesture you wish to create: Circle

Choose an instrument from the following
1 - Guitar
2 - Piano
3 - Bass
4 - Synth
5 - Organ
Instrument: Guitar
```

Figure 4.1 – Console display when the program is started. Green text has been input by the user.

The gesture name and instrument preference are saved to .txt files with line breaks between each one and are later read and evaluated during the recognition stage, achieved using basic FileWriter and FileReader objects. The line breaks simplify the task of counting/numbering gestures.

Once this initial work is done, we move onto the next step, which is training the HMMs.

4.1.2 Retrieving Data

To create a HMM the positional data must be captured over the local network. Ten repetitions of the same gesture are completed, as prompted by the console. User interaction is required as each time the capturing stops, a System.in.read(); method waits for the enter button to be pressed before it continues. This allows the user to return to the starting point of the gesture without contaminating the data and gives them more control. The program only captures for a set amount of time so it is important to complete the required gesture before it stops reading data points. I experimented with various thread pauses while capturing the 100 data points, and found that a Thread.sleep(50); within the loop allowed the ideal time for most simple gestures to be performed in full, without being too detrimental to the accuracy of the readings.

There are in fact 101 iterations of the capture loop, however the first data point is never captured. This is because the first packet sent from the sensor is always zero, so it is ignored using a simple Boolean.

```
setSocket();
sendMessage(MSG_RUN_CONTINUOUS);
System.out.println("Capturing...");

for (int i=0 ; i<101 ; i++){
        receiveMessage();

        if (firstElement == true){
                firstElement = false;
                }
```

Figure 4.2 – Part of the runContinuous() method.

Figure 4.2 shows part of the code used in order to collect data points - s*etSocket*(); is associated with the MotionStar communication and establishes a Datagram Socket, while sendMessage(); then sends a direct command to the MotionStar device ordering it to broadcast a continuous stream of data points. Right at the beginning of the program there is also a command used to turn on and connect to the MotionStar device, which uses the MotionStar class. The majority of the communication aspects, including most of the MotionStar class, had been previously established through other projects and other uses

of the device, and have only been slightly modified by myself [15].

Once the observations are captured they are placed into a List object, and when the next training performance is captured, it is also added to the same list to create a list of lists which is compatible with Jahmm to create HMMs.

4.1.3 Creating And Optimising HMMs

Once a complete list of observations is acquired, I can use Jahmm's classes to generate and save optimal HMMs based on the positional data of the sensor. Firstly the *KmeansLearner* generates a HMM with a specified number of states based on calculations using the k-means algorithm. This was chosen to be 8 states after some testing and evaluating how this number could be optimised to improve performance; for example, to better handle complex gestures. The effects of changing the number of states will be discussed later.

```
factory = new OpdfIntegerFactory(100);
kml = new KMeansLearner<ObservationInteger>(8,factory,toHmm);
Hmm<ObservationInteger>hmm = kml.iterate();
```

Figure 4.2 – Creating a HMM using KMeansLearner

The KMeansLearner has three parameters; the number of states the resulting HMM will have, an *OpdfFactory*, and a vector of observation sequences that the HMM will be based on. As I will be dealing with integers, there is a dedicated *OpdfIntegerFactory* which can be used to generate a new observation probability distribution function (OPDF) based on the 100 observations of each training sequence. The observations themselves are held in *toHmm.* Calling kml.iterate(); returns a newly improved HMM with better cluster estimations.

By only using one iteration as opposed to the 'learn' method (which keeps iterating until a fixed point it reached i.e. the HMM no longer changes), it returns a better starting point to be used for the Baum-Welch algorithm as the HMM does not depend on the temporal dependence of observations [16].

```
BaumWelchLearner bwl = new BaumWelchLearner();
bwl.setNbIterations(20);
bwl.learn(hmm, toHmm);
return hmm;
```

Figure 4.3 – Baum-Welch iterations on the HMM

The Baum-Welch algorithm is used to further optimise the HMM's parameters. The HMM returned in figure 4.3 has had 20 iterations of the algorithm performed on it. It uses the existing HMM and the same observation sequence as before, and outputs the HMM that best matches the set of sequences given, according to the Baum-Welch algorithm. In the Evaluation section I will discuss how I decided on the number of Baum-Welch iterations.

This happens three times for each individual gesture, with a separate HMM for each direction of movement. This is a relatively untidy method but it was the quickest way to expand upon my experimental code, and due to time constraints and the fact that it does not affect the system drastically, I decided to continue in this fashion. The experimental code produced in the earlier stages of the project expanded upon an idea found elsewhere of recognising gestures drawn on a computer's mouse pad [19]. The observations were simpler and not in three dimensions, and as time was pressing I decided to use the same principles in my own project.

Each HMM is saved into its own text file with a Jahmm-specific syntax using the *HmmWriter* class provided. They are labelled accordingly with the name of the gesture and an X, Y, or Z appended to each one.

As mentioned previously, once a gesture's HMMs are saved and its details added to the reference files, the user can either begin the process again, or continue to the recognition phase.

4.2 Recognising a Gesture

4.2.1 Reading saved files

The recognition process begins with the reading of the relevant files and counting the number of gestures present in the gesture list. It's useful to know how many gestures there are to be able to give a correct length to the *gestureName* and *instrumentName* arrays which will then store their respective information. As the user could have saved between one and five gestures before continuing to this process and the program begins it automatically, there is no safe way to assign a correct number beforehand.

