Cheryl Hung

Gesture-controlled Robotics using the Vicon Motion Capture System

Computer Science Tripos

King's College

2009

Proforma

Name:	**Cheryl Hung**
College:	**King's College**
Project Title:	**Gesture-controlled Robotics using the Vicon motion capture system**
Examination:	**Computer Science Tripos, 2009**
Word Count:	**9,500**
Project Originator:	Cheryl Hung
Supervisor:	Cecily Morrison, Laurel Riek

Original Aims of the Project

This project aims to demonstrate the use of machine learning techniques for the purposes of gesture recognition. I have chosen to use full body motion capture as input, capturing three-dimensional kinematic data from arm gestures performed by users. The intention is to use two well known computational models, neural networks and hidden Markov models, in order to convert the gestures into commands to drive robots.

Work Completed

- I have completed a module for real-time preprocessing of the raw input data, including translating relative arm coordinates into body coordinates.
- In addition to the original two recognisers, I have implemented a further two recognisers, which are heuristics-based and aggregative respectively.
- I recorded nearly a hundred gestures for use in training and evaluation, including multiple users and negative examples.

- Finally, I have implemented the multiplayer extension, control for the physical robots and motion capture and also full simulations of both the coordinates stream and command stream.

Special Difficulties

None

Declaration

I, Cheryl Hung of King's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivations

New paradigms for human-computer interaction are becoming a reality. Amongst consumer goods, new ways of interaction encourage innovation beyond the keyboard and mouse. The Apple iPhone is the first touchscreen cellular phone to be widely adopted, while the Nintendo Wii brings accelerometer-based control to home gaming. The Opera browser uses mouse gestures for navigational input, while the Microsoft Surface is aimed towards diverse markets, from corporate events to restaurants.

The philosophy which is common to these systems is the convergence towards "everyday" simplicity. The aim is to allow intuitive, natural interaction so that anybody (in theory) can use them without training, by mirroring or mimicking behaviour of the real world.

Robots, by contrast, have generally been operated by skilled engineers and researchers. The motivation was to bring the power of human intuition to the human-machine interface, using techniques from machine learning. The medium that I chose to investigate and exploit is gesture recognition, a specialization of pattern recognition.

1.2 Previous work

Use of gestures, especially pointing, is an age-old technique for communication. The developing field of machine learning and artificial intelligence over the last sixty years has created an interest in classifying and interpreting these techniques. Artifical neural networks, and especially hidden Markov models, have been shown to model the world realistically to some extent.

The use of hidden Markov models is particularly prominent in speech recognition. However, it has also been applied to gesture recognition; for example, tracking of hand gestures to use as pointing gestures for human-robot interaction[18]. Hidden Markov models have also been used to qualify arm gestures in stroke patients[8], and to control unmanned aerial vehicles[14].

1.3 Project aims

While machine vision is one option for obtaining user input, motion capture systems provide far more detailed location data, down to hundredths of a millimeter. Gesture recognition is a two-phase problem; the training with prerecorded datasets and the run-time interpretation. Rather than accepting arbitrary gestures as input, I chose to train on a finite set of predefined gestures, converting these into commands that are used to drive the robot.

This project uses machine learning techniques for gesture recognition in order to control a remote vehicular robot. Three very different approaches were taken to the same problem; a heuristic evaluator, artifical neural networks, and hidden Markov models. The gesture data was recorded from a motion capture system using arm and body markers for training purposes and was preprocessed for invariance to translation and rotation before feature extraction. This allows users to be facing in any direction at any point, and also to move around the area whilst performing a gesture. A voting system was implemented to aggregate the results and intepret the real time data as commands. Furthermore, an extension for multiple users was implemented, and the system demonstrated successfully on live motion capture data to control physical hardware.

Chapter 2

Preparation

2.1 Requirements Analysis

The project is divided into three phases; preprocessing of motion capture data, recognition and intepretation into commands, and execution on hardware. The overall requirement is that the system should allow the user to control a small robot using arm gestures, in real time. Broken down into subgoals:

- The system should implement multiple recognisers, including neural networks and hidden Markov models. In order to be transparent to the programmer, it should provide a consistent API for training and recognition for each classifier.
- The system should receive data from the motion capture system and preprocess it in order to have the properties of location and rotation invariance.
- The stream of raw data should be segmented into gestures without requiring an operator.
- A training module should be read gesture data files from disk, train for a particular user, and save and load this trained state.
- The system should be fully usable without any physical hardware; in other words, a simulation for a stream of coordinates and a representation of virtual robot.
- Finally, the system should intepret the commands in order to drive the robot.

2.1.1 Hardware

The main requirements of the hardware were:

1. The system should accept a range of commands, in order to allow training of multiple gestures. A simple vehicle sufficed for this purpose.
2. The system should have wireless capabilities for independent movement.
3. The system should provide a high level API or library, for ease of development.
4. Purchasing multiple units for multiuser applications is easier with commercially available robots.
5. The system should have high battery life and low power consumption, for testing purposes.
6. The system should be relatively low cost (see budgeting for more)

The main alternatives considered were:

1. Lego Mindstorms, already owned by the Computer Laboratory and including a C++ API, with purchased modules for wireless and touch sensors. This was dismissed as building a vehicle, testing for range of movement and so on would take excessive development time.
2. iRobot Roomba, a low cost vacuum cleaning robot with bump sensors, with third party peripherals for wireless and webcam. Reaching the serial ports on the Roomba would require removal of hardware and deconstruction of the vacuum innards.
3. iRobot Create[11], an educational robot similar to the Roomba, without vacuum parts. A device for wireless could be attached by serial cable. A setup based on the OLPC Telepresence[12] also provides a Python library for interfacing with both the laptop's webcam and robot's motors and bump sensors.
   This was the option chosen, for ease of development and cost effectiveness.

The budget was £500, funded equally by the Computer Laboratory Outreach programme and King's College. The final expenditure came to £585 as follows:

Two iRobot Creates at $229.99 = £306.80
OLPC XO laptop at $220.00 = £151.86
OLPC XO laptop at $170.00 = £117.35

UK/US Step Down Adapter = £8.99
Total: £585.00

The main reason for the increase was that the basic iRobot Create comes without a rechargeable battery, and requires 12 AA batteries per robot per hour. The upgraded package comes with a command module as well as rechargeable battery; however it was found that the command module cannot be used, as it considers the laptop attached via USB to be a peripheral rather than the laptop being the USB master device to the robot slave.

2.2 Motion Capture System

The interface to the Vicon Motion Capture System in the Rainbow Group is provided by their proprietary software package Vicon Real-Engine, TARSUS[3]. This software collates the data from ten infra-red cameras to reconstruct the three dimensional marker positions in real time, and further combines these into either user-defined objects or full skeletons. There were two choices to be considered; the objects (body parts), and the gestures to be recognised.

The objects are needed firstly for input to a gesture, and secondly to represent the user's position so that the gestures can be calculated with invariance to translation and rotation. For the gesture input, one or two hands or arms were considered; two gives a wider range of movement (and thus a larger input space) and forearms have more rigidity than hands. This gives better recognition rates from the Vicon system because it increases the likelihood that the seen objects will match the saved body part.

Options for positional data were belt, hat and body. Having experimented with all three options, empirical results showed that the belt was easily occluded and confused by the arm markers, while the hat gave poor rotational data due to the independent movement of the head. The final configuration chosen was twelve markers spread across two arms and body (upper chest and back).

The data output is six data values per object; an Axis Angle triplet for rotation and a triplet for global translation in millimetres. An alternative rotation system which is also available is Euler angles.

In order to simplify data processing while providing adequate control over the robot, most of the pre-set gestures were constrained to one arm movement within two dimensions (x-z or y-z planes in a z-up world). The commands chosen to provide basic steering and speed were accelerate, decelerate, turn left, turn right and start/stop, the latter being a test case for a more complex gesture, in this case a single clap at chest level. Figure 2.1 shows the arm gestures for each command.

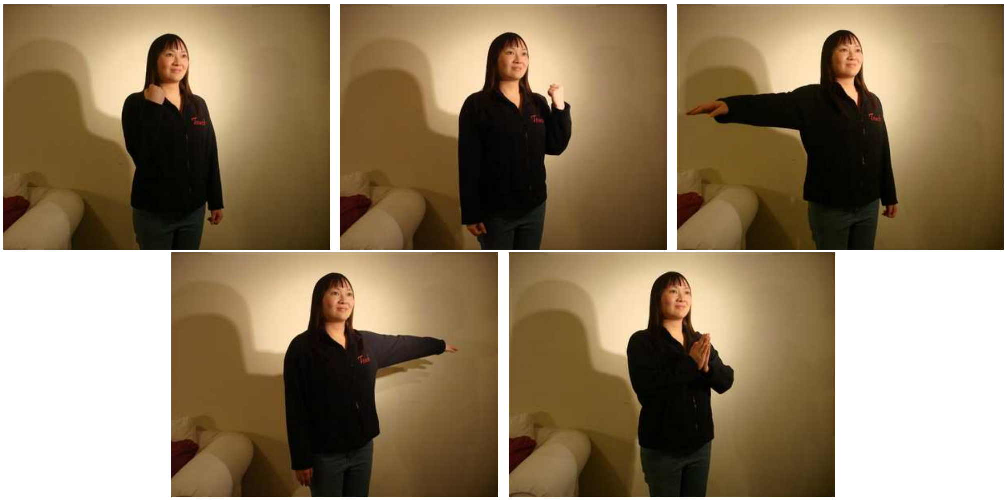

Figure 2.1: Top row: arm gestures for *accelerate*, *decelerate*, and *turn right*. Bottom row: *turn left*, and *start/stop*.

$$\theta = \sqrt{x^2 + y^2 + z^2}$$

$$c = \cos(\theta)$$
$$s = \sin(\theta)$$
$$t = 1 - c$$

$$R = \begin{pmatrix} tx^2 + c & txy + sz & txz - sy \\ txy - sz & ty^2 + c & tyz + sx \\ txz + sy & tyz - sx & tz^2 + c \end{pmatrix}$$

Figure 2.2: Axis angle rotation matrix.

2.2.1 Axis Angles

Axis angles are also known as exponential coordinates or rotation vectors. This parametrizes orientation by a three dimensional Cartesian vector, describing a directed axis and the magnitude of rotation. The rotation matrix shown in figure 2.2 is used to rotate around an arbitrary axis where (x, y, z) is a unit vector on the axis of rotation, and θ is the angle of rotation.[9]

2.2.2 Euler Angles

With Euler angles, the rotation matrix is decomposed into three rotations from a reference frame, (x, y, z). The rotated orientation system is denoted in upper case letters, (X, Y, Z).

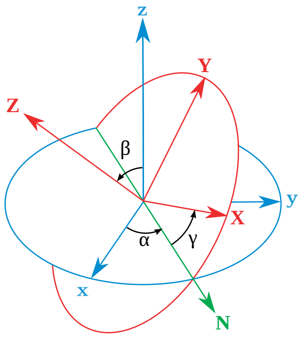

Figure 2.3: Euler Angles[1]

The line of nodes (N) is the line of intersection between the xy and XY coordinate planes, and the new coordinate system is parametrised by (α, β, γ).

In the standard z-x-z convention,

- α is the angle between the x-axis and the line of nodes, modulo 2π
- β is the angle between the z-axis and the Z-axis, modulo π
- γ is the angle between the line of nodes and the X-axis, modulo 2π

However, when the xy and XY planes are identical and the z and Z axes are parallel, not all points in the coordinate system can be uniquely identified. This phenomenon known as *gimbal lock*. Euler angles are also prone to angle flips at the extremities of the ranges of α, β, γ. For this reason, Euler angles are harder to deal with than axis angles, which vary smoothly.

2.3 Pattern recognition

There are many different techniques in the field of machine learning used for pattern recognition. The purpose of the pattern recognition phase is to classify the observations into categories, based on extracted features. For supervised learning, a training set of patterns is labelled with the correct classification; unsupervised learning (such as the K-means clustering algorithm) evaluate the raw, unlabelled data and attempt to infer the patterns, for example by minimizing the root mean squared error based on Euclidean distance. The implementation of hidden Markov models offer both learning techniques, in the form of the Baum-Welch algorithm and the K-means clustering algorithm. Backpropagation is the most standard supervised learning technique for neural networks.

$$\phi(v_i) = 1/(1 + e^{-v_i})$$

Figure 2.4: The sigmoid function.

Statistical techniques for pattern recognition are based on finding a classifier $h : X \mapsto Y$ which maps $x \in X$ to $y \in Y$ in a close approximation to the actual ground function $g : X \mapsto Y$ where the training set $(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)$ are instances of $X \times Y$ (the Cartesian product of the X and Y domains).

Among the more popular techniques are support vector machines, naive Bayes classifier, k-nearest neighbour, neural networks and hidden Markov models. I chose the latter two as an example of a general classifier and temporal (or sequential) classifier respectively.

2.3.1 Neural Networks

An artificial neural network is an interconnected set of nodes which individually perform simple processing, but which exhibits complex behaviour as a whole system. This is the connectionist approach to pattern recognition from large sets of data, inspired by biological neural networks. Each unit combines the inputs by means of an activation function, which *fires* or adjusts its output non-linearly based on the value of the input. Common choices for the activation function are the $\tanh = \frac{\sinh x}{\cosh x}$ function or the sigmoid function, which have the property of being differentiable:

In the discussion I use the following notation:

w_{ji} connects node i to node j
a_{ij} is the activation for node, the weighted sum of the inputs $= \sum_i w_{ji} z_i$
g is the activation function
$z_j = g(a_j)$
bias input $= 1$

The most common model is the multilayer perceptron[5], where the overall structure is feedforward and there are three layers of nodes; an input layer, a hidden layer and an output layer.

Since this is a supervised learning technique, the first phase is to present training data. The goal is to adjust the weights w so as to minimise the overall error, denoted $E(\mathbf{w})$. The training sequence is a vector of labelled inputs:

$$s = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m))$$

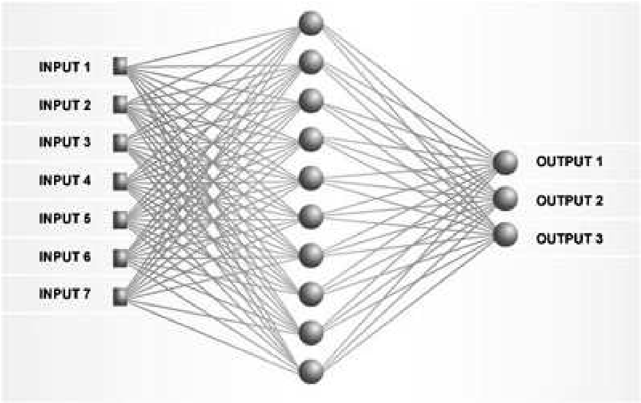

Figure 2.5: Multilayer perceptron, with a single layer of hidden nodes. Each layer of nodes is fully connected with the next layer.

