Cheryl Hung

Gesture-controlled Robotics using the Vicon Motion Capture System

Computer Science Tripos

King's College

May 12, 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:	**wordcount**
Project Originator:	Cheryl Hung
Supervisor:	Cecily Morrison, Laurel Riek

Original Aims of the Project

To utilise the machine learning techniques of neural networks and hidden Markov models for the purposes of gesture recognition. The input device chosen was the motion capture system, and the arm gestures used to drive robots.

Work Completed

All 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.

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

Finally, I have implemented the multiplayer extension.

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

New paradigms for human-computer interaction are becoming a reality. Amongst consumer goods, new ways of interaction are forcing a rethink 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 such as corporate events and restaurants.

The philosophy which is common to these systems is defined by the term "ubiquitous computing". The aim is to allow intuitive, natural interaction so that anybody (in theory) can use them without training, but instead by mirroring or mimicking behaviour of the real world.

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

While machine vision is one option for solving this problem, motion capture systems provide far more detailed location data, down to hundredths of a millimeter. Rather than allowing users full freedom of gestures, I chose to train on a finite set of predefined gestures, converting these into commands that are used to drive the robot.

The two computational models used internally for this translation are Artificial Neural Networks and Hidden Markov Models. Gesture recognition is really a two-phase problem; the training with prerecorded datasets and the run-time interpretation.

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. 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.

2.2 Hardware

The main requirements of the hardware were:

1. It should have a range of commands it is willing to accept, in order to allow training of multiple gestures. A simple vehicle sufficed for this purpose.
2. It should have wireless capabilities for independant movement, and sensors and webcam for the collision based game.
3. A high level API or library is preferred, for ease of development.
4. Commercially available robots makes purchasing for multiuser applications easier.
5. High battery life and low power consumption, for testing purposes.
6. 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 and testing for range of movement etc. 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, 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 reference: OLPC Telepresence also provided a Python library for interfacing with both the laptop's webcam and robot's motors and bump sensors.

Command module?

diagram: robot

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 @ \$220.00 = £151.86 OLPC XO laptop @ \$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.

2.3 Motion Capture System

The Vicon Motion Capture system in the Rainbow Group is controlled by Tarsus. This software collates the data from the 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 needed firstly for input to a gesture, and secondly representing the user's position so that gestures can be calculated invariant to translation, rotation and spatial

Figure 2.1: Left to right: arm gestures for *accelerate*, *decelerate*, *turn left*, *turn right*, and *start/stop*.

displacement. 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, giving better recognition rates from the Vicon system. Options for positional data were belt, hat and body. 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).

diagram: markers

The data outputted is six data values per object; an Axis Angle triplet for rotation and a triplet for global translation in mm. 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 last 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.

2.3.1 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 following rotation matrix is used to rotate around an arbitrary axis where (x,y,z) is a unit vector on the axis of rotation, and theta is the angle of rotation.

reference: Graphics Gems

(x, y, z)

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

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

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

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

In the 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, the Euler system is subject to a phenomenon known as gimbal lock, since not all points in the coordinate system can be uniquely identified. 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.

2.4 Pattern recognition

There are many different techniques 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. However given the set of gestures are fixed for this application, the algorithms for supervised learning have higher accuracy ratings so unsupervised learning will not be considered further.

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 classifier respectively.

2.4.1 Neural Networks

An artificial neural network is an interconnected set of nodes which individually perform simple processing, but 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 non-linearly given sufficient input. Common choices for the activation function are the tanh function or the sigmoid function, which have the property of being differentiable:

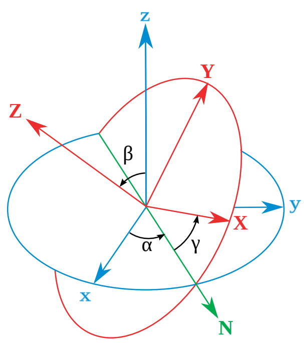

Figure 2.2: Euler Angles

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

diagram: single neuron

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, where the overall structure is feed-forward and there are three layers of nodes; an input layer, a hidden layer and an output layer.

diagram: multilayer perceptron

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:

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

The backpropagation algorithm is an application of gradient descent for minimization of error. 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 wi and adjust the weight vector to lower 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

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

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*[2]:

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

The importance of this result is the resulting reduction in time complexity, as training neural networks is very time consuming. The majority of the work is done calculating $a_{ij} = \sum_i w_{ji} z_i$. There are W derivatives to calculate for W, letting W be the total number of weights and biases. The naive approach of explicit evaluating each derivative 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 $O(W)$.

I researched several neural network libraries, including Joone 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.

2.4.2 Hidden Markov Models

First order Markov processes are a class of statistical model which state that the probability of being in some future state is only dependant on the current state, also known as the memoryless model since the "memory" or past states have no effect on the next state.