```
File file = new File("GestureList.txt");
LineNumberReader lnr = new LineNumberReader(new FileReader(file));
lnr.skip(Long.MAX_VALUE);
noOfGestures = lnr.getLineNumber() + 1;
```

Figure 4.4 - countGesture method.

By jumping to the end of the file, which had each new gesture name added on a new line, the *LineNumberReader* simply returns which number line is the final line. A value of 1 is added to *noOfGestures* just to simplify the task of array population slightly, shown below. Jahmm's *HmmReader* loads the saved HMM text files in a very similar way, and it then creates HMM objects such as *hmm1X,* which represents the X direction HMM of the first gesture in the list, and so on.

```
FileInputStream in = new FileInputStream("GestureList.txt");
BufferedReader br = new BufferedReader(new InputStreamReader(in));
for(int i = 1; i<gestureName.length;i++ ){
      gestureName[i] = br.readLine();}
```

Figure 4.5 – Populating the gesture name array

The instrument names are also loaded in this way from the MIDI.txt file, and as they were written in the same order and the same time as the gesture names, the i^{th} index of *gestureName* corresponds to the i^{th} index of *instrumentName*, which is how I will later

assign the correct instrument to the gesture that is identified.

4.2.2 Performing and comparing

The user is prompted to perform a gesture in much the same way as during the recording phase. The *runContinuous* method is the same and it populates a list with observation integers, which is the list of observations that will be compared to the HMMs of each saved gesture. The *evaluate* and *getProbability* methods then work together to determine the gesture which best matches the ones which were loaded earlier. Figure 4.6 demonstrates the process of calculating the probability of a sequence given a HMM. This is done three times for every possible gesture, as each new sequence (x, y, z) is compared respectively against each of the gesture's three HMMs. I then calculate the sum of these probabilities. If the sum is a sensible probability function (i.e. 0 ≤ Sum ≤ 1) and above a pre-set threshold, *bestMatchProb,* then in theory a gesture has been recognised. The threshold was decided upon by evaluating a large number of performances and observing the probability outputs. By observing the sum of probabilities for a gesture I knew to be correct and comparing it to a sum for a gesture I knew to be incorrect, I could find a sensible threshold. Any gesture performance that is entirely incorrect will produce a Zero.

```
fbc = new ForwardBackwardCalculator(sequence, hmm);
probability = fbc.probability();
return probability;
```

Figure 4.6 – Calculating the probability of sequence x|y|z given HMM x|y|z

As the program progressed through each evaluation (through the switch statement working through the various triplets of HMMs) multiple versions of the same variable were created. For example, with two gestures, the program would generate two different *probabilityX;* this was observed through trial and error and resorting to printing outputs onto the console. This meant that when the probability of both gestures was above the pre-set threshold, it would assume the first gesture encountered was the correct one, and not necessarily the one with the highest probability. By resetting that threshold, *bestMatchProb,* each time it was above the initial value, it ensured the set of HMMs with

the highest probability was deemed correct. The i^{th} set of HMMs (*hmmNo*) corresponds to the i^{th} index in the *gestureList* and *instrumentList* arrays, thus identifying the successful gesture name and instrument sound.

4.2.3 MIDI Note

Once a gesture is recognised the program should play a MIDI note. The *midiOutput()* method takes an integer, *instrumentNumber,* as a parameter which has been changed to represent the desired instrument. By reading the name of the instrument from the MIDI.txt file this number is changed to represent the corresponding number from the MIDI sound bank. When defining the MIDI synthesizer and channel it is possible to change the instrument sound using the *programChange* method – each number from the possible 128 programs represents a different instrument or instrument variation. The note and volume were arbitrarily chosen, and there is a *Thread.sleep()* command which controls the length of the output before the note is turned off.