The backpropagation algorithm is an application of gradient descent for minimization of error[5]. The first stage is to initialize w to a random set of weights. Calculate $E(\mathbf{w})$; if this is greater than a threshold value, calculate the gradient $\partial E(\mathbf{w})/\partial \mathbf{w}$ of $E(\mathbf{w})$ at each point w_i and adjust the weight vector to reduce the error:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right| \mathbf{w}_i$$

The forward propagation stage is to calculate a_j and z_j for all nodes, given an input example p.

$$\frac{\partial E_p(\mathbf{w})}{\partial w_{ji}} = \frac{\partial E_p(\mathbf{w})}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i$$

where $\delta j = \frac{\partial E_p \mathbf{w}}{\partial a_j}$ and $\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ij}} \left(\sum_k z_k w_{jk} \right) = z_i$

There are two cases for calculating ∂_j:

1. j is an output node

$$\delta j = \frac{\partial E_p(\mathbf{w})}{\partial a_j} = \frac{\partial E_p(\mathbf{w})}{\partial z_j} \frac{\partial z_j}{\partial a_j} = \frac{\partial E_p(\mathbf{w})}{\partial z_j} g'(a_j)$$

2. j is not an output node

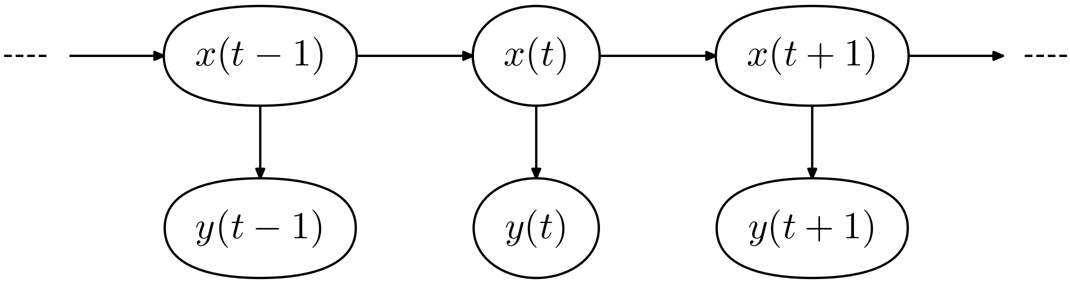

Figure 2.6: Hidden Markov models

$$\delta j = \frac{\partial E_p(\mathbf{w})}{\partial a_j} = \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \frac{\partial E_p(\mathbf{w})}{\partial a_k} \frac{\partial a_k}{\partial a_j} = g'(a_j) \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \delta_k w_{kj}$$

since $\frac{\partial a_k}{\partial a_j} = \frac{\partial}{\partial a_j} \left(\sum_{k \in \{k_1, k_2, \ldots, k_q\}} w_{ki} g(a_i) \right) = w_{kj} g'(a_j)$

This gives the *back-propagation formula*[5]:

$$\delta j = g'(a_j) \sum_{k \in \{k_1, k_2, \ldots, k_q\}} \delta_k w_{kj}$$

The importance of this result is the reduction in time complexity, as training neural networks is very time consuming. The majority of the work is done calculating $a_{ij} = \sum_i w_{ji} z_i$. Letting W be the total number of weights and biases, there are W derivatives to calculate for each weight. The naive approach of explicit evaluating each derivative numerically in $O(W)$ operations is therefore $O(W^2)$ in time complexity. By contrast, backpropagation (using $\frac{\partial E_p(\mathbf{w})}{\partial w_{ji}} = \delta_j z_i$ to calculate the derivatives) reduces the computational complexity to $O(W)$.

2.3.2 Hidden Markov Models

First order Markov processes are a class of statistical model in which the probability of being in some future state is only dependent on the current state. This is also known as the memoryless model, since the “memory” or past states have no effect on the next state.[19]

$$Pr(S_t | S_{0:t-1}) = Pr(S_t | S_{t-1})$$

where $S_{0:t-1} = (S_0, S_1, \ldots S_{t-1})$ is the set of states from time 0 to time $t-1$.

At each time t, a symbol is emitted from state i. For a Bakis HMM, the only options at $t+1$ are to stay in state i or move to state $i+1$[6]. Each state is parametrized by an emission probability of staying in the same state and a transition probability of moving to state $i+1$.

A hidden Markov model is one where these probabilities characterizing the states are unknown, and for a training sequence $E_{0:t} = (E_0, E_1, \ldots, E_t)$, it is unknown which state the observation lies in. To find the probabilities, the Baum-Welch algorithm takes a vector of training sequences and iterates using Expectation-Maximization. The expectation is calculated by the forward-backward algorithm, which is an example of dynamic programming and the use of memoization to avoid recomputing solutions to sub-problems.

For a Markov Model, $Pr(E_t|S_t)$ is the sensor model and $Pr(S_t|S_{t-1})$ is the transition model. $Pr(S_0)$ denotes the prior state. The assumption is that the probabilities do not change over time, as the observations are stationary processes.

There are four basic inference tasks that can be calculated using this formalism[19]:

1. Filtering: deducing the current state that we might be in, $Pr(S_t|e_{1:t})$
2. Prediction: deducing the future state that we might be in, $Pr(S_t|e_{1:t})$ for some $T > 0$
3. Smoothing: deducing the past state that we might have been in at time T, $Pr(S_t|e_{1:t})$ for some $0 \leq t < T$
4. Viterbi path : deducing the most likely sequence of states so far, $\underset{s_{1:t}}{\text{argmax}} Pr(S_t|e_{1:t})$

The Viterbi algorithm solves the final task, which finds the maximum of the probabilities of taking each path through the states:

$$\begin{aligned}
\max_{s_{1:t}} Pr(s_{1:t}, S_{t+1}|e_{1:t+1}) &= c \max_{s_{1:t}} Pr(S_{t+1}, s_t) Pr(e_{t+1}|S_{t+1}) Pr(s_{1:t}|e_{1:t}) \\
&= c Pr(e_{t+1}|S_{t+1}) \max_{s_t} \left(Pr(S_{t+1}|s_t) \max_{s_{1:t-1}} Pr(s_{1:t-1}|e_{1:t}) \right)
\end{aligned}$$

The algorithm therefore walks forwards over all possible sequences, calculating the most likely sequence for reaching the final states and selecting the most likely sequence of states. The time complexity of the Viterbi algorithm is $O(t)$ in the length of the sequence, t, since it performs the forward message pass phase once per possible path. The space complexity is also linear in t, for storing the t pointers which indicate the best stage sequence.[19]

```
for each possible state sequence:
  for each state at time t, $s_t$:
    forward a message $m_{1:t}$ combining the filtered estimate $Pr(S_t|e_{1:t})$ and updated with
      the new observation, to give the probability of the best sequence reaching this state
  return maximum of the probabilities and follow the pointers back to find the most likely
    sequence
```

There are two major training algorithms, Baum-Welch and K-Means. Both of them are implementations of the Expectation-Maximization phase described above.

Baum-Welch

The Baum-Welch training algorithm was first introduced by Baum L.E. and Weiss, N[4]. It is a two-phase algorithm; the first requires calculating the forward and backward probabilities to be passed as messages, and the second phase calculates the expected count of the particular transmission-emission pair.

K-means

The k-means algorithm, also known as Lloyd's algorithm[15], is a heuristic solution to the NP-hard problem of partitioning data $x_1, x_2, \dots, x_n$ into k-clusters, $S = s_1, s_2, \dots, s_k (k < n)$ and minimizes the within-cluster sum of squares error:

$$\underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x_j \in S_i} \|x_j - \mu_i\|^2$$

where μ_i is the mean of S_i.

2.4 Feature extraction

Neural networks have a fixed number of input nodes and so variable length data must be converted to a fixed size feature vector. This technique, known as dimension reduction, also reduces the search space significantly, which reduces training times and can improve recognition rates.[5]

There are several ways of doing this. For a discretely sampled time series, as the position data, the Discrete Fourier Transform encodes the signal as a coefficients of linear combinations of basis functions in a new frequency domain.[20] Haar wavelets are an alternative method which also uses linear combinations of wavelet functions, that can represent discontinuities better than the Fourier method.[21]

However these techniques were deemed excessively complex for the purposes of gesture recognition which only has a finite set of planar gestures, as in my case. I have therefore assumed that the simpler method of extracting axis-aligned ranges of movement is sufficient for distinguishing the gestures without adding the computational overhead of these other techniques.

2.5 Development environment

The majority of the system was developed under Linux with the Ubuntu 8.10 distribution. Programming was using a mixture of NEdit, gedit, and the command line tools `javac`, `java` and `python`.

The XO laptops run a modified version of Fedora Linux, while the PC used during live capture runs Windows XP.

The SRCF (Student-Run Computing Facility[2]) is used as a central server for data transfer.

2.5.1 Tools

Subversion was used for version control, with Google Code Project Hosting providing backup support. The libraries used were:

Joone: Java Object Oriented Neural Engine [16]
Jahmm: Java Hidden Markov Model [7]
SCOP: Server COmmunication Protocol [10]
PyRobot: robotics control [13]
TARSUS: motion capture

Rather than implementing all the algorithms from scratch as originally planned, I chose to use established libraries for my machine learning needs. This came about on advice from my supervisor and has reduced the workload in this case, to give me time to implement a more complete final system.

I researched several neural network libraries, including Joone[16] and JavaNNS. The Joone library has particularly good tools for writing third party modules, as well as support for distributed training, which would be useful for higher performance in the CPU intensive training stage.

Java libraries for Hidden Markov Models included Jahmm[7], jHMM, and HmmSDK. I chose Jahmm as it has a good API and support for processing of vectors.

Decoupling various components using SCOP allows message passing between different languages, which was important to allow the Java client to communicate with the Python robotics control. This also meant that I could use a bottom-up, modular approach to each sub-problem.

This dissertation was typeset using LaTeX together with `bibtex`. The graphs shown in the evaluation were generated using `gnuplot`; other diagrams were created using Inkscape.

2.5.2 Languages

The choice of languages was based on ease of development, portability, availability of libraries, and type safety. Python was chosen for readability and rapid prototyping, while Java was used for interfacing with powerful libraries. In a few cases certain classes have been prototyped in Python before being ported to Java.

The robot control library, **PyRobot**, is written in Python. The client used to interface with the TARSUS system is written in Java[17]. Message passing of simple string data is handled by SCOP, which has ports to both Python and Java.

This project uses several architectures; the OLPC XO runs a modified version of Fedora Linux, the Tarsus system runs on Windows XP and the development environment was Ubuntu Linux. Therefore it is important to use architecture independent languages, making it easy to perform processing on a more powerful computer than the XO laptop.

Chapter 3

Implementation

3.1 System Overview

Figure 3.1 shows how components are decoupled for re-usability. During training, the training sequences are read from disk and preprocessed to form data for training the pattern recognition modules. When live, the data capture subsystem sends arrays of raw data values to the SCOP server. The preprocessing module additionally listens to the event stream for gesture data. The output from all three services is interpreted as one of five commands by Control and is sent to the SCOP server on a different stream. For testing purposes, a GUI provides alternative input methods of mouse and keystrokes.

The relay program on the XO laptop converts this into drive commands and the PyRobot library handles low level opcodes. The monitor shows the output for a turtle program which responds to the same commands, again for test purposes.

Message passing provides inter-language communication via an SCOP server running on the SRCF. Logically, the processing can take place on a remote computer, but for performance reasons, I chose to perform processing on the same machine that is used to collect Vicon data.

3.1.1 Component Interfaces

All SCOP messages are ASCII strings. Each class that interacts with the server has a name and can express an interest in one or more streams, or `endpoints`, by opening sockets.

NB: All names in the following discussions containing `p1` have `p2` equivalents for the second player.

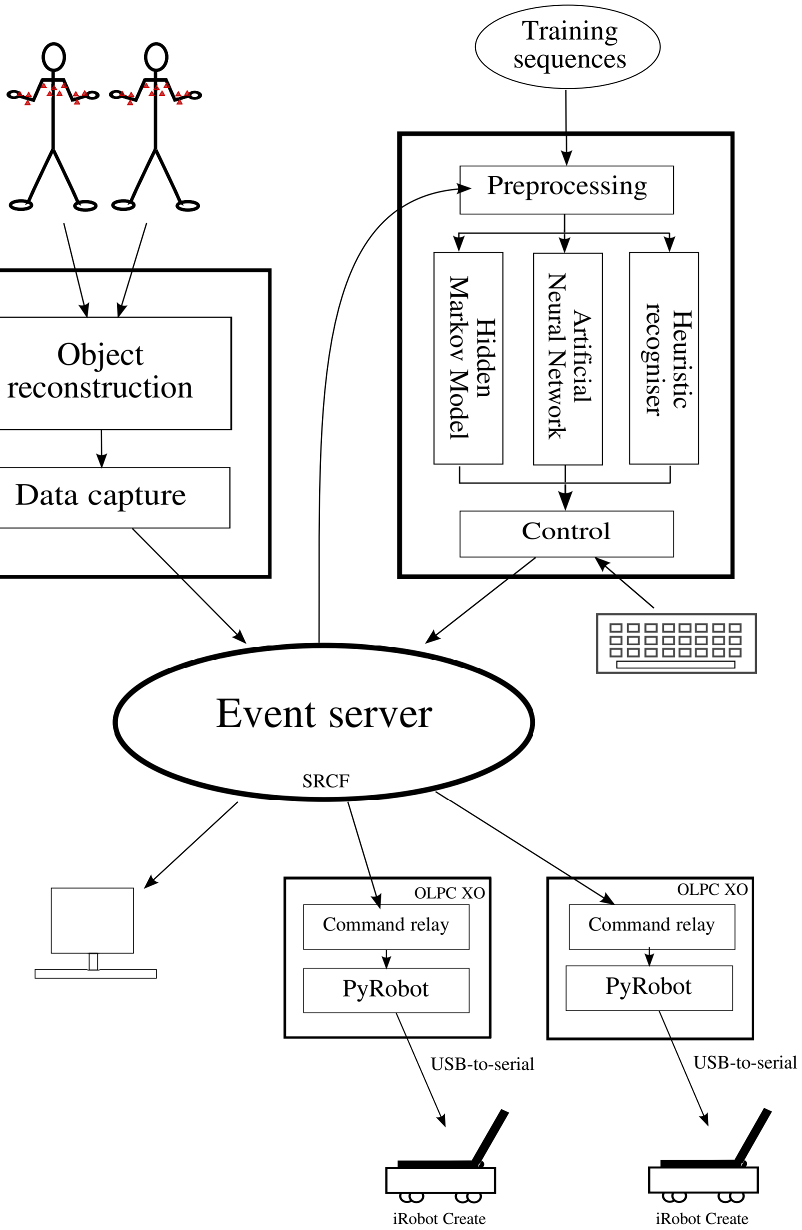

Figure 3.1: System Diagram

3.1.2 Protocols and data formats

This table shows the protocols for interaction between various components. The three major data streams indicated by `coordserver`, `ctrlserver` and `statusserver` can be redirected completely independently in a way that is transparent to users. Modules publish events and subscribe to interests denoted `p1*` under the SCOP identifiers `*p1`. The `^`, `&` and `|` refer respectively to whether an instance of a class can listen to a single endpoint (taking a command line argument for `p1` or `p2`), to both endpoints simultaneously, or to either one or two as needed.

Figure 3.2 shows how the system components are decoupled throughout. The robots are connected via 802.11 wireless.

There were two options for drive commands; accelerate and decelerate, or drive forwards and backwards for a set time. Accelerate and decelerate with continual motion was chosen to avoid users having to continually issue commands to their robot. Start/Stop was chosen to provide an instantaneous way to halt the robot. This meant that turning should also be a smooth turn, rather than *stop* followed by a *turn in place* and then a *start*.