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

where $S_{0:t-1} = (S_0, S_1, \ldots S_{t-1})$

diagram: hidden markov models

At each time t, a symbol is emitted from state i. For a Bakis HMM, at $t+1$ the only options are to stay in state i or move to state $i+1$. 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, it is unknown which state the observation lies in. To find the probabilities, the Baulm-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 - the use of memoization to save 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.[3] 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:

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, $\operatorname{argmax}_{s_{1:t}} Pr(S_t|e|1:t)$

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

$$\begin{aligned}
\max_{s_{1:t}} Pr(s_{1:t}, St+1|e1:t+1) &= c \max_{s_{1:t}} Pr(S_{t+1}, s_t) Pr(et+1|St+1) Pr(s_{1:t}|e_{1:t}) \\
&= cPr(et+1|St+1) \max_{s_t} \left(Pr(S_{t+1}|s_t) \max_{s_{1:t-1} Pr(s_{1:t-1}|e_{1:t}}) \right)
\end{aligned}$$

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 of 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.

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, giving the probability of the best sequence reaching this state
return maximum of the probabilities, following the pointers back to find most likely sequence

The Baum-Welch algorithm performs the training, analogous to the backpropagation phase of the neural networks.[1]

Matrix multiplication to calculate f Re-estimation EM Pg 731

Alternative Java libraries for Hidden Markov Models include Jahmm, jHMM, and HmmSDK.

2.5 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.

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. Haar wavelets are an alternative method which also uses linear combinations of wavelet functions, which can represent discontinuities better than the Fourier method.

However these techniques were deemed excessively complex for the purposes of gesture recognition with a finite set of planar gestures. The simpler method of extracting axis-aligned ranges is sufficient for distinguishing the gestures without adding the computational overhead of these other techniques.

2.6 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.

2.6.1 Tools

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

Joone: Java Object Oriented Neural Engine
Jahmm: Java Hidden Markov Model
SCOP: Server COmmunication Protocol
PyRobot: robot and laptop control, motors, sensors and webcam
Tarsus: motion capture and object reconstruction

This dissertation was typeset using LaTeXand bibtex. The graphs shown in the evaluation were generated using `gnuplot`.

2.6.2 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. Cecily Morrison's client and both pattern recognition engines are Java. Message passing of simple string data is handled by SCOP, which has ports to both Python and Java.

Since the OLPC XO runs a modified version of Fedora Linux, the Tarsus system runs on Windows XP and the development environment was Ubuntu Linux, using architecture independent languages makes it easy to perform processing on a more powerful computer to 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 double 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.

3.1.1 Component Interfaces

All SCOP messages are ASCII string messages. 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 containing `p1` have `p2` equivalents.