```
if(instrumentName[bestMatchNo].equalsIgnoreCase("guitar")){
                        instrumentNumber = 27;
    ...

    Synthesizer synth = MidiSystem.getSynthesizer();
    synth.open();
    MidiChannel[] channels = synth.getChannels();

    channels[0].programChange(instrumentNumber);
```

Figure 4.7 – Assigning the guitar program (instrument) to channel 0 before playing the note

Upon successful recognition a message is displayed on the console confirming which gesture has been recognised, as well as the name of the instrument associated with it. Different gestures can have the same instrument, but will also play the same note, so the console output will be the only way to tell which gesture has been identified in that case. A useful addition to the program would be to have these different gestures produce a different note if they are found to have the same instrument, however there was not enough time to implement this.

5. Results

Here I will briefly discuss the current system's output and operation during a complete run through by saving a single gesture 'lineUp', and then continuing to the gesture recognition stage. This could be considered a simple black-box test.

After being presented with the input fields shown in the previous section and saving the gesture's name and preferred instrument, the user repeats the gesture ten times, the HMMs are created and saved, and they are then given the option of creating a new gesture. An input of 'y' or 'Y' restarts the program and learns another gesture, whereas 'n' or 'N' automatically begins the recognition program.

```
Training iteration 10
Press Enter, and perform gesture 'lineUp'

Capturing...

Stopped capturing

Press Enter to continue

Gesture data succesfully saved.
Would you like to create another gesture? (Y/N)
n
```

Figure 5.1 – The end of the gesture recording stage

The gestures and HMMs are loaded, and a list of all possible gestures is displayed. Below is the result of a successful recognition, where the correct gesture was performed- a MIDI note was also played.

```
Your saved gestures are:
Gesture #[1] = lineUp

Recognition iteration 1
Press Enter, and perform a saved gesture

Capturing...

Stopped capturing

Best match is: lineUp (piano)
```

Figure 5.2 – A successful recognition

In the event of a gesture producing a probability that is too low to be considered a success, the program simply tells the user it was unsuccessful and continues to the next performance. Below we can see the result when a different gesture is performed. There is obviously no sound output.

```
Capturing...