<table>
<tr><th>SCOP server</th><th>Endpoint</th><th>Event sources</th><th>Event sinks</th></tr>
<tr><td>coordsserver</td><td>p1coords</td><td>capturep1, simulatep1</td><td>windowp1</td></tr>
<tr><td></td><td colspan="3">The coordsserver transmits both players' streams of coordinates, labelled p1coords and p2coords. The data format consists of eighteen floating point values, followed by a string which expresses the case where possible failure of object tracking by TARSUS has caused a body part to drop out. {
<body-ax>, <body-ay>, <body-az>,
<body-tx>, <body-ty>, <body-tz>,
<leftarm-ax>, <leftarm-ay>, <leftarm-az>,
<leftarm-tx>, <leftarm-ty>, <leftarm-tz>,
<rightarm-ax>, <rightarm-ay>, <rightarm-az>,
<rightarm-tx>, <rightarm-ty>, <rightarm-tz>,
<"ok"|"dropout">
}

All double values are accurate to six significant figures, in decimal. The axis angles values (ax,ay,az) are denormalised and in degrees (−180° to +180°). Lost objects (reported as (0,0,0) for the angles) are converted to (0,0,1) and the status field set to "dropout". When non-zero values appear the status field is reset to "ok". Translations (tx,ty,tz) are in millimetres. The default rate is 100fps.</td></tr>
<tr><td>ctrlserver</td><td>p1ctrl</td><td>controlp1, windowp1</td><td>viewp1, relayp1</td></tr>
<tr><td></td><td colspan="3">The ctrlserver handles the output from gestures, as single lower case characters representing an accelerate, decelerate, turn left, turn right or start/stop command:
<"a"|"d"|"l"|"r"|"s">
The commands may be acted upon by the simulated or physical robots. Typically, a user issued around one per second.</td></tr>
<tr><td>statusserver</td><td>p1status</td><td>windowp1</td><td>feedbackp1</td></tr>
<tr><td></td><td colspan="3">This stream is a direct transcription of the <"ok"/"dropout"> field of the coordinates. It reflects whether the data provided by TARSUS can be used for the purposes of recognition.
<"ok"|"dropout"></td></tr>
</table>

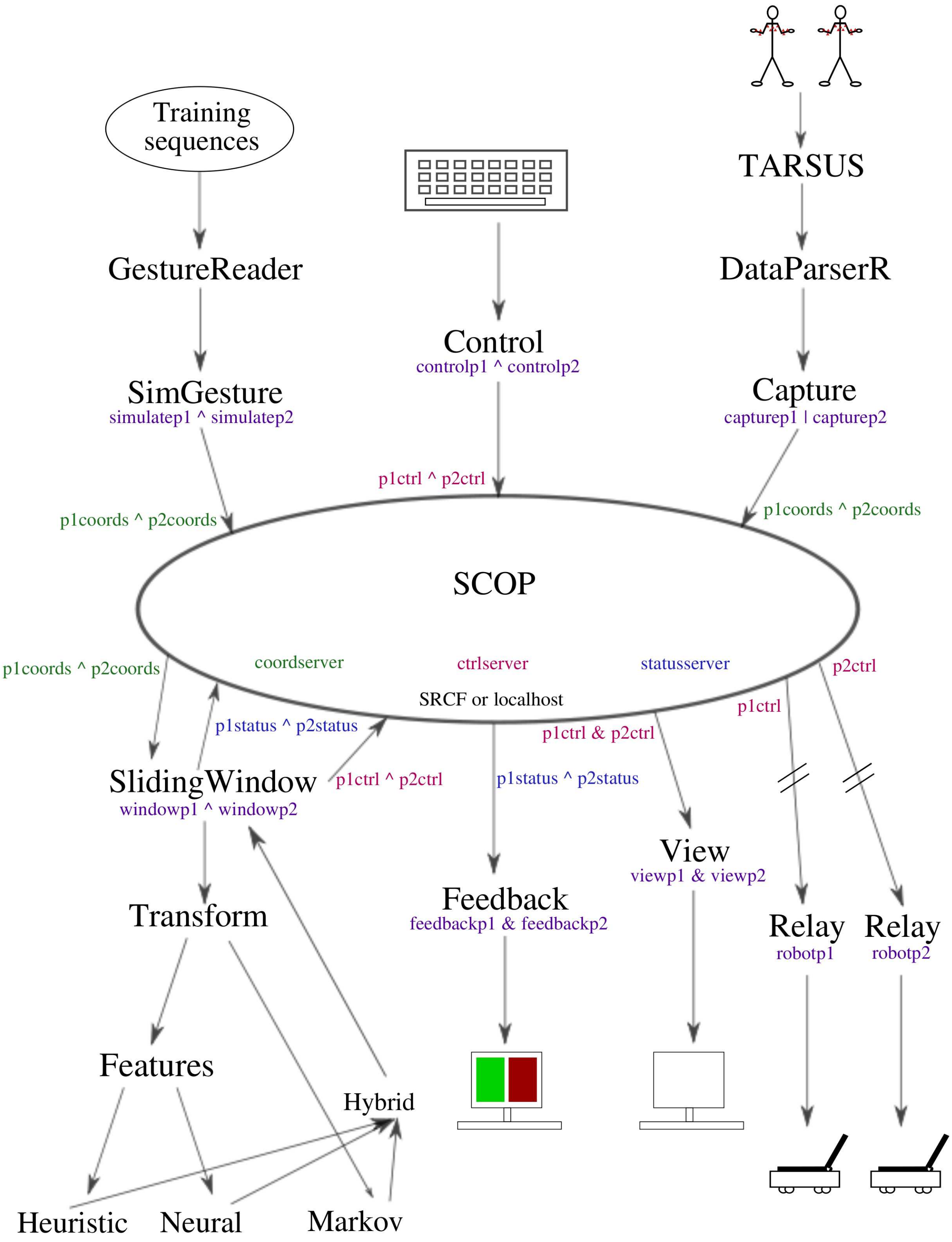

Figure 3.2: Data flow diagram

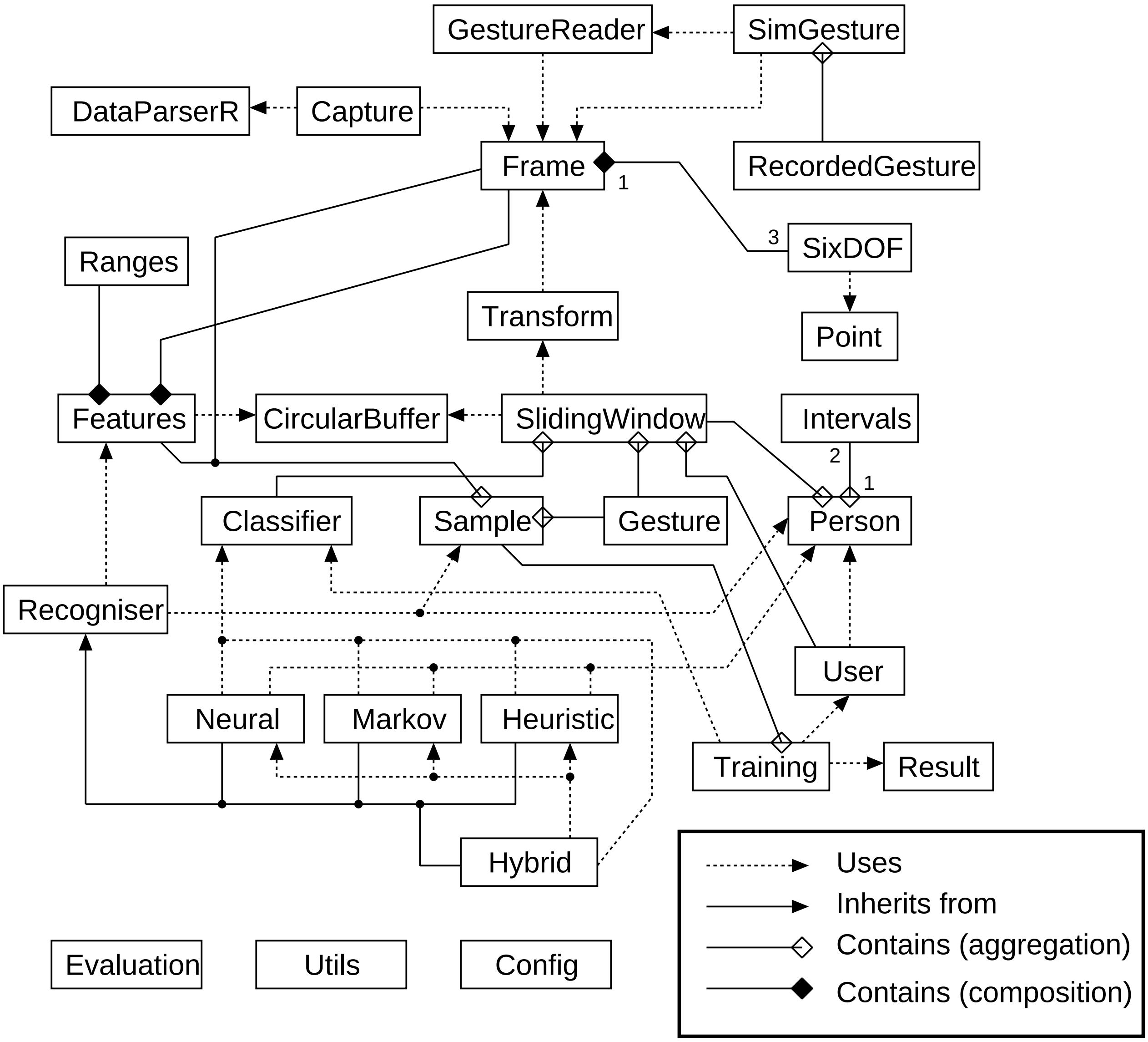

Figure 3.3: Class diagram. Note that **Evaluation**, **Utils** and **Config** interact with almost all other classes and have been truncated for clarity.

3.1.3 Breakdown of components

Figure 3.3 shows the class diagram for the Java classes. The Python modules **control**, **view**, **feedback**, **relay** and **eval** are independent and thus not included on the diagram.

The method signatures for each class are listed in more detail within the pertinent section.

3.2 Motion capture

The TARSUS system triangulates markers using infra-red emitted from the cameras, which is bounced back from the reflective markers. It uses stereoscopic techniques to resolve these into world coordinates, using ten cameras for redundancy when a marker is occluded for whatever reason.

With three objects per user and six data-points per object, the Tarsus server sends eighteen or thirty six (for two players) values per frame via TCP/IP on port 800, at 100 frames per second.

Capture
`int FRAME_RATE`
`public static void output(double[] data, int startpos, SCOP scop, String label)`
`public static void main (String[] args)`

DataParserR
`DataInputStream is`
`DataOutputStream os`
`String[] channels`
`public double[] getData()`
`private String[] parseInfoPacket()`
`private void matchStrings(String[] channels)`
`private double[] parseDataPacket() throws IOException`
`private static int readNum(DataInputStream is) throws IOException`
`private static double readDoub(DataInputStream is) throws IOException`

The **Capture** class uses the Java client **DataParserR.java** to multiplex on the requested channel names, determining whether one or two players are currently using the system. It opens up to two sockets to a SCOP server running on localhost and forwards `p1coords` and `p2coords` to the gesture recognition phase. It also performs downsampling to a configurable frame rate (default is 100fps) to reduce burstiness and to avoid overloading the server buffers.

There are two potential problems with the data: glitches and dropouts.

- Glitches are caused by incorrect object reconstruction. They are transcribed as valid data values from an object which have a high rate of change in a brief period. For example, a glitch could be:
  `-1, -2, -1, 53, 168, 168, 89, -4, -3, 0, ...`

- Dropouts are the result of improper triangulation from the cameras, and so the objects are not recognised at all and result in all rotational data being reported as 0.0 and translation data repeated from the last known values. A dropout is reported as:
  `33, 33, 34, 35, 35, 0, 0, 0, 0, 36, 35, 35, ...` (rotation)
  `697, 692, 680, 669, 669, 669, 669, 669, 669, 624, ...` (translation)

Dropouts are handled gracefully by setting a 19th flag in the stream, and converting rotations smaller than a given ϵ into harmless zero rotations about the Z-axis during preprocessing. Glitches are relatively hard to detect, relatively harmless, and the solution for fixing is less clear, so they are currently ignored.

3.2.1 Training data

Training data consists of five sequences for each of the five gestures, giving a total of twenty five gestures taken from each session, recorded by two users for robustness. The total number of input vectors used for training was 75.

In addition, two extra sets were recorded. The first is for calibration, consisting of stationary data at the origin, axis aligned translations, and 90 degree rotations. The second is examples of non-gestures, such as the neutral arms by side position. These provided negative examples to the pattern recognition engine.

The data is formatted as comma-separated values (CSV) files, as in figure 3.4. All values are double precision floating point numbers to six significant figures. Rotations are in the range ±180° and translations are in millimetres from the origin in a z-up world.

The following two classes are used for training purposes. **GestureReader** reads in and parses the data from disk, while **SimGesture** simulates a live user performing those gestures. This allows the recognition engine to be tested without needing a live data stream from the Vicon system.

GestureReader
`public static ArrayList<Frame> getData(String filename)`
`static SixDOF parse(String s)`

RecordedGesture
`String filename`
`ArrayList<Frame> data`

```
BodyP1
Frame,BodyP1<A-X>,BodyP1<A-Y>,BodyP1<A-Z>,BodyP1<T-X>,BodyP1<T-Y>,BodyP1<T-Z>,
1,-0.991818,3.4194,-102.972,-715.545,820.366,1186.36,
2,-0.987695,3.40335,-102.978,-715.479,820.352,1186.33,
...
242,-1.61596,3.84801,-101.36,-696.667,821.255,1186.13,

LeftArmP1
Frame,LeftArmP1<A-X>,LeftArmP1<A-Y>,LeftArmP1<A-Z>,LeftArmP1<T-X>,LeftArmP1<T-Y>,LeftArmP1<T-Z>,
1,106.316,100.107,-19.2348,-660.681,1039.83,928.706,
2,106.457,99.9967,-19.219,-660.629,1039.9,928.689,
...
242,106.069,102.104,-19.6354,-649.414,1042.67,927.508,

RightArmP1
Frame,RightArmP1<A-X>,RightArmP1<A-Y>,RightArmP1<A-Z>,RightArmP1<T-X>,RightArmP1<T-Y>,RightArmP1<T-Z>,
1,27.848,44.5584,-47.155,-670.526,559.707,938.067,
2,28.3683,44.5345,-47.6479,-670.71,559.914,937.873,
...
242,30.3788,43.5149,-55.0284,-642.649,555,938.528,
```

Figure 3.4: An example of a typical training file.

SimGesture

`static ArrayList<RecordedGesture> read_gesture(String gesture_dir)`
`public static void interpolate_gestures(SCOP scop, RecordedGesture from_gesture, RecordedGesture to_gesture, int duration)`
`private static Frame interpolate_frames(Frame from, Frame to, double weight)`
`private static SixDOF interpolate_sixdof(SixDOF from, SixDOF to, double w)`
`public static double interpolate_angle(double from, double to, double w)`
`static void framesync()`
`public static void replay_gesture(SCOP scop, RecordedGesture gesture)`

The **GestureReader** class contains a `getData()` method, which reads in the data line by line until it finds an object name it recognises ("BodyP1", "LeftArmP2" etc), ignores the column headings, and parses each frame until it finds whitespace. Each frame becomes a **Frame** object, containing references to three **SixDOF** objects for body, left arm and right arm. The **SixDOF** (six degrees of freedom) holds the six double precision floating point numbers corresponding to <R-X>, <R-Y>, <R-Z>, <T-X>, <T-Y>, <T-Z>; calling `normalise()` sets the angle of rotation and normalises the rotation tuple to a unit vector. The **SimGesture** class is used to replay the data in the same format as a Vicon stream on the same SCOP stream, rendering it identical to the real data. It uses **GestureReader**.`getData(filename)` on all CSV files from a single person and replays them in a random order, appending the dropout boolean in the same way as **Capture**. It assumes that all recorded gestures begin and end in the neutral position, and so movement between gestures is represented by linearly interpolating all values for a random length of time. It also performs a `framesync` in parallel to ensure that data is emitted at 100fps.