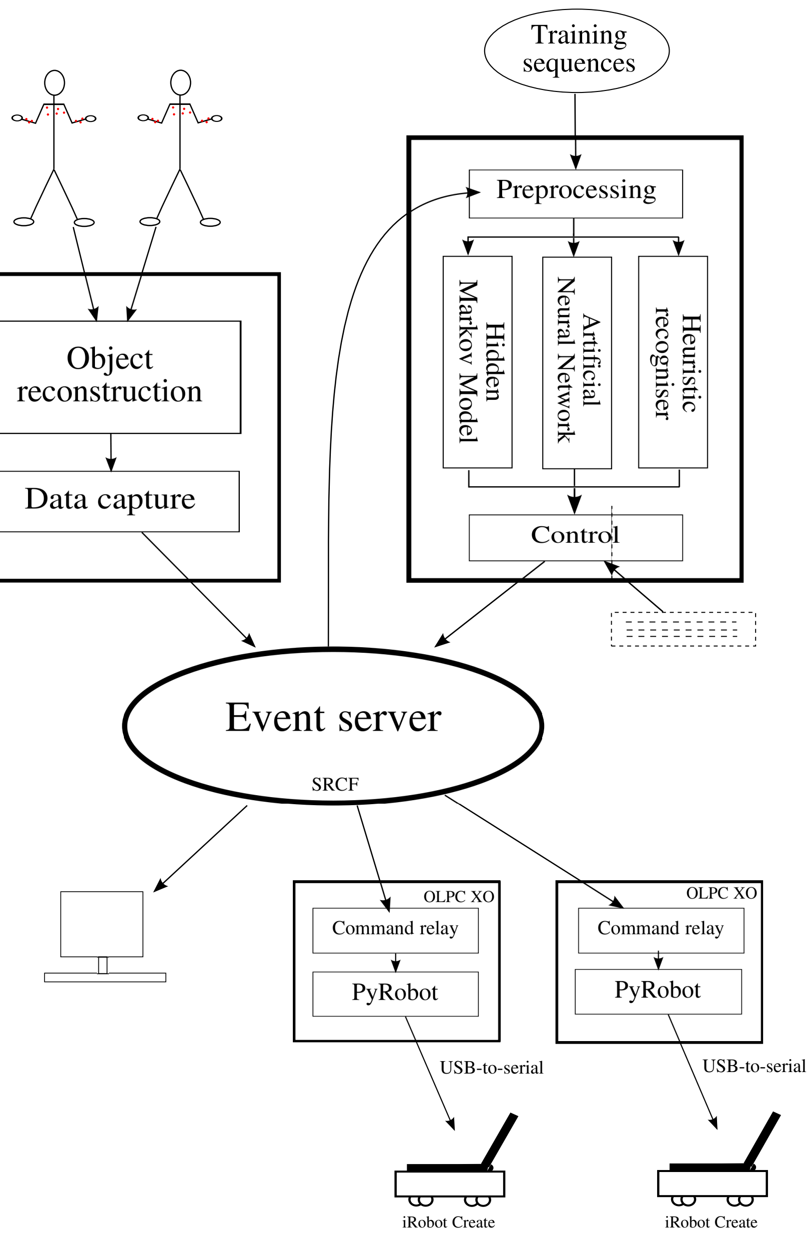

Figure 3.1: System Diagram

3.1.2 Protocols and data formats

	Endpoint	Event sources	Event sinks
coordsserver	p1coords	capturep1, simulatep1	windowp1

	The coordsserver transmits both players' stream of coordinates, labelled p1coords and p2coords. The data format consists of eighteen floating point values, followed by a string expressing the result of filtering the data for a body part dropping 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 to accurate to six significant figures. 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.		

ctrlserver	p1ctrl	controlp1, windowp1	viewp1, relayp1
	The ctrlserver handles the output from gestures, as single lower case characters representing a single 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. Commands are typically issued around once per second.		
statusserver	p1status	windowp1	feedbackp1
	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">		