Stopped capturing

*Unable to recognise gesture!*
Try again.
```

Figure 5.3 – Unsuccessful recognition

5.1 Evaluation

I will now test the system in various ways and observe how it responds. I intend to examine the effects of changing variables such as the number of training steps performed to create the HMMs as well as observing whether complex and simple gestures require different values to get the best results. Also, it will be interesting to judge how well the system responds to very similar gestures, or if the overall performance is affected by having a larger number of saved gestures, compared to only having one.

Firstly I will experiment with the number of training steps by reducing it to just 1 step for a simple line down gesture. By being very careful with the positioning of the sensor during the recognition phase and making a gesture as similar as possible to the one that was just used to create the HMM, numerous successful recognitions were made, however any slight change resulted in a failure. As the HMM is based on just one set of observations it cannot accommodate for variations very well even if they are small, so the performance must be almost identical. Despite this, using one test iteration is technically feasible for such a simple gesture as the success rate was 80% when the performances were precise. Precision in gesture performances would be a necessity for the creation of a full instrument such as virtual piano keys, or a drum kit, so it could be considered a positive result in that respect.

A complex gesture can be recognised with a good level of accuracy when trained with 10 or more training data sets, however with fewer than 8 sets the system does not

have a good success rate. In this context a 'complex' gesture could be defined as a gesture with two or more relatively rapid changes of direction, whereas a 'simple' gesture moves in only one direction, either x, y, or z.

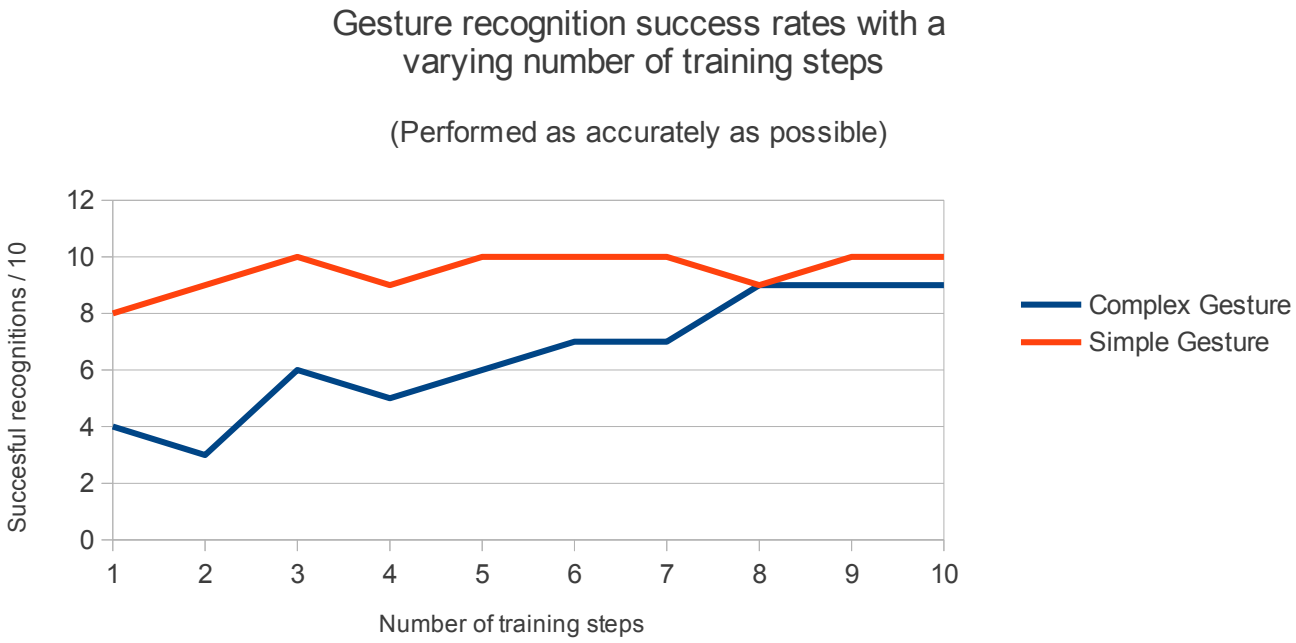

Figure 5.4 – Chart demonstrating the effect of a varying number of training data sets on two different gestures

This is an interesting result – if the user is creating a simple gesture then any more than two training steps has a limited effect on the accuracy of recognition, and could be considered a waste. However a complex gesture requires around 10 or more for it to be a useful model. Currently the best option is to leave the default number of training steps as 10 due to time constraints as it is better to overcompensate, but ideally the user would choose the number of training steps that best suits the gesture, at the start of the program. This would save a laborious task for the user in addition to being less computationally expensive.

When there are more than two simple gestures saved, successful recognition rates appear to drop and the program decides that another gesture was the one performed. When tested with five 'simple' gestures recognition was often successful, but the incorrect gesture was sometimes chosen to be the output. This suggests the system is not yet

accurate enough to handle a large number of gestures. The default maximum number of states was chosen to be 5 arbitrarily, but it could be increased. One possible reason for this is that the models themselves are in fact the same for gestures that are the opposite of one another (straight line up, straight line down). The system does not currently take into consideration the direction in which the sensor is *moving*, it only tracks the *position*; it would be useful to observe the start and end points of such gestures and be able to differentiate between them by having positive or negative vector notations associated to them. With a much larger number of saved gestures (say, 100) it is likely that similar aspects of each state, such as start and end points, will confuse the system to the point where it no longer works, meaning that overall, it does not scale well.