3.2.2 Preprocessing

The data requires significant preprocessing to convert the feature vector from world coordinates to body coordinates. This is performed by the **Transform** class together with the **SixDOF**, **Frame**, and **Point** classes.

Transform
`static void process(ArrayList<Frame> data)`
`static void process(Frame f)`

SixDOF
`double ax, ay, az, angle, ty, ty, tz`
`static final double EPSILON = 1.0e-5`
`SixDOF(double[] a, int offset)`
`public void normalise()`
`public String toString()`
`void rotate (double bearing)`
`double calcHeading()`
`void translate(SixDOF axes)`

The **SixDOF** class holds doubles for `ax,ay,az,tx,ty,tz` and converts the `ax,ay,az` Axis Angle triplet into a normalised vector plus an angle for the rotation magnitude. It converts small rotations into a zero rotation about the z-axis in order to avoid floating point errors.

Frame
`SixDOF body, left, right`
`Frame(String s)`
`Frame(double[] a, int offset)`
`public String toString()`
`public double[] toDoubles()`

The **Frame** class contains three of these **SixDOFs** for body, left arm and right arm. It has multiple constructors for loading from file or live stream, and can also be converted back to the same formats.

Point
`static final double EPSILON = 1.0e-5`
`Point(double x, double y, double z)`

`static Point rotatePoint(double ax, double ay, double az, double angle,`
`double x0, double y0, double z0)`

Point contains a static `rotatePoint` method which implements the conversion from axis-angle triplets to matrix rotations. It takes an axis angle and a point to be rotated and returns a **Point**.

```
double ax, ay, az, angle; //axis angle
double x0, xy, xz; //Point

double s = Math.sin(-angle);
double c = Math.cos(-angle);
double t = 1 - c;

Point p = new Point();

p.x = (t*ax*ax + c)*x0 + (t*ax*ay + s*az)*y0 + (t*ax*az - s*ay)*z0;
p.y = (t*ax*ay - s*az)*x0 + (t*ay*ay + c)*y0 + (t*ay*az + s*ax)*z0;
p.z = (t*ax*az + s*ay)*x0 + (t*ay*az - s*ax)*y0 + (t*az*az + c)*z0;
```

3.2.3 Gesture segmentation

The raw gesture data consists of a stream of vectors of 18 double-precision floating-point numbers, at 100 frames per second. In order to discover the start and end of gestures, the recogniser is repeatedly run on multiple rectangular windows of different sizes until a success is reported. Since the preprocessing and feature extraction is performed on every frame, I have implemented an optimization using dynamic programming to store processed frames in a circular buffer containing the last 500 frames of data.

SlidingWindow

`static User user`
`static Person person`
`static Classifier classifier`
`static SCOP scopin, scopout, scopstat`
`static String player`
`private static boolean recognised(CircularBuffer buf, int windowsize, int framecounter)`

CircularBuffer

`CircularBuffer(int size, Frame init)`
`private Frame[] circ`
`private int ptr, size`
`void add(Frame f)`
`Frame get(int windowsize, int index)`

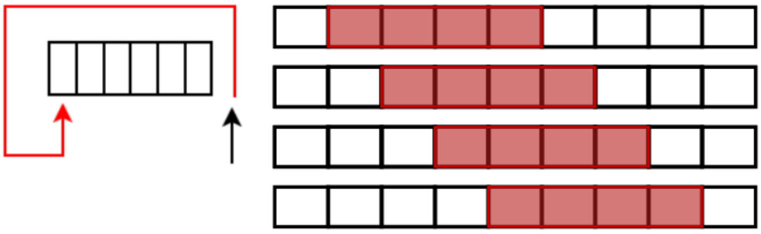

Figure 3.5: Accessing beyond the length of the circular buffer causes the index to wrap around to overwrite the beginning.

A circular buffer holds a list of frames and a pointer to indicate the next position for an incoming data frame. **SlidingWindow** performs preprocessing on incoming frames, adds them to the active position until the buffer is full, and from then on overwrites the oldest frames.

Every ten frames, **SlidingWindow** runs the specified recogniser on the circular buffer with a range of window sizes from 50 frames (0.5 seconds) to 350 frames (3.5 seconds). Trials show that almost all clearly defined gestures fall within these boundaries.

```
Loop forever:
  Get message from SCOP
  Add received frame to circular buffer
    for windowsize = 50 to 350, step size = 10
    if (command recognised in circular buffer window)
      emit the command to ctrlserver
```

If a gesture is recognised from the current window, the buffer continues to fill with arriving data but no more attempts at recognition are made until the minimum windowsize is exceeded. In addition, **SlidingWindow** monitors the frames for a flag indicating a dropout; if a frame contains data where at least one object is unavailable, it emits a “dropout” message to the `statusserver`, and “ok” when the object returns.

This status information is used by the Python class **feedback** to display two rectangles, for player 1 and player 2, as in figure 3.6. A red rectangle indicates that the data is currently unavailable, while a green rectangle indicates a good detection rate. This feedback was highly rated in my user study of the system, allowing users to distinguish between poor detection from the Vicon system and poor recognition from a recogniser.

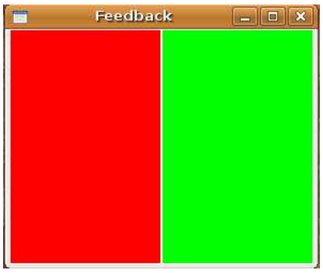

Figure 3.6: **feedback** listening to `p1status` and `p2status` and showing that `p1status` has reported a dropout.

3.2.4 Feature extraction

Neural networks have a fixed number of input nodes, but the feature vector has a variable number of frames. Therefore it is also necessary to perform feature extraction to create a fixed size set of variables to characterise the gesture. Performing feature extraction also reduces the search space for the recogniser, in order to increase the probability of successful matching. The gesture contains a lot of redundancy, since frames are taken at a high frame rate and show close temporal correlation.

Choosing the minimal features which extracted the most information to distinguish the gestures was an important step. Using the entire data for each gesture (6 data points * 3 objects * 300 frames/gesture = 5400 data values, for each example) would be equivalent to template matching and would require excessive processing times. Since the gestures were chosen to be axis aligned relative to the body, the distinguishing features are the range of dx,dy,dz values that each prototypical gesture may take, which is not sensitive to factors such as the person's height or their neutral posture.

Each gesture is symmetric and begins and ends with a neutral pose, so I defined a further "relocation" feature indicating a closed gesture: a double defining the sum of squares error of the distance moved from the beginning. Thus a gesture which is at the peak displacement will have a large relocation value, while a closed gesture which has returned to the starting pose has a small displacement value.

The seven features which are extracted are LeftArm $\{dx,dy,dz\}$ and RightArm $\{dx,dy,dz\}$, normalised to between 0 and 1, and *displacement*, the sum of squares error of the first and last frames, in millimetres.

Features
`Features(ArrayList<Frame> data)`
`Features(CircularBuffer buf, int windowsize)`
`Ranges leftarm, rightarm`
`double displacement`
`double calc_displacement(Frame first, Frame last)`
`void extract(double[] a)`

Ranges
`private double[] min, max`
`void update(SixDOF bodypart)`
`double get_delta(int axis)`

Features takes a list of **Frames** and holds two **Ranges** for left and right arms, and a double containing the displacement value calculated as follows:

```
double displacement = 0.0;
displacement += square(last.left.tx - first.left.tx);
displacement += square(last.left.ty - first.left.ty);
displacement += square(last.left.tz - first.left.tz);
displacement += square(last.right.ax - first.right.ax);
displacement += square(last.right.ay - first.right.ay);
displacement += square(last.right.az - first.right.az);
```

Ranges is a utility class which contains six values for the minimum and maximum of x, y and z.

3.3 Recognition

3.3.1 Recogniser

The **Recogniser** is a base class which defines an interface containing two method that all recognisers are expected to override, namely `train` and `recognise`.

Recogniser
`public static Gesture recognise(Person person, Features features)`
`public void train(ArrayList<Sample> samples, String out_file)`

Gesture, **Person** and **Classifier** classes are type-safe enumerations of the valid inputs. They provide accessor and mutator methods, and can be transformed into commands, strings, persons and so on.

Gesture
TurnLeft
TurnRight
Accelerate
Decelerate
StartStop
NoMatch
MultiMatch
Gesture(int cmd)
public String toAction()
public String toString()
static Gesture lookup(String name)
public boolean equals(Gesture g)

User
CHERYL
DAVID
User(int who)
private int userid
String name()
void set(int who)
Person get_person()
static String all_usernames()
static int lookup(String name)

Classifier
HEURISTIC
NEURAL
MARKOV
HYBRID
Classifier(int which)
private int id
String name()
void set(int which)
static String all_classifiers()
static int lookup(String name)
Gesture recognise(Person person, Features features)
void train(ArrayList<Sample> samples, String out_file)

The **Person** class is the union of all trained data related to a single specified user. It aggregates **Interval**s, neural network and hidden Markov model parameters, and also contains a constructor populating these fields from data files. It represents the saved behaviour of a **User**.

Person
`Intervals[] left, right`
`String neural_file`
`NeuralNet nnet`
`DirectSynapse netout`
`int neural_seq`
`String markov_root`
`Person()`

3.3.2 Heuristic recogniser

Heuristic extends Recogniser
`static double CLOSED_THRESHOLD`
`public static Gesture recognise(Person person, Features features)`
`private static boolean match(Person person, Features features, int gesture)`

Intervals
`private double[][] range`
`double get_min(int axis)`
`double get_max(int axis)`
`void setX(double from, double to)`
`void setY(double from, double to)`
`void setZ(double from, double to)`
`void stationary(int wobble)`

This application-specific recogniser depends strongly on the particular gestures chosen. It makes use of prior knowledge about the domain; specifically, patterns for how each gesture is characterised.

The first off-line training stage is to calculate all features on the training dataset, excluding outliers. The **Person** class defines the valid intervals of the specified user for the left and right ranges, as the maximum and minimum ranges that are permissible. The threshold for a closed gesture was experimentally determined to be around $3000mm^2$, but is editable in the configuration file.

When given a **Person** and **Features** calculated from a list of frames, the first check is that the displacement is less than the specified threshold. If this succeeds, the left and right **Range**s are compared with the **Person**'s **Interval**s. If the ranges fall within the permissible intervals, a **Gesture** is returned.

3.3.3 Neural network recogniser

Neural extends Recogniser implements NeuralNetListener
`public static Gesture recognise(Person person, Features features)`
`public void train(ArrayList<Sample> sampleslist, String out_file)`
`private void init_parameters()`
`static void saveNeuralNet(NeuralNet nnet, String filename)`
`static NeuralNet restoreNeuralNet(String filename)`
`void set_columns(MemoryInputSynapse syn, int first, int last)`
`public void errorChanged(NeuralNetEvent e)`
`public void netStarted(NeuralNetEvent e)`
`public void netStopped(NeuralNetEvent e)`
`public void netStoppedError(NeuralNetEvent e, String error)`
`public void cicleTerminated(NeuralNetEvent e)`

There are seven input nodes corresponding to the seven features, a variable number of nodes in the hidden layer and five output nodes indicating a single gesture. Each gesture is labelled with a vector of expected output; for example, an accelerate gesture is `(1,0,0,0,0)`, a Turn Right gesture is `(0,0,0,1,0)` and a non-gesture is `(0,0,0,0,0)`.

The output of this is the list of weights which can be saved to disk. The Joone library is structured using the callbacks `errorChanged`, `netStarted`, `netStopped`, `netStoppedError` and `cicleTerminated` [sic], which are defined in the **NeuralNetListener** interface.

Training can be done in one of three modes; the standard Backpropagation algorithm, batch mode, and Resilient Backpropagation, an alternative which only uses the sign of the gradient rather than the magnitude and thus generally converges much faster.

3.3.4 Hidden Markov Model recogniser

Markov extends Recogniser
`public static Gesture recognise(Person person, Features features)`
`public void train(ArrayList<Sample> samples, String out_file)`
`private static ArrayList<ObservationVector> toObservationVectors(ArrayList<Frame> frames)`
`static void save_hmm(Learner learner, String filename)`
`static Hmm restore_hmm(String filename)`
`private void init_parameters()`

Learner
`Learner(int states, int dimensions, int iterations, Gesture g)`
`void add_sequence(ArrayList<ObservationVector> seq)`
`void learnbw()`
`void learnkm()`

Each gesture is modelled by a different Markov process, so five hidden Markov models are created for the five gestures. This recogniser uses the raw data rather than the extracted data, since it is highly dependent on the temporal characteristics of the input vector.

Two learning algorithms are used, Baum-Welch (`learnbw()`) and K-Means clustering (`learnbw()`). The Baum-Welch iteratively improves on a starting hidden Markov model, and so one iteration of k-means is used to initialise the Baum-Welch training. Since the training sequences used are relatively long, a modification known as the Baum-Welch Scaled Learner is used, and the results reported as natural logarithms of probabilities to avoid underflow.

The recognition phase is an implementation of the Viterbi algorithm which finds the most likely state sequence, which corresponds to the most likely gesture issued by the user.

3.3.5 Decision logic

Hybrid extends Recogniser
`public static Gesture recognise(Person person, Features features)`
`public void train(ArrayList<Sample> samples, String out_file)`

This aggregates the results from each recogniser and uses a majority voting system to decide which, if any, are correct. The output of each recogniser is a single gesture, so any two recognisers matching is considered correct. The training method calls the `train` on the neural network and hidden Markov model recognisers, as training is not valid for the heuristic recogniser.

This is an application of the safety engineering technique of triple modular redundancy. If any one of the methods fails to recognise the gesture or comes to an incorrect decision, the voting logic can use the other two pattern recognition techniques to mask the failure.

3.3.6 Training

Training
`User user`
`Classifier classifier`
`Training(ArrayList<Sample> samples, Classifier clf)`

Sample
`String pathname`
`Gesture gesture`
`ArrayList<Frame>`
`Features feat`
`Sample(String pathname, Gesture g)`

The offline training phase is implemented with an instance of the **Training** class and a set of **Sample**s. The constructor calls the `train()` method on the specified classifier, and can be used as a command line tool on a directory of gestures. This can be used to train a classifier and save the results to a data file.

3.4 Framework

3.4.1 Simulated control

Two graphical user interfaces were created which simulated input (via mouse and keyboard) and output (turtle graphics) respectively for alternative feedback. These were completely decoupled and communicated only via the SRCF. Since these were independent from the rest of the system, they are written in Python using the Python SCOP library and Tkinter, the Python graphical libraries. I used the Model-View-Controller design pattern to separate display from control:

- **control** displays a control panel which accepts either keystrokes or mouse clicks and emits commands on either the `p1ctrl` or the `p2ctrl` streams.

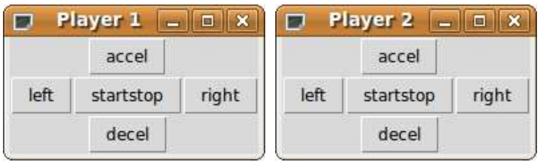

Figure 3.7: **control** emitting actions to `p1ctrl` and `p2ctrl`

- **view** listens to both streams and uses turtle graphics to represent the two players which can change in velocity or angle; an instance of **Arena** holds two instances of **Turtle** and directs the appropriate commands to update each turtle.