3.1.3 Breakdown of components

Class diagrams diagram: class diagram

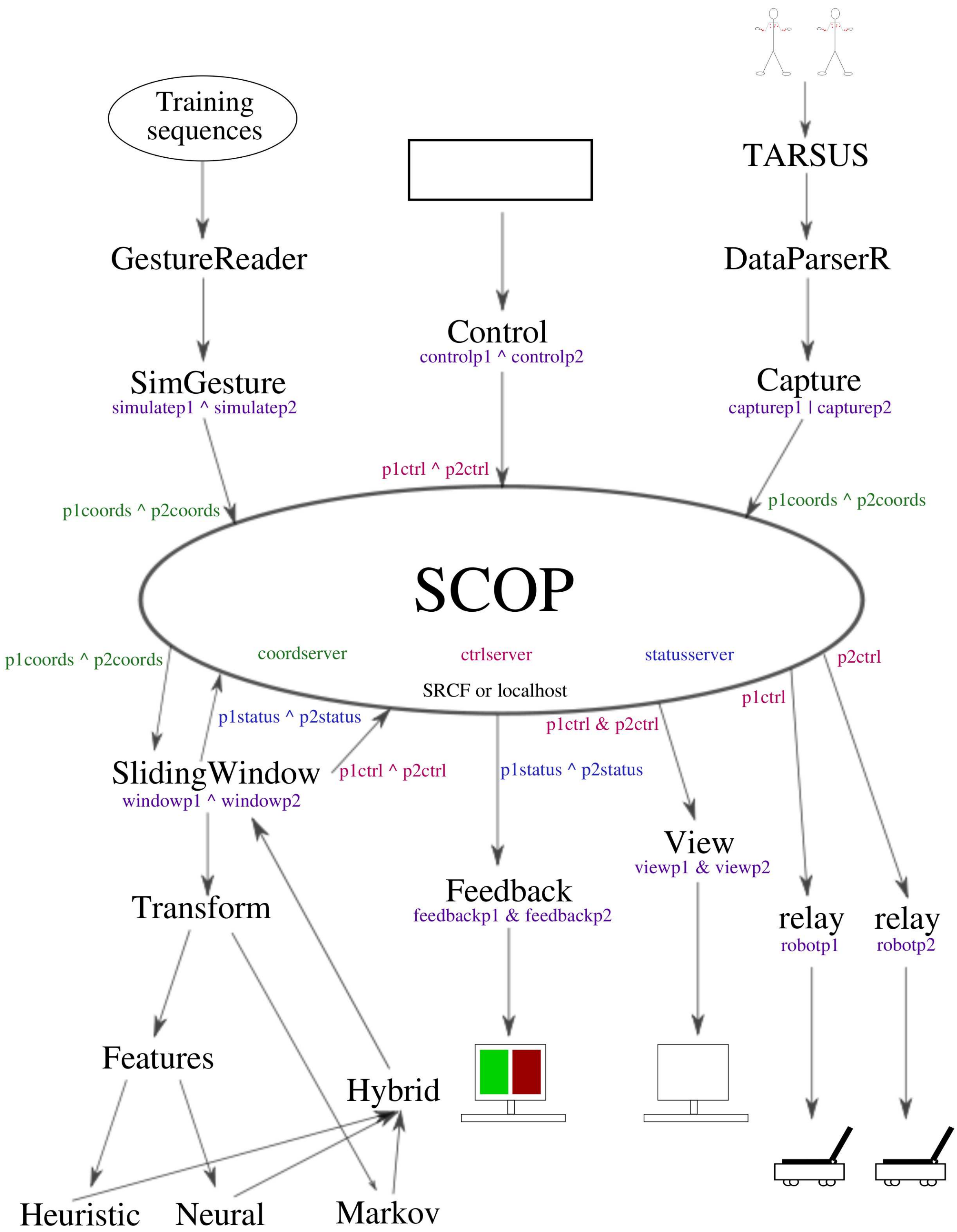

Figure 3.2: Data flow diagram

3.2 Motion capture

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

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

The Tarsus system performs the following processing in real time on the raw camera data:

Triangulation of twelve marker positions (using a variable number of between 3 and 10 cameras) in three dimensional space Interpolation and smoothing of the marker data, to reduce jitter Pairwise comparison between sets of the marker angles and three skeletons of defined features Using the notation <A-X, A-Y, A-Z, T-X, T-Y, T-Z>, it send the rotation and translation data for each object. If some markers are occluded or there is insufficient rigidity in the angles for object reconstruction, T-X, T-Y and T-Z are held at their last known position, and A-X, A-Y, and A-Z are set to 0.0.

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.

The Capture class uses DataParserR.java, a Java client provided by Cecily Morrison, 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 framerate (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, transcribed as valid data values from an object which often have an improbably high rate of change and return to the correct values shortly.

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. For example, a glitch could be :

-1, -2, -1, 53, 168, 168, 89, -4, -3, 0, ...

whereas 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 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 five gestures, giving a total of twenty five gestures taken from each session, recorded by two people. 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, so that the pattern recognition engine can give negative examples.

The data is formatted as CSV files. All values are double precision floating point numbers to six significant figures. Rotations are in the range $\pm 180°$ and translations are in mm from the origin in a z-up world.

```
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,
```

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

SimGesture
`static ArrayList<RecordedGesture> read_gesture(String gesture_dir)`
`+ static void interpolate_gestures(SCOP scop, RecordedGesture from_gesture, RecordedGesture to_gesture, int duration)`
`- static Frame interpolate_frames(Frame from, Frame to, double weight)`
`- static SixDOF interpolate_sixdof(SixDOF from, SixDOF to, double w)`
`- static double interpolate_angle(double from, double to, double w)`
`static void framesync()`
`+ static void replay_gesture(SCOP scop, RecordedGesture gesture)`
`+ static void main(String[] args)`

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. 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 appends the dropout boolean, in the same way as Capture.

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.java file together with the Geometry.java file, which defines geometric classes for SixDOF, Frame, and Point.

SixDOF
`double ax, ay, az, angle, ty, ty, tz`
`static final double EPSILON = 1.0e-5`
`SixDOF(double[] a, int offset)`
`+ void normalise()`
`+ 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 contains the following code, which 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.

```
double angle = 1.0
angle *= Math.sqrt(ax * ax + ay * ay + az * az)
if (angle > EPSILON):
    ax /= angle;
    ay /= angle;
    az /= angle;
else:
    angle = 0.0;
    ax = ay = 0.0;
    az = 1.0
```

Frame
`double x,y,z`
`static final double EPSILON = 1.0e-5`
`static Point rotatePoint(double ax, double ay, double az, double angle, double x0, double y0, double z0)`

The Frame class contains three of these SixDOFs for body, left arm and right arm.

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

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

Sliding window

SlidingWindow
static User user
static Person person
static Classifier classifier
static SCOP scopin, scopout, scopstat
static String player
+ static void main(String[] args)
- static boolean recognised(CircularBuffer buf, int windowsize, int framecounter)

The raw gesture data consists of a vector 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.

diagram: Circular buffer

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 window sizes of 50 frames (0.5 seconds) to 350 frames (3.5 seconds), with trials showing 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 an "dropout" message to the `statusserver`, and "ok" when the object returns.

This status information is used by feedback.py to display two rectangles, for player 1 and player 2. 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 a user study, allowing users to distinguish between poor detection from the Vicon system and poor recognition from a recogniser.

diagram: feedback.py

3.2.4 Feature extraction

Since neural networks have a fixed number of input nodes, but the feature vector has a variable number of frames, 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 high speed 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, the distinguishing features are the range of dx,dy,dz values that each prototypical gesture may take. However, since each gesture is symmetric and begins and ends with a neutral pose, I defined a further 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 neutral pose has a small displacement value.