To overcome the problem I decided to experiment with the probability threshold. By increasing the threshold I did achieve slightly more sensible results, as the program was essentially being more strict, and would only acknowledge a recognition when the gesture was extremely similar. However this had a negative effect later on as consequentially very few gestures were being recognised when they should have been, so the threshold was changed to its original value. The problem remains, although it is more a problem of accuracy and scalability, and not functionality.

The number of states that a HMM contains must be defined explicitly as a parameter during its creation. The number of states can have a significant effect on recognition. I tested the same 'complex' gesture as used earlier (four lines in varying directions) with 10 training iterations, and created HMMs using just two states, and then with incrementally more states. With just two states recognition was 0% for this gesture - the model was too simplistic. I subsequently tested a simple gesture using the same system specifications and found that although it did recognise the gesture to be *slightly* similar to the saved one, the probabilities were below the threshold, and so *successful* recognition was also 0% for simple gestures. This result surprised me as I was expecting the system to be able to recognise a basic gesture with very few states.

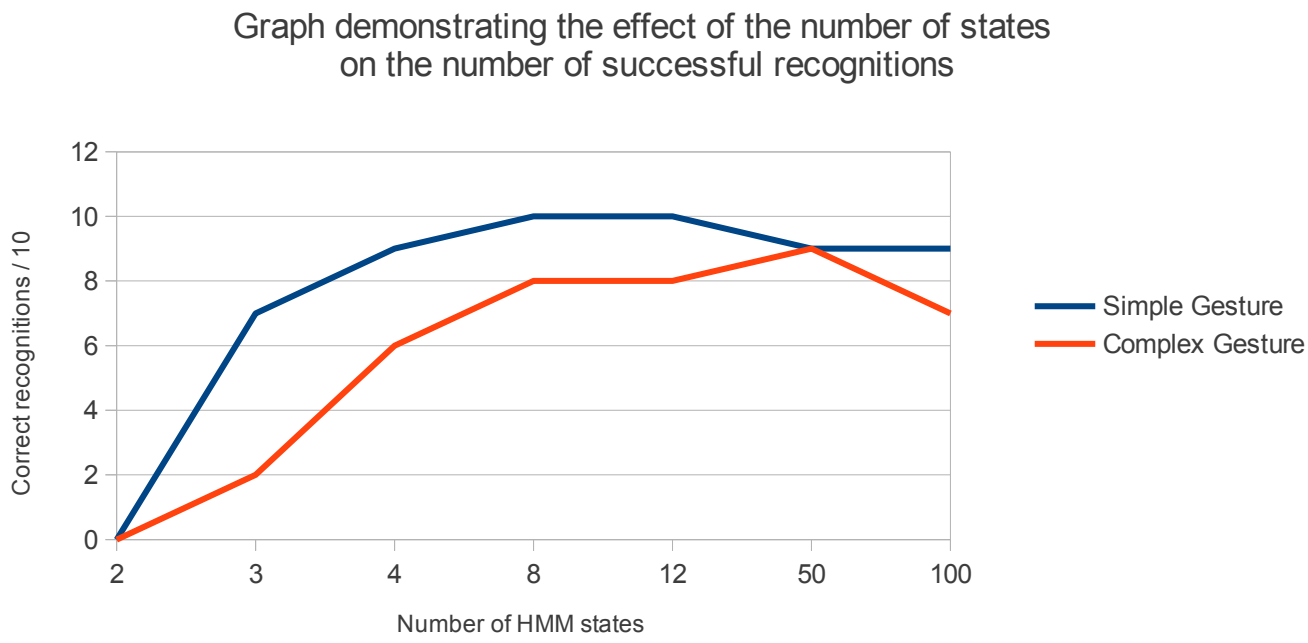

Figure 5.5 – Graph of varying number of states

By experimenting further it became clear that the absolute minimum number of states required, even for a simple gesture, was 3. By going to the other extreme and generating 100 states, the same as the number of observations, the program took an exceptional (and unacceptable) amount of time to generate and save the HMMs, however recognition rates for simple gestures were good. Complex gesture recognition was only around 70%.

These results were interesting, and it was essential to choose a default number of states that provided the best possible recognition rates regardless of gesture complexity, and did not take too much time and create files that were comparatively very large; the safest number to use based on the results was 8 states, as although the results might seem slightly better using 50, it slowed down the program considerably.

It's important to remember these experiments took place with myself performing many gestures repeatedly, and that the results could have been effected slightly by fatigue, or by learning how best to perform the gesture to gain a positive result. I think overall the results remain valid and the assumptions on optimal variables remain true.

As well as the number of states, I was also required to choose an appropriate number of times to perform the Baum-Welch algorithm. The following text shows parts of the HMM files produced (the first two states) for the y-direction of the same gesture, but each one trained with a different number of iterations.

ONE B-W ITERATION

State 1 Pi 0.2 A 0.965 0.035 0 0 0 0 0 0

State 2 Pi 0.6 A 0.025 0.972 0 0.003 0 0 0 0

TWENTY B-W ITERATIONS

State 1 Pi 0.8 A 0.974 0.023 0.003 0 0 0 0 0

State 2 Pi 0.2 A 0.022 0.978 0 0 0 0 0 0

FIFTY B-W ITERATIONS

State 1 Pi 1 A 0.996 0.002 0.002 0 0 0 0 0

State 2 Pi 0 A 0 1 0 0 0 0 0 0

ONE HUNDRED B-W ITERATIONS

State 1 Pi 1 A 0.961 0.036 0.003 0 0 0 0 0

State 2 Pi 0 A 0.076 0.924 0 0 0 0 0 0

There were a total of 8 states. 'A' is a list of state transition probabilities, and 'Pi' is the initial state probability. We can assume that for a gesture such as this downward line, the initial state probability value for the second state should not be more than the first state, such as in the first example, so one iteration is not useful. The results in the second example, using 20 iterations, are much more typical of the gesture's movement and would be an ideal model to use for recognition. With 50 and 100 iterations the results are very similar; they appear to have been overly optimised to the training data which could make recognition more difficult, as the gestures would need to be more precise. This is why I chose to use twenty iterations.