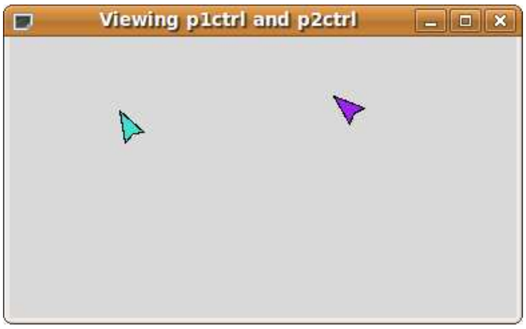

Figure 3.8: **view** listening to `p1ctrl` and `p2ctrl`

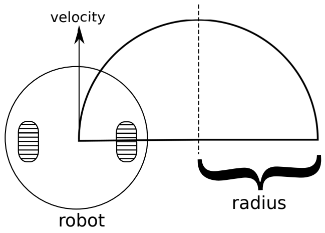

Figure 3.9: Using the turning radius to drive the robot's wheels.

3.4.2 Robot control

The **relay** class, running on the OLPC XO laptop, listens to either `p1ctrl` or `p2ctrl` and converts them into drive commands. It uses the Python implementation of SCOP to open a socket to SRCF:51234.

Since the laptops are separate from the desktop PC running all other components, the robot's player numbers are assigned from an environment variable. The initilisation of **relay** includes looking up the hostname on the laptop

Converting "Turn Left/Right" commands into a turning radii is shown in Figure 3.9 and is achieved by driving the two wheels on the robot at different speeds. An alarm is set for when the turn completes and the robot resumes straight line motion.

Accelerate and decelerates increment and decrement the velocity; turn left and right change the radius; and start/stop either performs an initialising acceleration, or sets the velocity to zero. The PyRobot library converts the Drive(velocity, radius) commands into 4 byte opcodes and operands. It serialises these for transmission over the USB-to-serial link in order to control the wheel motors.

A discrete turn command is transformed into a change of turning radius for a set period of time. Furthermore, since users issue commands asynchronously, a turn command may be issued while the robot is in the middle of a turn. For this case, the alarm is incremented by 0.5 seconds per turn instruction. Figure 3.10 demonstrates the algorithm to calculate the correct amount of turning time.

3.4.3 Networking

Decoupling various components allows message passing between different languages, which was important to allow the Java client to communicate with the Python robotics control. SCOP, a lightweight middleware framework, allows particularly simple events, messaging and RPC in C++, C, Java, Python and Scheme. SCOP hides the client and server setup

```
Update(speed, turn):
    Calculate robot velocity and radius parameters from speed and turn
    If speed $= 0$, simulates turning on spot with turning radius $= \epsilon$
Loop forever:
    If currently turning, check if alarm time has occured yet
        If so, cancel turn direction and call Update()
    Wait for input from SCOP, with a timeout of when the alarm will expire
    If timeout occurs first, cancel any current turn direction and call Update()
    Else if input arrives first, get message from SCOP
        Start/stop, Accelerate or Decelerate messages:
            Adjust speed appropriately and call Update()
        Turn left and Turn right messages:
            If not currently turning, record turn direction and set alarm for current time + 0.5 secs
            If currently turning, extend existing alarm until current time + 0.5 secs
            If currently turning in opposite direction, cancel turn
            Call Update()
```

Figure 3.10: Turn time algorithm

and silently discards data streams if there are no listeners. This makes it particularly simple to create and run such a distributed system by passing processed gesture data from the Java client to the Python robotics library.

SCOP assigns a name to each resource and an optional source hint to each stream. In order to listen to both players' commands, **Transform**, **control** and **view** open two sockets to listen to two independent streams. **relay** only listens to the stream of its user, either `p1ctrl` or `p2ctrl`. All other classes may listen to `p1` or `p2` as necessary.

The SRCF was used to provide a SCOP server running on a domain name, so that all units can reach it irrespective of whether they were wired or wireless and without knowing IP addresses. When testing on the King's College wifi network, it was found that broadcasting on non-standard ports is refused, including 51234. To compensate for this, **tunnel.sh** sets up ssh port forwarding to the SRCF, so that sockets opened on the OLPC XO appear as if connected from the SRCF's port 51234.

The three streams used for interprocess communication are `p1coords`, `p1ctrl` and `p1status` (and their equivalents for `p2`). In order to allow these to be distributed, three constants are defined in the configuration file: `coordserver`, `ctrlserver` and `statusserver`.

3.4.4 Configuration

Config
`public static String lookup(String key)`
`public static String set(String key, String value)`
`private Config()`
`private HashMap<String,String> dict`
`private static String lookup(String key)`
`private void supply_defaults()`
`private void check_add(String key, String defaultvalue)`
`private String do_lookup(String key)`

The **Config** class ensures that all processes with an interest in a stream are talking to the same server, and to allow the three streams to use different sockets as necessary. For example, the `coordsserver` stream is the most intensive (19/38 double floating point values at 100 fps), so by specifying "localhost", the overhead of TCP/IP network communication is reduced.

The <key, value> pairs are parsed from a file in the user's home directory and stored in a java.util.HashMap. If the file is not present or the values not defined, the class uses default values. In order to ensure that all classes read from the same configuration values, the **Config** class uses the Singleton design pattern. A static variable of type **Config** is set to null, and the first lookup initialises it from the configuration file. Subsequent requests only perform lookups on the instantiated object.

Chapter 4

Evaluation

The aim of this section is to compare the different recognisers in terms of accuracy and performance when recognising a set of gestures, some of which are previously unseen. Accuracy is calculated as a percentage of hits, misses, false positives (no match recognised as a gesture) and false negatives (any gesture not recognised).

The second evaluation metric is determining the optimal training parameters for neural networks and hidden Markov models. These are also testing using accuracy and performance, and additionally the neural networks report the final root mean squared error (RMSE) at the end of training.

4.1 Methodology

Evaluation
`static double error()`
`static double[] accuracy()`
`private static int match(Sample s, Gesture g)`
`static long performance()`
`private static void print_results()`

Result
`CORRECT`
`FALSE_POS`
`FALSE_NEG`
`INCORRECT`

The **Evaluation** class provides a recogniser-independent interface to the training and recognition APIs. It represents the union of all the legitimate combinations of modes (training, recognition with and without negative examples), criteria (performance, error, and accuracy) and classifiers (heuristic, neural, markov and hybrid), as well as interface to the **Config** file to set parameters. It also reads in a training.dat file and parses it for **Sample**s according to the type of evaluation that is performing; for example, recognition without negative examples only produces **Samples** for which the gesture is not a `NoMatch`. It uses **Result** to score the accuracy of recognition.

The python **eval** module provides a command line tool to iterate over a certain parameter (for example, momentum between 0.0 and 1.0 in steps of 0.1), appending the results to .dat files.

4.1.1 Hardware

The hardware used for performance measurements comprised a 2.8 Ghz Intel dual-core CPU with 1 Gb main memory, and 160 Gb storage (hard drive).

4.1.2 Defaults

Unless the parameter is under specific evaluation, the following values were otherwise supplied as defaults throughout:

Neural networks: Basic backpropagation, 1000 epochs, 20 nodes in hidden layer, learning rate = 0.8, momentum = 0.3, no training on negative examples

Hidden Markov models: Baum-Welch, 5 hidden states, 10 iterations

4.2 Experimental results

4.2.1 Comparison of gesture recognition methods

Figure 4.2 shows basic comparisons between each the four recognisers, based on accuracy and time taken to recognise a set of gestures. For this purpose, eleven gestures were reserved as unseen gestures, so that any training would be done on a subset of the recorded examples. Eighteen further were negative examples, and eighteen more normal gestures.

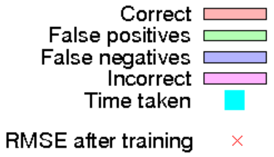

Figure 4.1: Key for all graphs. See Appendix D for tables.

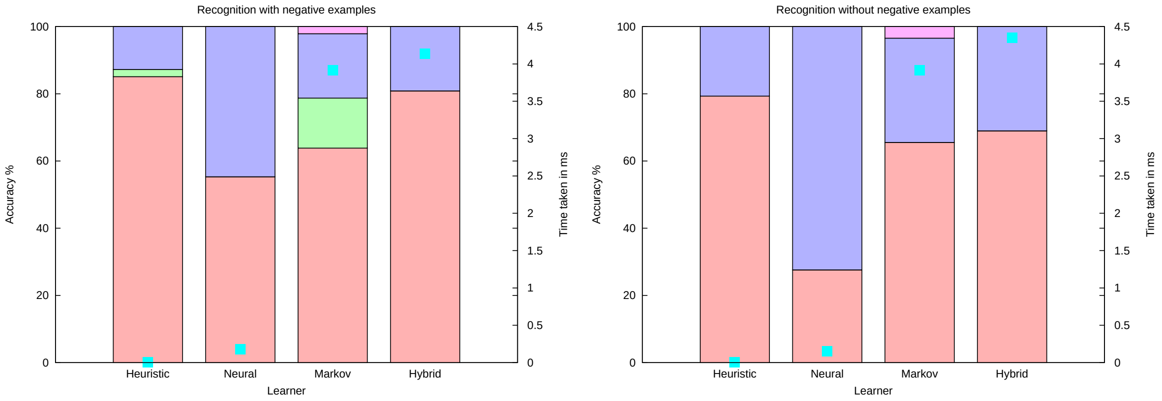

Figure 4.2: Comparison of properties of all the recognisers.

The data shows that the heuristic recogniser is by far the fastest and most accurate, which is unsurprising as the parameters are fine-tuned for the specific application. In comparison, neural networks are cheap but rather less accurate, especially when they have only been trained on known gestures. Hidden Markov models are expensive but provide good results, while the hybrid recogniser, being the aggregation of all the recognisers, takes the most computation and provides good results.

In terms of development, hidden Markov models are the most flexible as they do not require hand-chosen features. Also, neural networks and hidden Markov models generally cope better with outliers, compared to the heuristic recogniser which was trained with manual outlier removal.

4.2.2 Neural network training parameters

The neural networks have a number of variables which can be adjusted to obtain better results.

The graphs in figure 4.3 relate to a single training parameter; the first shows accuracy and performance data, as before, whilst the second shows the root mean squared error between the predicted results and the training examples, after training. A point to note is that the

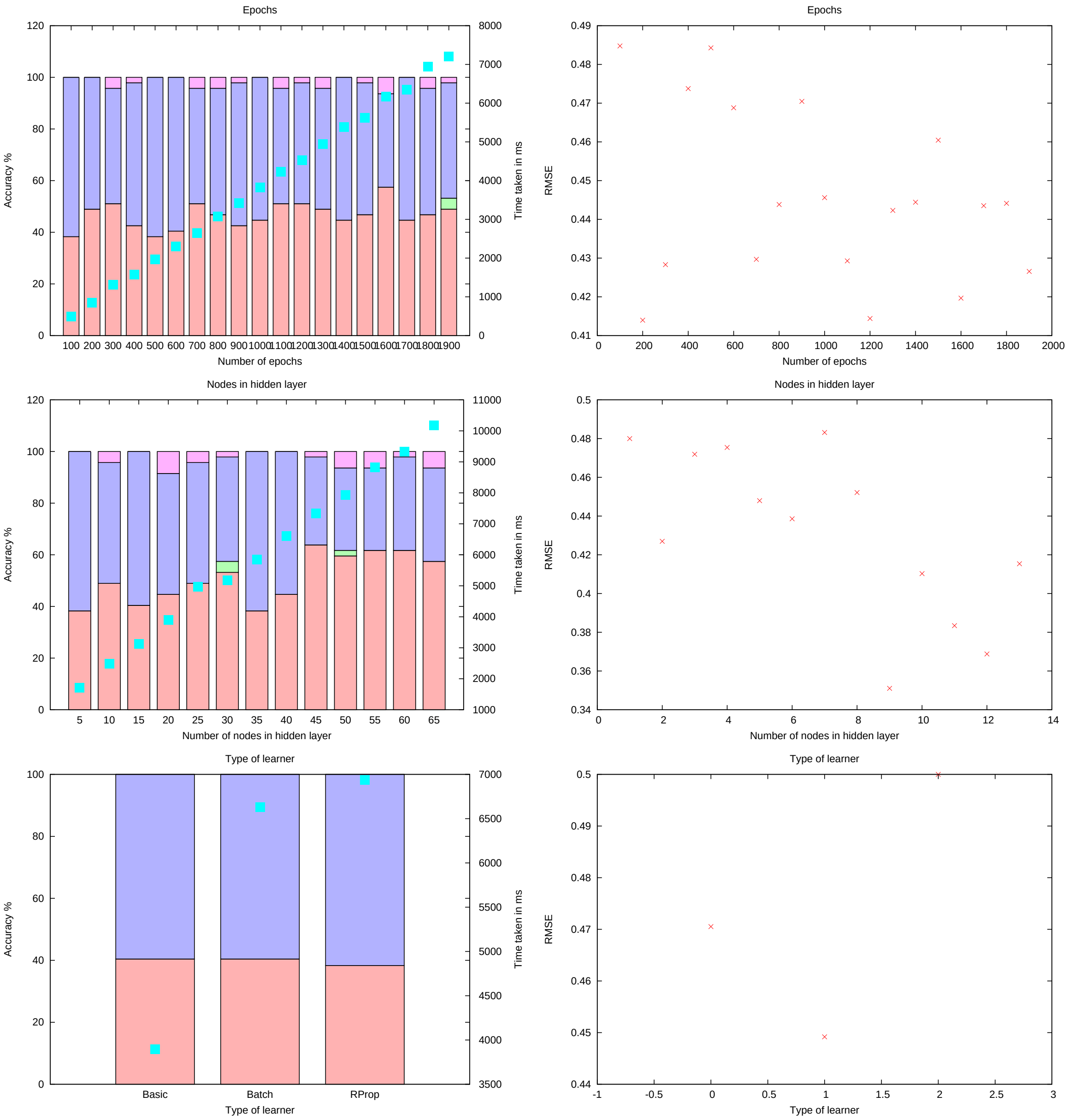

Figure 4.3: Comparison of number of epochs, number of nodes in the hidden layer and type of learner in the neural networks recogniser.