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)

Features.java contains a utility class, Ranges, containing six values for the minimum and maximum of x, y and z. The 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 mm. It 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);
```

3.3 Recognition

3.3.1 Recogniser

Recogniser
`+ static Gesture recognise(Person person, Features features)`

The Recogniser.java file holds defines the Recogniser interface containing a single method that all recognisers are expected to override.

The file also holds Gesture, Person and Classifier classes which are enumerations of the valid inputs:

Gesture
`TurnLeft`
`TurnRight`
`Accelerate`
`Decelerate`
`StartStop`
`NoMatch`
`MultiMatch`

User
`CHERYL`
`DAVID`

Classifier
`HEURISTIC`
`NEURAL`
`MARKOV`
`HYBRID`

The Person class is the union of all trained data related to a single specified user. It aggregates Intervals, neural network and Hidden Markov Model parameters, and also contains static methods to populate these fields from data files.

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

3.3.2 Heuristic recogniser

Heuristic extends Recogniser
`  static double CLOSED_THRESHOLD`
`+ static Gesture recognise(Person person, Features features)`
`- static boolean Match(Person person, Features features, int gesture)`

Intervals
`- 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 3000, 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 Ranges are compared with the Person's Intervals. If the ranges fall within the permissible intervals, a Gesture is returned.

3.3.3 Neural network recogniser

Neural extends Recogniser

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

There are six input nodes for LeftArm dx,dy,dz and RightArm dx,dy,dz, a layer of hidden nodes and five output nodes corresponding to the five gestures. 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).

3.3.4 Hidden Markov Model recogniser

Markov extends Recogniser

```
+ static Gesture recognise(Person person, Features features)
+ void train(ArrayList<Sample> samples, String out_file)
- static ArrayList<ObservationVector> toObservationVectors(ArrayList<Frame>
frames)
static void save_hmm(Learner learner, String filename)
static Hmm restore_hmm(String filename)
- 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 dependant on the temporal characteristics of the input vector.

3.3.5 Decision logic

Hybrid extends Recogniser
`+ static Gesture recognise(Person person, Features features)`
`+ 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 a Training class and a set of Samples. The constructor calls the `train()` method on the specified classifier, and can be used as a command line tool on a directory of gestures.

3.4 Framework

3.4.1 Simulated control

Two graphical user interfaces was 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.py displays a control panel which accepts either keystrokes or mouse clicks and emits commands on either the `p1ctrl` or the `p2ctrl` streams.

Figure 3.3: Control.py emitting actions to `p1ctrl` and `p2ctrl`

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

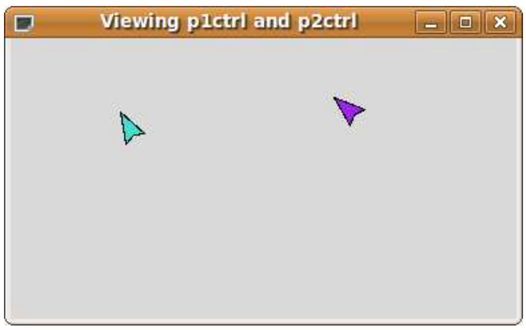

Figure 3.4: View.py listening to `p1ctrl` and `p2ctrl`

reference: Berkley

3.4.2 Robot control

diagram: relay.py

Relay.py uses the Python implementation of SCOP to open a socket to SRCF:51234. Each robot runs a separate instance of **relay.py** and listens to either `p1ctrl` or `p2ctrl`, depending on which one is assigned to which user.

Since the laptops are separate from the desktop PC running all other components, the robot's player numbers are assigned from an environment variable.

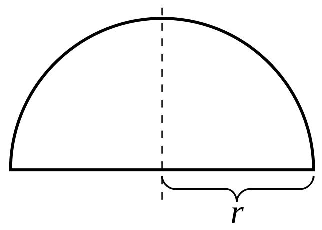

Figure 3.5: Turning radius

`Update(speed, turn)`:
Calculate robot `velocity` and `radius` parameters from `speed` and `turn`
If `speed` = 0, simulates turning on spot with turning radius = ϵ
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 3.6: Turn time algorithm

There were two options when deciding about drive commands; accelerate/decelerate vs drive forwards/backwards for a set time. Accelerate/decelerate with continual motion was chosen to avoid users having to continually issue commands to their robot.

This meant that turning should also be a smooth turn, rather than stop/turn in place/start. Converting a "Turn Left/Right" command into a turning radii is shown in figure 3.5.

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, an alarm is set for when the turn completes and the robot resumes straight line motion, and the alarm is incremented by 1 second per turn instruction.

Accelerate and decelerates increment and decrement the velocity; turn left and right increment and decrement 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.

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 written by Dr D Ingram, allows particularly simple events, messaging and RPC written in C++ with ports to C, Java, Python and Scheme. SCOP hides the client and server setup 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 gesture data from the Java client through Python processing with Java libraries to the Python robot 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.java, control.py and view.py open two sockets to listen to two independent streams. Relay.py only listens to the stream of its user, either `p1ctrl` or `p2ctrl`.