The output of the system, the MIDI note, is still a basic operation. There is one note played per recognition, using the program (instrument) that was specified by the user. For certain instruments I decided the pitch should be lowered, to achieve a more realistic sound. The note plays for 1 second, and is then turned off. This is technically what the project set out to achieve, however producing more interesting sounds or combinations of notes (chords) would be more desirable.

With regard to the user interface, I have not had the opportunity to test its usability on an inexperienced user. Ideally I would have conducted a survey amongst peers and found the most desirable format, content, and syntax, and then tested my current system and changed it accordingly. Although due to the scope of the project this was not a major concern.

Overall the system works well. Each component has been been evaluated, and on a complete run through of the program starting from scratch and creating multiple gestures, everything comes together successfully to complete the required task. The best possible HMMs are created and saved. The recognition class either starts automatically following the gesture creation, or it can be used as a standalone class to recognise gestures that were saved earlier. Purposefully incorrect gesture performances are ignored, and correct gestures are recognised with a good rate of success, though the success rate does seem to decline slightly when there are more than three saved gestures.

6. Future Work

The project is currently at a stage where it works to an acceptable standard and achieves all of the previously outlined goals, however its functionality and implementation do have room for improvement. Some features that could be accomplished in the future but were not possible due to time constraints include:

- Creating a specific instrument that uses gesture recognition, such as a virtual piano, or a drum kit
- Continuous data retrieval and recognition, which would allow much greater freedom than the present system
- Using a single HMM for each gesture, rather than one for each plane of movement
- Calculating the velocity of movement and applying it to the volume of the MIDI note. Faster movement creates a louder sound
- Save sound outputs for later access

The framework is in place in order to create a specific instrument. The same gesture, such as a downward drum stroke or piano key stroke performed in the same manner but slightly to the left or right is considered distinct by the system, so it could easily be adapted. The system would output a single instrument sound, but each gesture could represent a different note of that instrument. This would require a physical reference point for gesture performances as the gestures must match the saved one exactly; this aspect is technically a limitation to the system's performance, but adapted in this way it can be taken advantage of.

Currently gestures are only recognised when performed within a set time after a prompt and the movements captured in that time frame are then analysed. This is not practical for a virtual instrument and greatly restricts the usability. The final version of this project would collect data continuously and analyse gestures continuously without a prompt. HMMs are good at dealing with such a task. As the data is collected the program would need to evaluate each possible set of observations, so for example, after collecting

100 observations they will be compared to the HMMs, then when the 101st is collected, the 1st observation can be disregarded and the remaining 100 can be evaluated, and so on. All beginning and end points must be considered as gesture boundaries are unknown. The use of the Viterbi algorithm is recommended for such a task, it is an efficient sub-optimal search algorithm ideal for continuous recognition [17].

As mentioned previously the system currently creates three HMMs for each gesture which then need to be loaded and generated again for gesture recognition. This is inefficient especially if there are fewer than five saved gestures as there are then unused variables, so there should be a way of normalising data points within a certain 3D space, or sphere, and then using that as an observation for training a single HMM that represents the entire gesture.