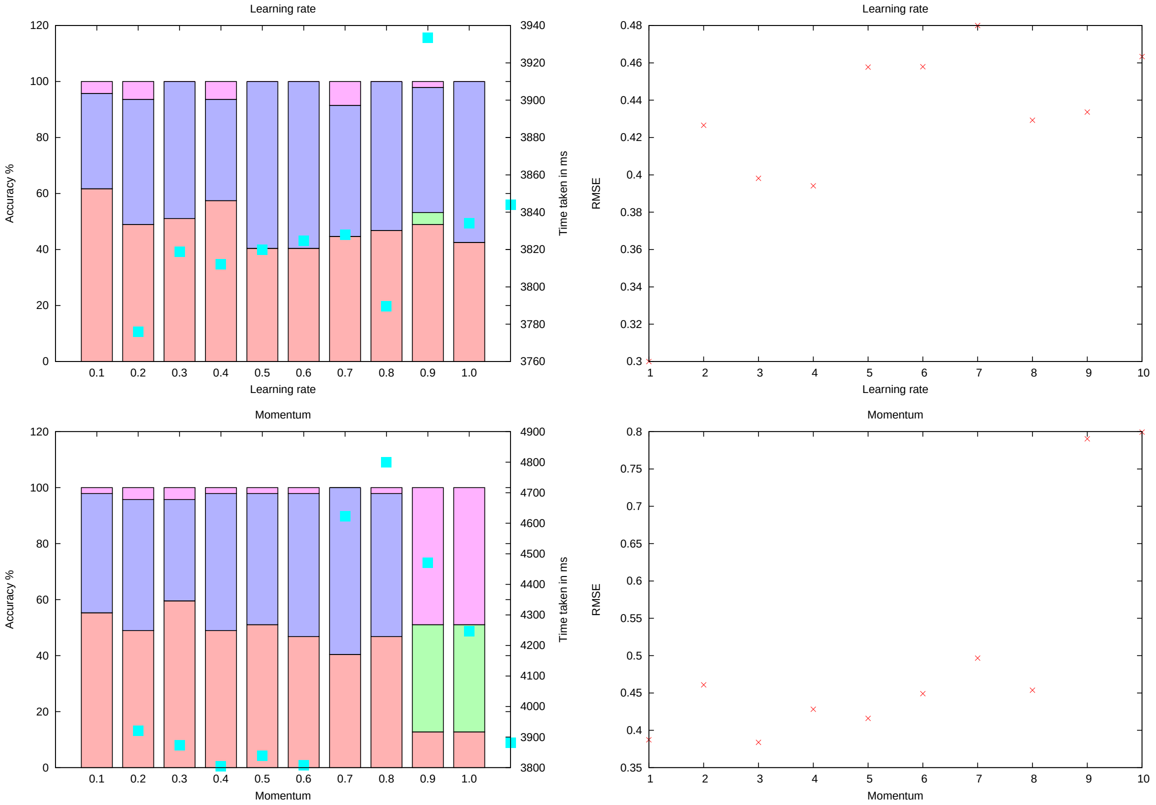

Figure 4.4: Comparison of learning rate and momentum in the neural networks recogniser.

timing now refers to the time taken to train the neural recogniser, rather than to recognise the gesture.

Although the number of epochs does not make much difference to the accuracy, it does somewhat reduce the RMSE in training. Varying the number of nodes in the hidden layer between 5 and 65 also shows a slight increase in accuracy and decrease in error. The most significant difference is the change from standard online backpropagation to batch propagation, which reduces the root mean squared error, but takes longer to train.

The neural network's internal training parameters may also be varied, as shown in 4.4. Setting a very low learning rate does increase accuracy, whilst setting the momentum too high means that the gradient descent oscillates excessively within the number of epochs specified, and so does not reach a good minimum. [1]

[1] A note to bear in mind throughout this presentation of results is that the neural network's initialisation step involves randomizing the weights, which means that the results are non-deterministic.

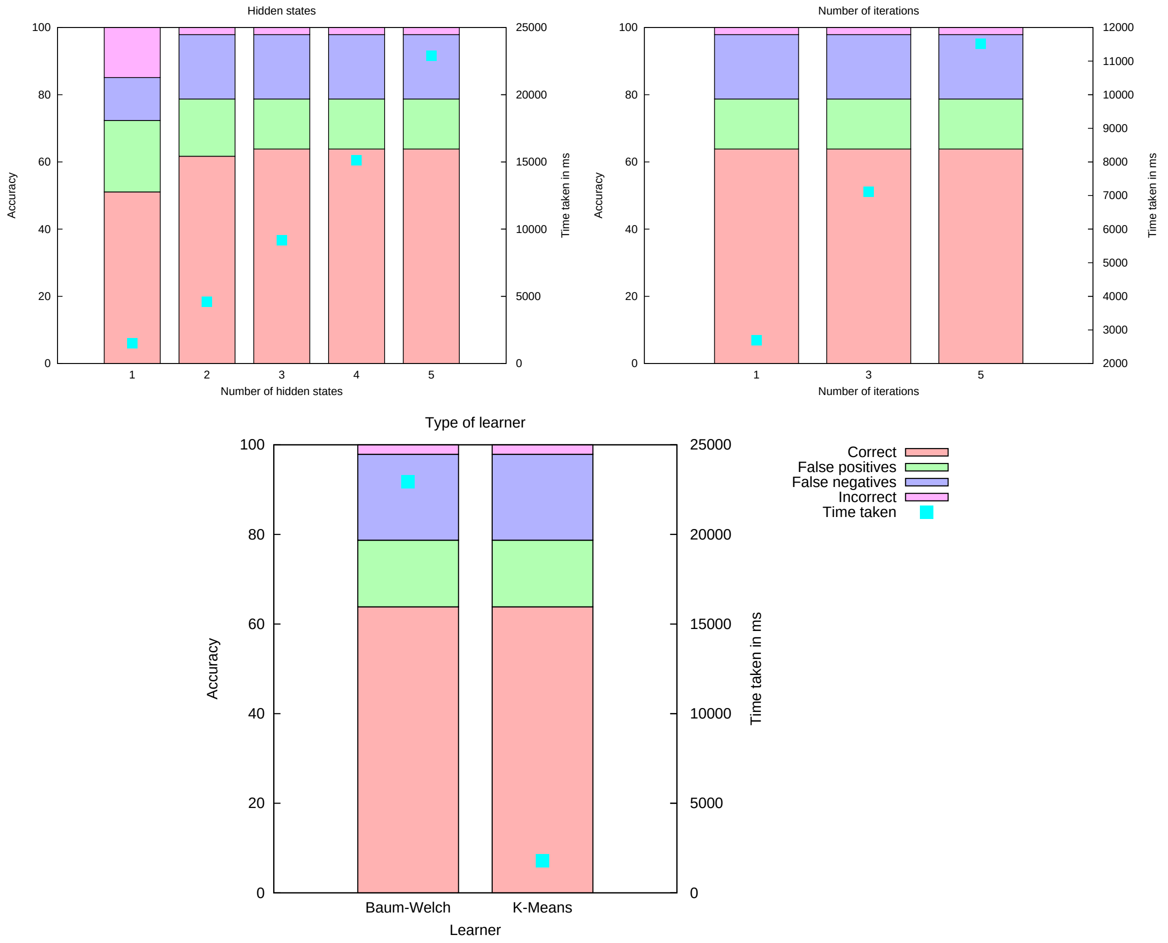

Figure 4.5: Comparison of number of hidden states, number of iterations and type of learner in the hidden Markov model recogniser.

4.2.3 Hidden Markov model training parameters

Hidden Markov Models may also be trained with different parameters, such as the number of hidden states in the recogniser and the number of iterations with which to run the Baum-Welch algorithm.

Figure 4.5 compares the results of varying these two parameters. The figure shows that a gesture can be successfully characterised in three states, which agrees with the expected result in that a gesture can be described in three phases; increasing displacement or distance from origin, no movement when the pose is held, and decreasing displacement. The time taken increases exponentially when increasing states or number of iterations, which is expected. The Baum-Welch algorithm performs equally as well as k-means clustering, but the k-means learner is faster and also does not require tagged examples, as it is an

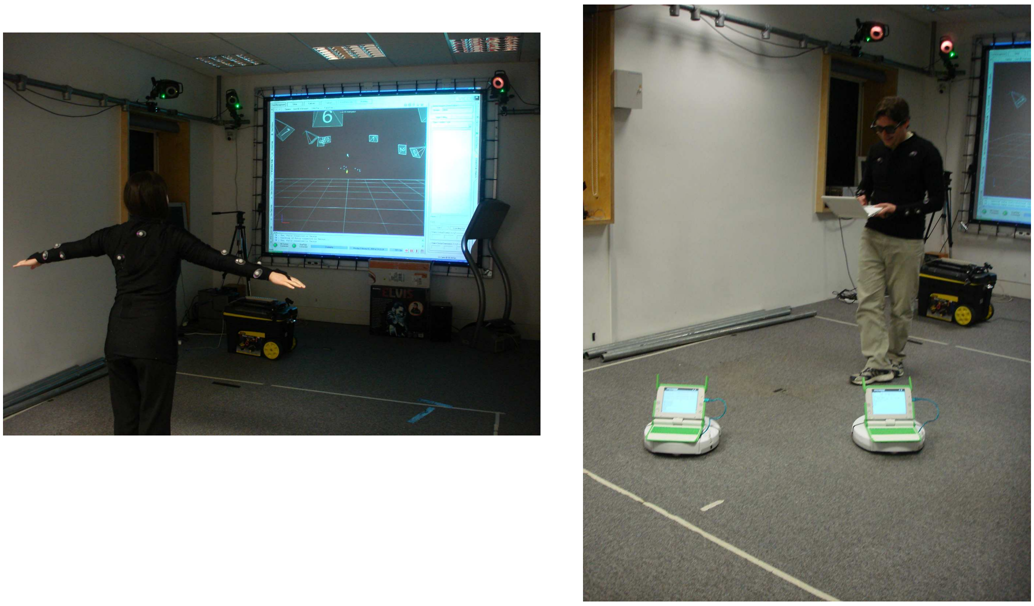

Figure 4.6: Photographs of demonstrations

unsupervised learning method.

4.2.4 Real time control

Figure 4.6 illustrates the alternative methods available for real time control; through arm gestures, or with a mobile device such as a laptop with Control.py running. The robots are connected wirelessly to the WGB network and are set to listen to a single player only, thus demonstrating the multiple user extension.

Figure 4.7 demostrates using the live system; the infra-red cameras are placed around the room, the user wears a black top with twelve markers and the screen projection shows both TARSUS reconstruction and the feedback window, which shows that player 2 is currently not in play.

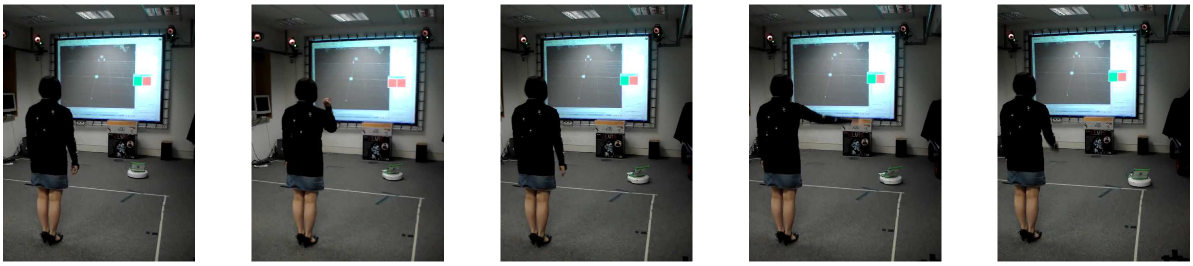

Figure 4.7: A sequence of stills showing the user issuing an Accelerate and a Turn Right command. The robot responds by moving towards the camera and changing angle.

4.3 Successes and failures

Overall, this project completed its original goals and surpassed them by also implementing extensions. It provides an end-to-end system for conversion of arm gestures into robot control with success rates of over 80%. In addition to the two originally planned recognisers, I implemented a further two recognisers and a full simulation system for every component of the system.

Chapter 5

Conclusions

5.1 Achievements

This project implements a working, complete system which has been demonstrated at the Computer Laboratory Rainbow group Outreach day and tried with multiple users. It combines fundamentally different techniques, and all parts are fully modularised and cleanly separated. This decoupling is a success of software engineering; almost all of the components are testing and can be used as standalone modules.

Integrating five libraries using two languages across three platforms was a major achievement of careful planning and understanding. The plan was completed to a budget, as stated in my project proposal, and the hardware obtained and succesfully integrated into the system. The recognition engine is usable in real-time, offering four classification techniques and 80% success rate with two users.

5.2 Reflection

I would like to have experimented with more input devices, as the Vicon system is expensive and time-consuming to set up. For example, accelerometers in the Nintendo Wii and many smartphones would make an ideal handheld controller.

5.3 Further work

Currently, the training and recognition is all performed on a single computer. Since the system is already decoupled, creating a distributed training environment would lead to an improvement in training and recognition times. This would be particularly straightforward given that the Joone library provides support for distributed training, and processing the five hidden Markov models can be done independently.

In the upcoming summer, this system will be used for educational purposes. The extensions which will be implemented in time for the Sutton Trust Summer School include a mobile application for control and writing a multiplayer game using the bump sensors from the robot. I hope that this will encourage others to experiment and interact with the system, testing its robostness and reliability.

Bibliography

[1] Euler angles image taken from wikipedia. `http://en.wikipedia.org/wiki/File:Eulerangles.svg`.

[2] Srcf, the student-run computing facility. `http://www.srcf.ucam.org/`.

[3] Vicon motion capture systems. `http://www.vicon.com/`.

[4] Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.

[5] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[6] J. R. Jr Deller. *Discrete Time Processing of Speech Signals*. IEEE Press, 2000.

[7] J-M. Francois. Jahmm, a java implementation of hidden markov model (hmm) related algorithms. `http://code.google.com/p/jahmm/`.

[8] Quintana G.E. Sucar L.E. Azcarate G. and Leder R. Qualification of arm gestures using hidden markov models, 2008.

[9] A. Glassner. *Graphics Gems*. Academic Press, 1990.

[10] D. Ingram. Scop, a library for straightforward distributed systems programming with events and messages in c++, java and python. `http://www.srcf.ucam.org/~dmi1000/scop/index.html`.

[11] iRobot Corporation. irobot create, open interface specifications. `http://www.irobot.com/filelibrary/pdfs/hrd/create/Create%20Open%20Interface_v2.pdf`.

[12] D. Kohler. Olpc telepresence. `http://www.instructables.com/id/OLPC-Telepresence/`.

[13] D. Kohler. Pyrobot, a high-level python interface to irobot's roomba and create. `http://code.google.com/p/pyrobot/`.

[14] J.-C. Lementec and P. Bajcsy. Recognition of arm gestures using multiple orientation sensors: gesture classification, 2004.

[15] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press, 1967.

[16] P. Marrone. Joone - java object oriented neural engine. `http://www.jooneworld.com/`.

[17] C. Morrison and A. Blackwell. *bodyPaint, a physical interface for exploring collaborative technologies*. PhD thesis, University of Cambridge, 2008.

[18] K. Nickel and R Stiefelhagen. Visual recognition of pointing gestures for human-robot interaction. *Image Vis. Comput.*, 2006.

[19] S. Russell and P. Norvig. *Artificial Intelligence; A Modern Approach*. Pearson Education Inc., 2003.

[20] Eric W Weisstein. Discrete fourier transform. `http://mathworld.wolfram.com/DiscreteFourierTransform.html`.

[21] Eric W Weisstein. Haar function. `http://mathworld.wolfram.com/HaarFunction.html`.