`p1coords` and `p2coords` represent the raw input streams from two users. `p1ctrl` and `p2ctrl` are the a,d,l,r,s commands as interpreted from the two users.

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 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
`- Config()`
`- static String lookup(String key)`
`- void supply_defaults()`
`- void check_add(String key, String defaultvalue)`
`- String do_lookup(String key)`

The Config.java file 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 the following default values:

framerate = 100
closedthreshold = 3000
`coordserver` = `www.srcf.ucam.org`
`ctrlserver` = `www.srcf.ucam.org`

In order to ensure that all classes read from the same configuration values, the Config.java file 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

4.1 Methodology

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

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 Samples 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.

The python eval.py 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 processing units used for performance measurements were 1Gb Intel dual-core main memory, and 160 Gb storage (hard drive).

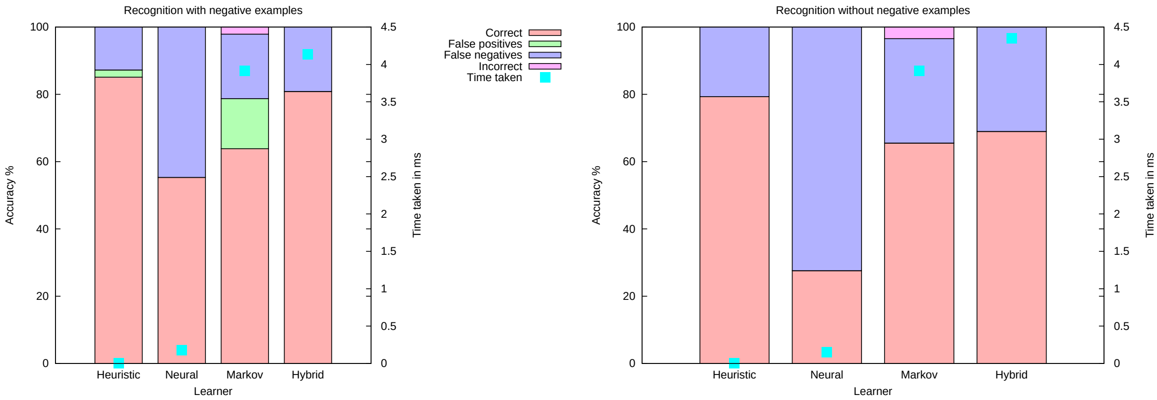

Figure 4.1: Comparisons of properties of all the recognisers.

4.1.2 Caveats

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.

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

Neural networks Learner = Basic backpropagation Epochs = 1000 Nodes in hidden layer = 20 Learning rate = 0.8 Momentum = 0.3 Do not train on negative examples

Hidden Markov models Learner = Baum-Welch Number of hidden states = 5 Iterations = 10

4.2 Comparison of gesture recognition methods

Figure 4.1 shows basic comparisons between each the four recognisers, based on time taken to recognise a set of gestures and accuracy. 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.

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 and the comparing ranges method is cheap. In comparison, neural networks are cheap but rather less accurate, especially when it has only been trained on known gestures. Hidden Markov models are very expensive but provide good results, while the hybrid recogniser, being the aggregation of all the recognisers, takes the most computation and provides good results.