MIDI notes are currently set at an arbitrary output volume, but this is not representative of a real instrument. The speed of movement has a profound effect on the sound and volume of an instrument, so the velocity of movement based on a simple Speed = Distance/Time calculation using the start and end positions of a gesture could be utilised to change the output volume accordingly. This would greatly improve the illusion of a real instrument.

Once the continuous recognition is working it would be a useful addition, and an extension of the main ideas of the project, to be able to save the resulting outputs. If there is sufficient musical note/instrument variance, the user could create or play interesting musical phrases/songs, record them, and play the sound back later. This would require very in-depth use of Java MIDI Sequencers and global timing techniques, but I believe it would be possible.

Most real-world instruments make use of both hands, so the introduction of another sensor into the system would create a far more realistic environment, as well as allow for more interesting gestures (even if the user is not trying to mimic an instrument) to be created. The current hardware would not allow for intricacies such as finger movements on a guitar fret board to be detected (an interesting example was discussed in the Interim Report [18]), but they could either be totally independent of each and create their own sounds, such as in a drum kit, or the position of one could directly affect the other.

This also introduces the idea of possibly changing the entire instrument depending on where one sensor is, while the other sensor determines the note.

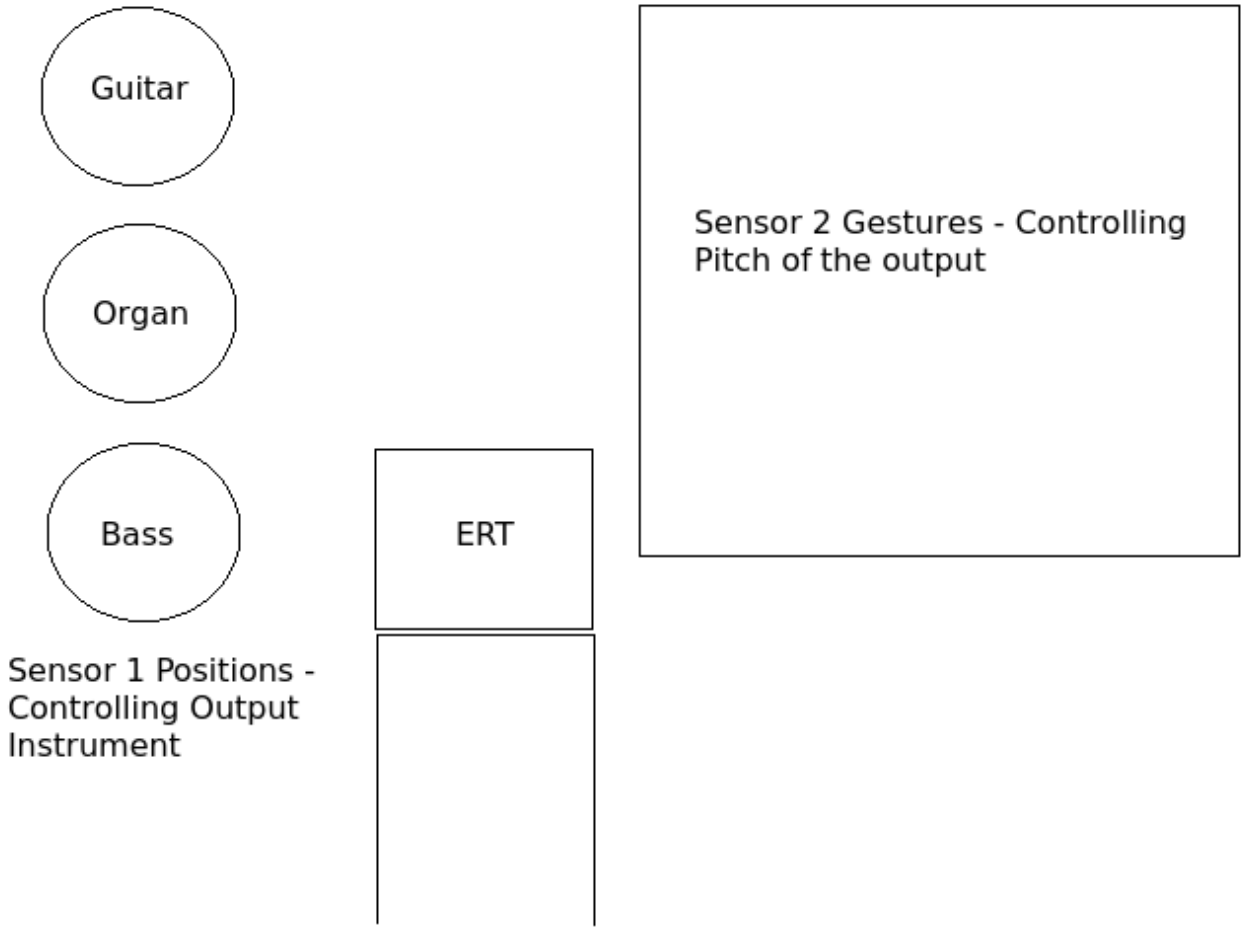

Figure 6.1 – Possible example implementation with two sensors

One other possibility would be to pre-load the system with gestures and instruments to save the user the task of actually training the gestures. This would require a vast amount of training data to be useful, as it would need to accommodate for different people performing the 'same' gesture, but in a slightly different way. This is similar to what is already happening, as discussed in the Background section with the same gestures being technically different, but it would need to be far more versatile and the amount of data required was beyond the scope of this project.

7. Conclusions

The aim of the project as stated earlier was to recognise gestures and produce a MIDI output accordingly. I decided to use the programming language I was most comfortable with, Java, and made use of a relevant freely available library, Jahmm, which helped significantly. To achieve these goals I created Hidden Markov Models, which are used in calculations that produce the likelihood of a performed gesture matching a gesture that has already been saved. HMMs were the ideal mathematical model on which to base the gestures, however I would have liked to associate on HMM per gesture, and not one for each plane. Each gesture has a user-defined name, and user-specified instrument sound from a possible five instruments. Multiple gestures can be saved and recognised and the number could easily be increased from the current maximum of five, although it is likely that the performance of the current system would worsen slightly as a result.

Possible future improvements have been discussed, and there are currently limitations to the system, but it is a good, working example of gesture recognition and motion capture that could easily be extended in the future. However it is not an ideal 'instrument', and remains a proof of concept.

The main goals of the project have all been achieved, and the system is fully functional with a *good* rate of success, which could still be improved.

8. Reflection