Appendix A

Code snippets from recognisers

A.1 Heuristic

```
class Heuristic extends Recogniser
{
   static double CLOSED_THRESHOLD = -1;

   public static Gesture recognise(Person person, Features features)
   {
      int command = Gesture.NoMatch;

      if(CLOSED_THRESHOLD < 0)
         CLOSED_THRESHOLD = Double.valueOf(Config.lookup("closedthreshold"));
      for(int i = 0; i < Gesture.num_gestures; i++)
      {
         if(match(person, features, i))
         {
            if(command == Gesture.NoMatch)
               command = i;
            else
               command = Gesture.MultiMatch;
         }
      }
      Gesture g = new Gesture(command);
      return g;
   }
   private static boolean match(Person person, Features features, int gesture)
   {
      Intervals left_range = person.left[gesture];
      Intervals right_range = person.right[gesture];
      double left_delta, right_delta;

      for(int axis = 0; axis < 3; axis++)
      {
         if(features.displacement > CLOSED_THRESHOLD)
            return false;

         left_delta = features.leftarm.get_delta(axis);
         if(left_range.get_min(axis) > left_delta ||
               left_range.get_max(axis) < left_delta)
            return false;

         right_delta = features.rightarm.get_delta(axis);
         if(right_range.get_min(axis) > right_delta ||
               right_range.get_max(axis) < right_delta)
            return false;
      }
      return true;
   }
};
```

A.2 Neural

```
class Neural extends Recogniser implements NeuralNetListener
{
   LinearLayer input;
   SigmoidLayer hidden, output;
   FullSynapse synapse_IH, synapse_HO;
   NeuralNet nnet;
   Monitor monitor;
   MemoryInputSynapse inputStream, samples;
   TeachingSynapse trainer;

   public static Gesture recognise(Person person, Features features)
   {
      double[] inputdata, outputdata;
      Pattern pin, pout;

      init(person); // Check person.nnet has been initialised

      inputdata = new double[Features.num_features];
      features.extract(inputdata);
      pin = new Pattern(inputdata);
      pin.setCount(person.neural_seq++);
      person.nnet.singleStepForward(pin);
      pout = person.netout.fwdGet();

      double[] a = pout.getArray();
      int command = Gesture.NoMatch;

      for (int i = 0; i < Gesture.num_gestures; i++)
      {
         if(a[i] > NEURAL_THRESHOLD)
         {
            if(command == Gesture.NoMatch)
               command = i;
            else
               command = Gesture.MultiMatch;
         }
      }

      return new Gesture(command);

   }

   void train(ArrayList<Sample> sampleslist, String out_file)
   {

      output_file = out_file;
      Sample samp;
      init_parameters();

      int num_samples = sampleslist.size();
      double[][] inputdata = new double[num_samples][Features.num_features];
      double[][] outputdata = new double[num_samples][Gesture.num_gestures];
      for(int i = 0; i < num_samples; i++)
      {
```

```
      samp = sampleslist.get(i);
      samp.feat.extract(inputdata[i]);
      for(int j = 0; j < Gesture.num_gestures; j++)
         outputdata[i][j] = 0.0;
      if(samp.gesture.command < Gesture.num_gestures)
         outputdata[i][samp.gesture.command] = 1.0;
   }

   input = new LinearLayer();
   hidden = new SigmoidLayer();
   output = new SigmoidLayer();
   input.setLayerName("input");
   hidden.setLayerName("hidden");
   output.setLayerName("output");
   input.setRows(Features.num_features);
   hidden.setRows(NUM_HIDDEN_NEURONS);
   output.setRows(Gesture.num_gestures);
   synapse_IH = new FullSynapse();
   synapse_HO = new FullSynapse();
   synapse_IH.setName("IH");
   synapse_HO.setName("HO");
   input.addOutputSynapse(synapse_IH);
   hidden.addInputSynapse(synapse_IH);
   hidden.addOutputSynapse(synapse_HO);
   output.addInputSynapse(synapse_HO);

   inputStream = new MemoryInputSynapse();
   inputStream.setInputArray(inputdata);
   set_columns(inputStream, 1, Features.num_features);
   input.addInputSynapse(inputStream);

   samples = new MemoryInputSynapse();
   samples.setInputArray(outputdata);
   set_columns(samples, 1, Gesture.num_gestures);

   trainer = new TeachingSynapse();
   trainer.setDesired(samples);
   output.addOutputSynapse(trainer);

   nnet = new NeuralNet();
   nnet.addLayer(input, NeuralNet.INPUT_LAYER);
```

```
      nnet.addLayer(hidden, NeuralNet.HIDDEN_LAYER);
      nnet.addLayer(output, NeuralNet.OUTPUT_LAYER);

      monitor = nnet.getMonitor();
      monitor.setTrainingPatterns(num_samples);
      monitor.setTotCicles(NUM_EPOCHS);
      monitor.setLearningRate(LEARNING_RATE);
      monitor.setMomentum(MOMENTUM);
      monitor.setLearning(true);

      //Add learner

      monitor.addLearner(0, "org.joone.engine.BasicLearner"); // On-line
      monitor.addLearner(1, "org.joone.engine.BatchLearner"); // Batch
      monitor.addLearner(2, "org.joone.engine.RpropLearner"); // RPROP

      monitor.setLearningMode(LEARNING_MODE);

      monitor.setSingleThreadMode(true);
      monitor.addNeuralNetListener(this);

      nnet.go();
   }
}
```

A.3 Markov

```
class Markov extends Recogniser
{

   ArrayList<Learner> learners;
   ArrayList<ArrayList<ArrayList<ObservationVector>>> sequences;

   public static Gesture recognise(Person person, Features features)
   {
      String filename;
      double[] probabilities = new double[Gesture.num_gestures];
      ArrayList<ObservationVector> framedata =
        toObservationVectors(features.rawdata);
      Hmm<ObservationVector> recog_hmm = new Hmm<ObservationVector>
        (5, new OpdfMultiGaussianFactory(features.rawdata.size()));

      for (int i = 0; i < Gesture.num_gestures; i++)
      {
         filename  = person.markov_root + new Gesture(i).toAction();
         recog_hmm = restore_hmm(filename);
         probabilities[i] = recog_hmm.lnProbability(framedata);
      }

      int command = Gesture.NoMatch;

      for (int i = 0; i < Gesture.num_gestures; i++)
      {
         if(!Double.isNaN(probabilities[i]))
         {
            if(command == Gesture.NoMatch)
               command = i;
            else
               command = Gesture.MultiMatch;
         }
      }

      return new Gesture(command);
   }

   void train(ArrayList<Sample> samples, String out_file)
   {
      init_parameters();
      output_root = out_file;

      for (Sample sample : samples)
      {
         Learner learner = learners.get(sample.gesture.command);
         learner.add_sequence(toObservationVectors(sample.data));
         Utils.log("Assigned " + sample.pathname +
          " to learner for " + learner.gesture.toString());
      }

      long timing = System.nanoTime();
      for (Learner learner : learners)
      {
         Utils.log("Training " + learner.gesture.toString());
         if (LEARNER == 0)
         {
            learner.learnbw();
         }
         else if (LEARNER == 1)
         {
            learner.learnkm();
         }
         else
         {
            Utils.error("Unknown learner; valid options are " +
               "0 for Baulm-Welch and 1 for K-Means");
         }

         save_hmm(learner, output_root + "_" + learner.gesture.toAction());
      }

        timing = System.nanoTime() - timing;

        Utils.log("Learner hidden-states iterations ms");
        Utils.results(LEARNER + " " + NUM_STATES +
                    " " + NUM_ITERATIONS + " " + timing/1000);
   }
};

class Learner
{
   Hmm<ObservationVector> hmm;
   ArrayList<ArrayList<ObservationVector>> sequences;
   BaumWelchScaledLearner bwl;
   KMeansLearner<ObservationVector> kml;
    OpdfMultiGaussianFactory factory;
    Gesture gesture;

   int num_states;
   int num_dimensions;
   int num_iterations;
   Learner(int states, int dimensions, int iterations, Gesture g)
   {
      num_states = states;
      num_dimensions = dimensions;
      num_iterations = iterations;
      gesture = g;

      factory = new OpdfMultiGaussianFactory(num_dimensions);
      hmm = new Hmm<ObservationVector>(num_states, factory);
      sequences = new ArrayList<ArrayList<ObservationVector>>();
   }

   void add_sequence(ArrayList<ObservationVector> seq)
   {
      sequences.add(seq);
   }

   void learnbw()
   {
      bwl = new BaumWelchScaledLearner();
      bwl.setNbIterations(num_iterations);
      //One iteration of KMeansLearner to initalise
      hmm = new KMeansLearner<ObservationVector>
                  (num_states, factory, sequences).iterate();
      hmm = bwl.learn(hmm, sequences);
   }

   void learnkm()
   {
      kml = new KMeansLearner<ObservationVector>
                  (num_states, factory, sequences);
      hmm = kml.learn();
   }
}
```

Appendix B

Simulated Gesture input

```
class SimGesture
{
   static ArrayList<RecordedGesture> read_gestures(String gesture_dir)
   {
      ArrayList<RecordedGesture> gestures;
      ArrayList<Frame> data;
      String filename;
      RecordedGesture record;
      File dir;
      File[] contents;

      gestures = new ArrayList<RecordedGesture>();

      dir = new File(gesture_dir);
      if(dir.exists() == false || dir.canRead() == false ||
            dir.isDirectory() == false)
      {
         Utils.error("Cannot read directory " + gesture_dir);
         System.exit(0);
      }
      contents = dir.listFiles();
      for(File f: contents)
      {
         if(f.getName().endsWith(".csv") == false)
            continue;
         filename = f.getPath();
         // System.out.println(filename);
         record = new RecordedGesture();
         record.data = GestureReader.getData(filename);
         record.filename = filename;
         gestures.add(record);
      }
      return gestures;
   }

   static double interpolate_angle(double from, double to, double w)
   {
      double v = 1 - w;
      double x1, y1, x2, y2, x, y;
      double theta;

      x1 = Math.sin(from) * v;
      y1 = Math.cos(from) * v;
      x2 = Math.sin(to) * w;
      y2 = Math.cos(to) * w;
      x = x1 + x2;
      y = y1 + y2;
      theta = Math.atan2(y, x);
      return theta;
   }

   static SixDOF interpolate_sixdof(SixDOF from, SixDOF to, double w)
   {
      // "w" is the weighting for the "to" body part (0.0 - 1.0)
      SixDOF sixdof = new SixDOF();
      double v = 1 - w;

      sixdof.ax = v * from.ax + w * to.ax;
      sixdof.ay = v * from.ay + w * to.ay;
      sixdof.az = v * from.az + w * to.az;
      sixdof.angle = interpolate_angle(from.angle, to.angle, w);

      sixdof.tx = v * from.tx + w * to.tx;
      sixdof.ty = v * from.ty + w * to.ty;
      sixdof.tz = v * from.tz + w * to.tz;
      return sixdof;
   }
```

```
static Frame interpolate_frames(Frame from, Frame to, double weight)
{
   // "weight" is the weighting for the "to" frame (0.0 - 1.0)
   Frame f = new Frame();

   f.body = interpolate_sixdof(from.body, to.body, weight);
   f.left = interpolate_sixdof(from.left, to.left, weight);
   f.right = interpolate_sixdof(from.right, to.right, weight);
   return f;
}

static void interpolate_gestures(SCOP scop, RecordedGesture from_gesture,
      RecordedGesture to_gesture, int duration)
{
   int num_frames;
   Frame from, to, f;

   Utils.log("Interpolating for " + duration + " ms");
   from = from_gesture.data.get(from_gesture.data.size() - 1);
   to = to_gesture.data.get(0);
   num_frames = (duration * FRAME_RATE) / 1000;
   for(int i = 0; i < num_frames; i++)
   {
      f = interpolate_frames(from, to,
            (double)(i + 1) / (double)(num_frames + 1));
      scop.emit(f.toString());

      framesync();
   }
}

static void framesync()
{
   framecounter++;

   long target_time = framecounter * 1000 / FRAME_RATE + start_of_stream;
   long current_time = System.currentTimeMillis();

   if(current_time >= target_time)
   {
      if(current_time > target_time + 100)
```

```
            Utils.log("Warning: SimGesture overload, lagging >10ms");
         return;
      }
      Utils.delay((int)(target_time - current_time));
   }

   static void replay_gesture(SCOP scop, RecordedGesture gesture)
   {
      Frame f;

      Utils.log("Replaying gesture ");
      Utils.results(gesture.filename);
      Utils.log(" (frames " + framecounter + " to " + (framecounter +
            gesture.data.size()) + ")");
      for(int i = 0; i < gesture.data.size(); i++)
      {
         f = gesture.data.get(i);
         scop.emit(f.toString());
         framesync();
      }

   }
}

class GestureReader
{
   static ArrayList<Frame> getData(String filename)
   {
      ArrayList<Frame> data = new ArrayList<Frame>();
      String line;
      int frameno = 0;
      SixDOF sixdof;
      char c;

      int bodypart = -1, startpart;

      try
      {
         BufferedReader in = new BufferedReader(new FileReader(filename));
         while((line = in.readLine()) != null)
         {
            if(line.length() == 0)
               continue;
            startpart = -1;
            c = line.charAt(0);
            if(c >= ’0’ && c <= ’9’)
            {
               sixdof = parse(line);
               while(frameno > data.size() - 1)
                  data.add(new Frame());
               if(bodypart == Body)
                  data.get(frameno).body = sixdof;
               else if(bodypart == LeftArm)
                  data.get(frameno).left = sixdof;
               else if(bodypart == RightArm)
                  data.get(frameno).right = sixdof;
               else
                  Utils.error("Frame data received for unknown body part");
               frameno++;
            }
            else if(line.startsWith("Belt") || line.startsWith("Hat") ||
                  line.startsWith("Body"))
               startpart = Body;
            else if(line.startsWith("Left"))
               startpart = LeftArm;
            else if(line.startsWith("Right"))
               startpart = RightArm;
            else
               Utils.error("Invalid line in data file.");

            if(startpart >= 0)
            {
               // Start new body part:
               bodypart = startpart;
               frameno = 0;
               if((line = in.readLine()) == null)
                  Utils.error("Unexpected end of file.");
            }
         }
         in.close();
      }
      catch(IOException e)
      {
         System.out.println("Cannot read from <" + filename + ">");
         System.exit(0);
      }
      Frame last = data.get(data.size() - 1);
      if(last.body == null || last.left == null || last.right == null)
         Utils.error("Different number of frames for different body parts.");
      return data;
   }
};
```