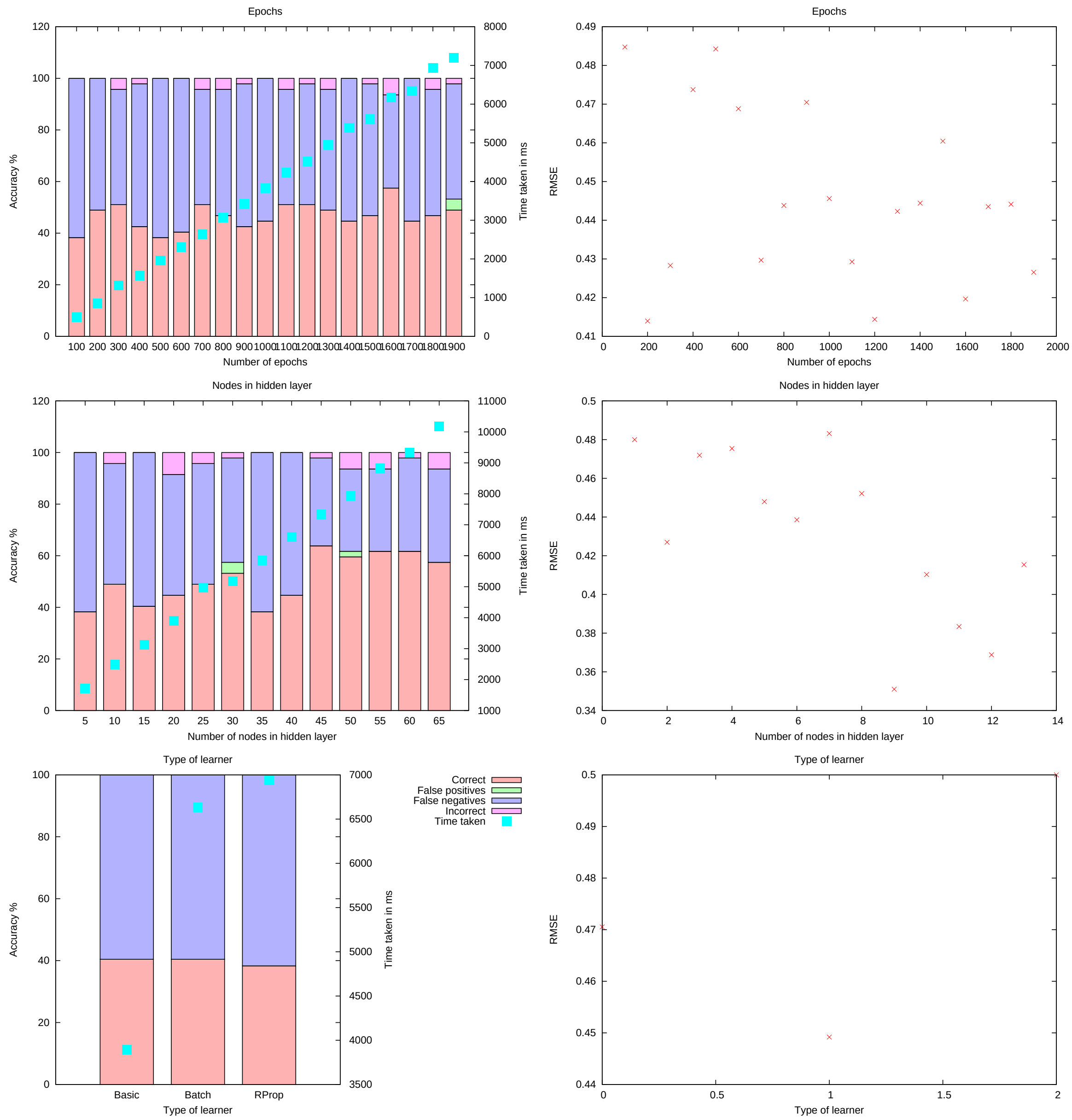

Figure 4.2: Comparisons of parameters in the neural networks recogniser.

4.3 Neural network parameters

The following pairs of graphs 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 timing now refers to the time taken to train the neural recogniser, rather than to recognise the gesture.

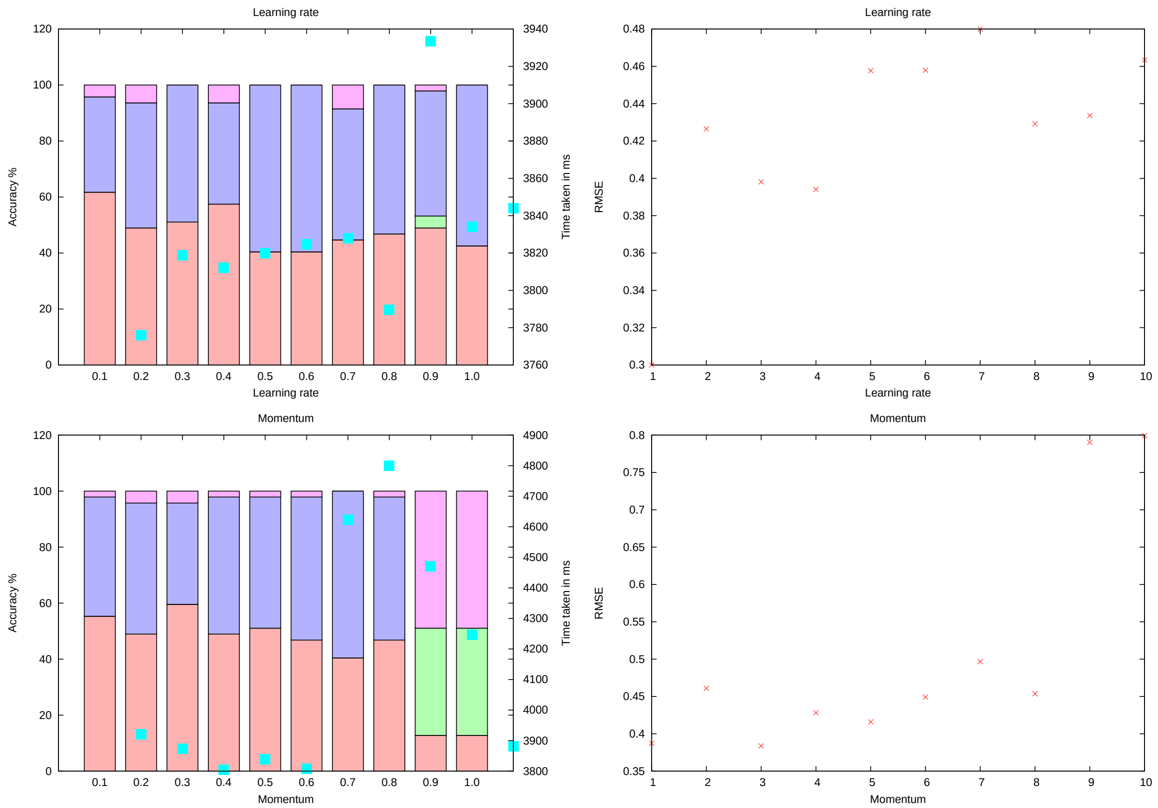

Figure 4.3: Comparisons of parameters in the neural networks recogniser.