Throughout this project I have had to adapt and change the way I look at certain problems in order to overcome them and I certainly feel that I have gained a significant amount of confidence in possibly completing future projects of this kind, and in general problem solving. A major skill I will take from this experience is the ability to regularly stop and view things from a different perspective, be it a coding error or a theoretical principal.

This has been my first ever large scale project spanning many months, so the challenges I faced were not solely research or programming oriented, but largely consisted of time management and organisational aspects. Having other modules to complete, lectures to attend, and numerous extra curricular activities and duties, I either had to perfect my time management, or I simply would have failed. At the beginning this seemed a daunting task, but I can now fully appreciate the benefits both academically and personally.

I set myself numerous time scales and deadlines which I regularly missed, despite being self-imposed. This has taught me to be more realistic in the future and to always allow time for unforeseen problems, either academically or in every day life, as they invariably appear.

Thankfully I made regular notes of my progress including the understanding of theories and many of the problems I encountered – without such comprehensive personal documentation this report would have been made significantly more difficult to write.

References

[1] Rabiner, L. R, 1989, "A tutorial on hidden Markov models and selected applications in speech recognition.", Proceedings of the IEEE 77(2): 257-286.

[2] L. Sucar et al, 2008, Gesture Therapy, International Conference on Health Informatics

[3] Gestrument App Translates Gestures Into MIDI, 2012, Synthtopia [online, accessed 28/11/12], available at: http://www.synthtopia.com/content/2012/11/14/gestrument-app-translates-gestures-into-midi/

[4] Michael J. Lyons & Nobuji Tetsutani, 2001, “Facing the Music: A Facial Action Controlled Musical Interface” Conference on Human Factors in Computing Systems
March 31 - April 5, Seattle, pp. 309-310

[5] Alex De Angelis, 2013, “Gesture Recognition Market To Increase At A Compound Annual Growth Rate Of 50%”, Companies And Markets [online, accessed 20/04/2013], available at: http://uk.finance.yahoo.com/news/gesture-recognition-market-increase-cagr-000000362.html

[6] MotionStar Installation And Operation Guide, 1999, Ascension Technology Corporation

[7] Jonathan. C. Hall, How To Do Gesture Recognition With Kinect Using Hidden Markov Models, 2011, personal blog, [online, accessed 21/04/2013], available at: http://www.creativedistraction.com/demos/gesture-recognition-kinect-with-hidden-markov-models-hmms/

[8] Hidden Markov Model Approach to Skill Learning and its Applications to Telerobotics, Yang, Xu, Chen, 1993, Carnegie Mellon University, Robotics Institute

[9] “Training Hidden Markov Models with Multiple Observations – A Combinatorial Method”, Li & al., IEEE Transactions on PAMI, vol. PAMI-22, no. 4, pp 371-377, April 2000.

[10] Wood. D, Methods For Multi-touch Gesture Recognition For Games, University of Cape Town

[11] Seminar on Hidden Markov Model and Speech Recognition , Nirav S. Uchat , Department of Computer Science and Engineering , Indian Institute of Technology, Bombay Mumbai

[12] François, J.-M. (2006). "Jahmm - Hidden Markov Model (HMM).", [online, accessed 10/12/12], available at: http://code.google.com/p/jahmm/

[13] Joe Starling, Motion Capture To MIDI, Interim Report, Cardiff University, December 2012

[14] MIDI Manufacturers Association Manual

[15] Final Year Project, Alex McCauley, Cardiff University, April 2012

[16] François, J.-M, Jahmm User Guide, available online at https://code.google.com/p/jahmm/downloads/detail?name=jahmm-0.6.1-userguide.pdf

[17] “Hidden Markov Model for Gesture Recognition”, Yang, Xu, 1994, Carnegie Mellon University, Robotics Institute

[18] Karjalainen et al, 2006, Virtual Air Guitar, J. Audio Eng Soc 54(10), pp 964 – 980

[19] Hidden Markov Models, Mouse Gesture Recognition, 2012, University of Lille, [online, accessed 1/05/13], available at [In French]: http://www.lifl.fr/~casiez/VisA/TP/TPMarkov/.