Appendix C

Gesture segmentation

```
class SlidingWindow
{
   public static void main(String[] args)
   {
      final int BUFFER_SIZE = 500;
      final int MIN_WINDOW = 50, MAX_WINDOW = 350, WINDOW_STEP = 10;
      CircularBuffer buf;
      msg = scopin.get_message();
      f = new Frame(msg);
      Transform.process(f);
      buf = new CircularBuffer(BUFFER_SIZE, f);
      start_time = System.currentTimeMillis();

      while(true)
      {
         before_block = System.nanoTime();
         msg = scopin.get_message();
         after_block = System.nanoTime();
         blocked_time += after\_block - before\_block;
         if(msg == null)
            Utils.error("Error reading message from SCOP");

         f = new Frame(msg);
         Transform.process(f);
         buf.add(f);
         framecounter++;
```

```
dropout = f.dropout();
if(dropout != dropped\_out)
{
   String status;
   if(dropout)
      status = "dropout";
   else
      status = "ok";
   scopstat.emit(status);
   Utils.log("New status: " + status);
   dropped_out = dropout;
}

if(framecounter % 1000 == 0)
{
   double fps, cpu;

   current_time = System.currentTimeMillis();
   elapsed_time = current_time - start_time;
   cpu = (double)blocked_time / 1000000.0 / (double)elapsed_time;
   cpu = 100.0 * (1.0 - cpu);
   fps = (double)(framecounter - laststatus) / (double)elapsed_time
         * 1000.0;
   Utils.log("FPS = " + String.format("%.1f", fps)
        + " CPU utilisation = " + String.format("%.1f", cpu) + "%");

   start_time = current_time;
   blocked\_time = 0;
   laststatus = framecounter;
}
if(framecounter % WINDOW_STEP == 0)
{
   int availableframes = framecounter - lastrecognition;

   for(int windowsize = MIN_WINDOW; windowsize <= MAX_WINDOW &&
         windowsize <= availableframes; windowsize += WINDOW_STEP)
   {
      if(recognised(buf, windowsize, framecounter) == true)
      {
         lastrecognition = framecounter;
         break;
```

```
               }
            }
         }
      }

      scop.close();
   }
}

class GestureReader
{
   static SixDOF parse(String s)
   {
      String[] values;
      SixDOF sixdof = new SixDOF();

      values = s.split(",");
      if(values.length != 7)
      {
         System.out.println("Parse error on line <" + s + "> (" +
               values.length + " values)");
         System.exit(0);
      }
      sixdof.ax = Double.valueOf(values[1]) * Math.PI / 180.0;
      sixdof.ay = Double.valueOf(values[2]) * Math.PI / 180.0;
      sixdof.az = Double.valueOf(values[3]) * Math.PI / 180.0;
      sixdof.tx = Double.valueOf(values[4]);
      sixdof.ty = Double.valueOf(values[5]);
      sixdof.tz = Double.valueOf(values[6]);
      sixdof.ax *= Math.PI / 180.0;
      sixdof.normalise();
      return sixdof;
   }

   static final int Body = 0;
   static final int LeftArm = 1;
   static final int RightArm = 2;

   static ArrayList<Frame> getData(String filename)
   {
      ArrayList<Frame> data = new ArrayList<Frame>();
```

```
String line;
int frameno = 0;
SixDOF sixdof;
char c;

int bodypart = -1, startpart;

try
{
   BufferedReader in = new BufferedReader(new FileReader(filename));
   while((line = in.readLine()) != null)
   {
      if(line.length() == 0)
         continue;
      startpart = -1;
      c = line.charAt(0);
      if(c >= ’0’ && c <= ’9’)
      {
         sixdof = parse(line);
         while(frameno > data.size() - 1)
            data.add(new Frame());
         if(bodypart == Body)
            data.get(frameno).body = sixdof;
         else if(bodypart == LeftArm)
            data.get(frameno).left = sixdof;
         else if(bodypart == RightArm)
            data.get(frameno).right = sixdof;
         else
            Utils.error("Frame data received for unknown body part");
         frameno++;
      }
      else if(line.startsWith("Belt") || line.startsWith("Hat") ||
            line.startsWith("Body"))
         startpart = Body;
      else if(line.startsWith("Left"))
         startpart = LeftArm;
      else if(line.startsWith("Right"))
         startpart = RightArm;
      else
         Utils.error("Invalid line in data file.");
```

```
            if(startpart >= 0)
            {
               // Start new body part:
               bodypart = startpart;
               frameno = 0;
               if((line = in.readLine()) == null)
                  Utils.error("Unexpected end of file.");
            }
         }
         in.close();
      }
      catch(IOException e)
      {
         System.out.println("Cannot read from <" + filename + ">");
         System.exit(0);
      }
      Frame last = data.get(data.size() - 1);
      if(last.body == null || last.left == null || last.right == null)
         Utils.error("Different number of frames for different body parts.");
      return data;
   }
};
```

Appendix D

Tables of results

Correctly recognised gestures, false positives, false negatives and incorrect recognitions are given in %.

D.1 All recognisers

D.1.1 Recognition rates including negative gestures

Recogniser	Correct	False positives	False negatives	Incorrect	Time in ms
Heuristic	85.106	2.128	12.766	0.000	2148
Neural	55.319	0.000	44.681	0.000	180550
Markov	63.830	14.894	19.149	2.128	3914882
Hybrid	80.851	0.000	19.149	0.000	4136745

D.1.2 Recognition rates excluding negative gestures

Recogniser	Correct	False positives	False negatives	Incorrect	Time in ms
Heuristic	79.310	0.000	20.690	0.000	2094
Neural	27.586	0.000	72.414	0.000	154373
Markov	65.517	0.000	31.034	3.448	3912585
Hybrid	68.966	0.000	31.034	0.000	4348238

D.2 Neural network parameters

D.2.1 Epochs

Epochs	Time in ns	Error	Correct	False pos	False neg	Incorrect
100	494014532	0.484754	38.298	0.000	61.702	0.000
200	852160050	0.413959	48.936	0.000	51.064	0.000
300	1310019711	0.428305	51.064	0.000	44.681	4.255
400	1569497171	0.473745	42.553	0.000	55.319	2.128
500	1963321175	0.484276	38.298	0.000	61.702	0.000
600	2303271715	0.468796	40.426	0.000	59.574	0.000
700	2641930678	0.429680	51.064	0.000	44.681	4.255
800	3069397925	0.443833	46.809	0.000	48.936	4.255
900	3418706638	0.470450	42.553	0.000	55.319	2.128
1000	3825216682	0.445603	44.681	0.000	55.319	0.000
1100	4230667724	0.429279	51.064	0.000	44.681	4.255
1200	4521389172	0.414404	51.064	0.000	46.809	2.128
1300	4943561402	0.442314	48.936	0.000	46.809	4.255
1400	5382021039	0.444414	44.681	0.000	55.319	0.000
1500	5619347221	0.460444	46.809	0.000	51.064	2.128
1600	6172803597	0.419664	57.447	0.000	36.170	6.383
1700	6343792158	0.443532	44.681	0.000	55.319	0.000
1800	6939302410	0.444124	46.809	0.000	48.936	4.255
1900	7202249519	0.426561	48.936	4.255	44.681	2.128

D.2.2 Number of neurons in the hidden layer

#neurons	Time in ns	Error	Correct	False pos	False neg	Incorrect
5	1704171340	0.479992	38.298	0.000	61.702	0.000
10	2484331946	0.426971	48.936	0.000	46.809	4.255
15	3127442384	0.471925	40.426	0.000	59.574	0.000
20	3898959784	0.475446	44.681	0.000	46.809	8.511
25	4975425019	0.447978	48.936	0.000	46.809	4.255
30	5182903552	0.438601	53.191	4.255	40.426	2.128
35	5853244268	0.483151	38.298	0.000	61.702	0.000
40	6601236191	0.452157	44.681	0.000	55.319	0.000
45	7328486162	0.351041	63.830	0.000	34.043	2.128
50	7928302619	0.410273	59.574	2.128	31.915	6.383
55	8820263051	0.383438	61.702	0.000	31.915	6.383
60	9336418384	0.368782	61.702	0.000	36.170	2.128
65	10175352804	0.415397	57.447	0.000	36.170	6.383

D.2.3 Learner

Backpropagation	Time in ms	Error	Correct	False pos	False neg	Incorrect
Standard	3894112	0.470532	40.426	0.000	59.574	0.000
Batch	6631781	0.449182	40.426	0.000	59.574	0.000
Resilient	6936377	0.500000	38.298	0.000	61.702	0.000

D.2.4 Learning rate

Learning rate	Time in ns	Error	Correct	False pos	False neg	Incorrect
0.1	3775847140	0.303077	61.702	0.000	34.043	4.255
0.2	3818757272	0.422700	48.936	0.000	44.681	6.383
0.3	3811971495	0.395833	51.064	0.000	48.936	0.000
0.4	3819905742	0.392023	57.447	0.000	36.170	6.383
0.5	3824802315	0.452159	40.426	0.000	59.574	0.000
0.6	3827846415	0.452357	40.426	0.000	59.574	0.000
0.7	3789670337	0.473228	44.681	0.000	46.809	8.511
0.8	3933531427	0.425247	46.809	0.000	53.191	0.000
0.9	3833952291	0.429401	48.936	4.255	44.681	2.128
1.0	3843970392	0.457509	42.553	0.000	57.447	0.000

D.2.5 Momentum

Momentum	Time in ns	Error	Correct	False pos	False neg	Incorrect
0.1	3920458256	0.387309	55.319	0.000	42.553	2.128
0.2	3873086034	0.460942	48.936	0.000	46.809	4.255
0.3	3804137815	0.383810	59.574	0.000	36.170	4.255
0.4	3838913256	0.428254	48.936	0.000	48.936	2.128
0.5	3808166121	0.415957	51.064	0.000	46.809	2.128
0.6	4622423029	0.449144	46.809	0.000	51.064	2.128
0.7	4800556495	0.496654	40.426	0.000	59.574	0.000
0.8	4470688803	0.453686	46.809	0.000	51.064	2.128
0.9	4246731707	0.790327	12.766	38.298	0.000	48.936
1.0	3882328899	0.799305	12.766	38.298	0.000	48.936

D.3 Hidden Markov model parameters

D.3.1 Number of hidden states

# of hidden states	Time in ns	Correct	False pos	False neg	Incorrect
1	1509917951	51.064	21.277	12.766	14.894
2	4581457629	61.702	17.021	19.149	2.128
3	9167806732	63.830	14.894	19.149	2.128
4	15113765856	63.830	14.894	19.149	2.128
5	22899169657	63.830	14.894	19.149	2.128

D.3.2 Number of iterations

# of iterations	Time in ns	Correct	False pos	False neg	Incorrect
1	2695617669	63.830	14.894	19.149	2.128
3	7112320489	63.830	14.894	19.149	2.128
5	11522937273	63.830	14.894	19.149	2.128

D.3.3 Learner

Learner	Time in ns	Correct	False pos	False neg	Incorrect
Baum-Welch	22927365907	63.830	14.894	19.149	2.128
K-Means	1799183626	63.830	14.894	19.149	2.128

Appendix E

Project Proposal

Computer Science Tripos Part II Project Proposal

Gesture-controlled Robotics using the Vicon Motion Capture System

C. J. Hung, King's College

Originator: C. J. Hung

24 October 2008

Special Resources Required:

Vicon Motion Capture System

Two iRobot Creates and OLPC XO laptops

Project Supervisor: Cecily Morrison, Laurel Riek

Signatures:

Director of Studies: Simone Teufel

Signature:

Project Overseers: Graham Titmuss & Markus Kuhn

Signatures:

Introduction and Description of the Work

The objective of this project is to provide real-time gesture-based control of robotics. Gestures can be characterized as temporally evolving data, so by exploiting the same techniques used in speech recognition, a stream of coordinates from the Vicon motion capture system can be used to drive a small robot.[1]

Since the motion capture system has the benefit of capturing full body data, I intend to explore recognizing gestures whilst moving around in 3 dimensional space. This could allow the user to control two more parameters by their location.

The project will implement two different machine learning techniques for gesture recognition. The two approaches can be summed up as the 'Connectionist Approach', ie. Artificial Neural Networks, and the 'Stochastic Approach', ie. Hidden Markov Models. Both are well researched and have many applications in pattern recognition.

Direct evaluation will be based upon the success rate, using a common test set of recorded motion capture samples, and speed/efficiency, since the robot must be driven in real time. The evaluation will also take the cost of training into account, using the same training data for ANN of different number of hidden nodes and HMM with different number of states.

In addition, the system should support multiple users, each controlling a single robot (limited to two for the physical robots). Interaction between robots is supported by the touch sensors for a simple collision based game.

Resources Required

The Vicon motion capture system will be used to collect data from gestures. Since most of the work can be done offline, initial work will include collecting and archiving multiple sets of data for training and testing purposes.

My current intention is to use a OLPC XO mounted upon an iRobot Create. This choice was based on several contributing factors, including ease of programming, ease of assembly and cost. In particular, the XO laptop handles wireless connectivity and includes a webcam.[2] I have secured all necessary funding from the Women@CL Outreach programme and college for buying two robots. The itinerary for each robot:

[1] However, this is not a hardware project; existing interfaces to the Vicon system and the robot will be used.

[2] This setup is based on http://www.instructables.com/id/OLPC-Telepresence/, which includes a Python library for driving the robot and laptop.

- XO laptop: approx £120 second hand, second hand from Ebay
- UK - US adapter: £5
- iRobot Create: \$130 = £75
- USB to serial adapter: £10

Total: £210 per robot[3]

I intend to write a virtual robot which uses the same interface for initial testing. In case that the hardware fails or is unavailable, the project will be continued using the virtual robots only.

Starting Point

I have previously used Python for a small summer project. The PyRobot library interfaces with the Create's motors and sensors as well as the OLPC's webcam, which means that integration with the robots should be fairly hassle free. The Vicon motion capture system has a Java interface.

Substance and Structure of the Project

The initial setting up will include writing a virtual robot and recording training and test data from the Vicon system.

The bulk of the project will be implementing the two gesture recognition schemes, Hidden Markov Models and Artificial Neural Networks. My intention is to write my own implementation of these using Python, a high level scriping language, rather than using a pre-existing package such as the MATLAB Neural Network Toolbox or Hidden Markov Model Toolkit (HTK). This will ensure that external influences will be minimized when it comes to comparing the two approaches.

The recognized gesture will be mapped to the robot controls, which will be sent wirelessly to the XO laptop. The gestures will most likely include:

- Start/stop (for the Create and webcam)

[3] One Wi-Fi adapter for the main computer (if needed): £10

- Accelerate
- Decelerate/Reverse
- Turn left
- Turn right

Hidden Markov Models

The gestures can be modelled as a Markov Chain, a discrete-time process where future states are only dependant on the present state. However, since the underlying state is not visible, a learning algorithm together with a training set must be used to find the most probable state and parameter probability distributions. The most commonly used technique is the Baum-Welch Procedure, a generalized expectation-maximization algorithm which uses the forward-backward algorithm.

Artificial Neural Networks

An alternative machine learning technique is the neural network, based on a set of interconnected nodes or neurons. In this paradigm the feed-forward network, specifically the multilayer perceptron, typically consists of three layers (input, hidden and output) where each neuron employs a non-linear activation function. The well known backpropagation algorithm will be used for supervised learning.

By using the same training set and test data as the Hidden Markov Model, I will be able to compare and contrast the different approaches in terms of success rates and efficiency.

In addition, I also intend to investigate gestures moving through 3 dimensional space, to allow users to walk around while performing a gesture. An initial approach would be to calculate arm gestures relative to something that represents the user's location eg. a belt, then to subtract the effect of the user moving around.

Once the models are working with the virtual robot, I will implement an extension for multiple users, each controlling a robot in a simple collision-based game. Finally the switch to using real robots will move to using the touch sensors and streaming video from the XO laptop's webcam.

Criterion for Success

By the end of the project, the following goals should be completed:

1. Implement the Baum-Welch algorithm for training the Hidden Markov Model.
2. Implement a feedforward neural network and backpropagation algorithm.
3. Train with a set of approximately five gestures.
4. Demonstrate that a virtual robot can be driven using these gestures in real time.

Timetable and Milestones

7 November, two weeks

One week each of studying Hidden Markov Models and Neural Networks. Decide on gestures. Investigate Jython for Python-Java interoperability.
Milestone: Write Introduction.

21 November, two weeks

Record training and test data from the Vicon motion capture system. Write a virtual robot and test.
Milestone: Write Preparation.

19 December, four weeks

Implement feedforward neural networks and backpropagation algorithm. Test using archived data and virtual robot.
Milestone: Write first half of Implementation.

23 January, five weeks

Implement the Baum-Welch algorithm for HMM. Test using archived data and virtual robot.
Milestone: Write second half of Implementation.

30 January, one week

Milestone: Write progress report.

13 February, two weeks

Integration with real time data. Switch to physical robots.

27 February, two weeks

3D gestures using filtering. Add multiuser facilities.

13 March, two weeks

Write a simple collision game using the Create touch sensors. Stream OLPC webcam to projector.

20 March, one week

Final online integration and testing.

10 April, three weeks

Milestone: Complete Implementation. Write Evalution.

24 April, two weeks

Milestone: Completed Dissertation.

8 May, two weeks

Proof read, Latex, bind.
Milestone: Submit Dissertation by 15 May.