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. However, changing the learner makes a significant difference to the RMSE, when changing from the basic backpropagation to batch propagation, in the same number of epochs.

The neural networks may also be varied in their training parameters in internal settings. Setting a very low learning rate does increase accuracy, whilst setting a too high momentum means that the gradient descent oscillates wildly and doesn't settle down to a good minimum within the number of epochs specified.

4.4 Hidden Markov model parameters

Figure 4.4 compares the number of hidden states and the number of iterations of which to perform the Baum-Welch algorithm. It is clear that a gesture can be successfully characterised in three states and the time taken increases exponentially when increasing states or

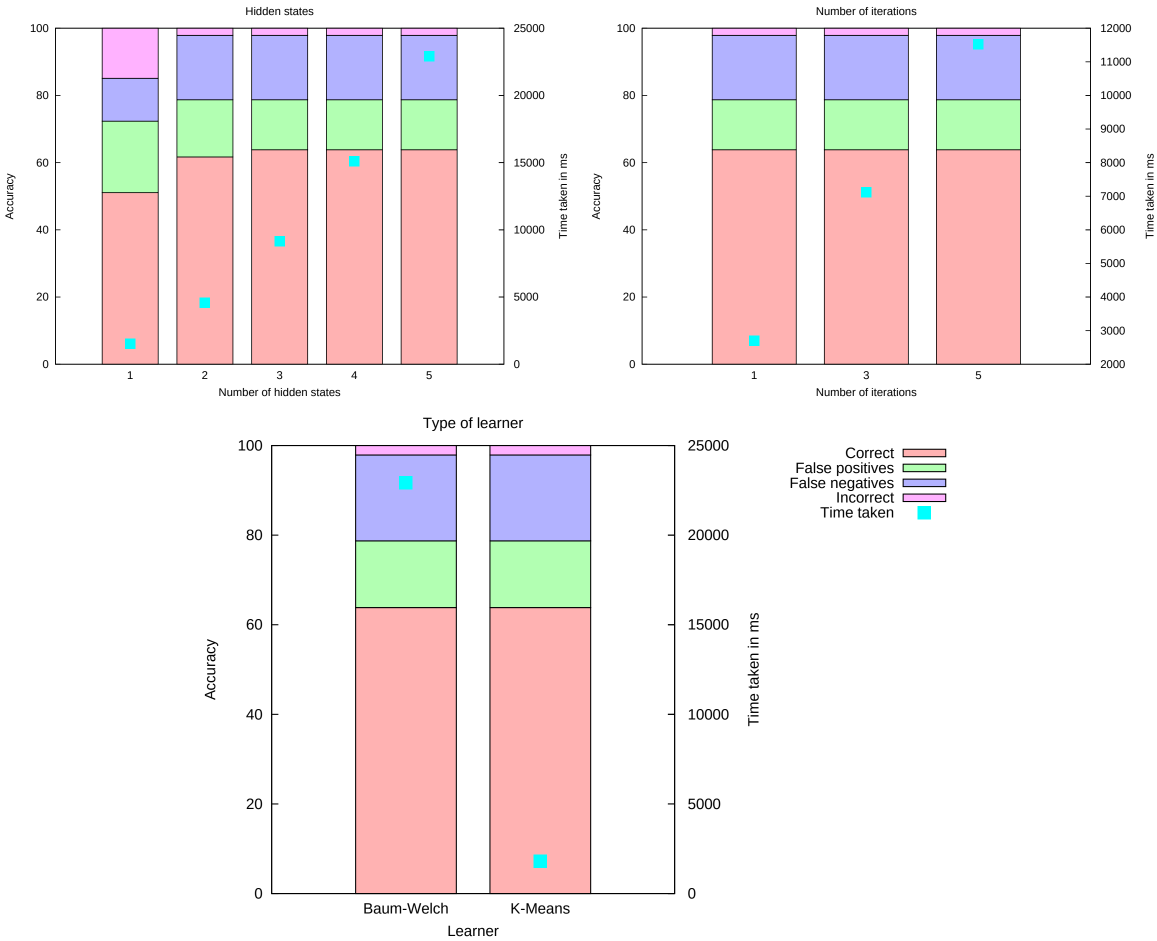

Figure 4.4: Comparisons of parameters in the hidden Markov model recogniser.

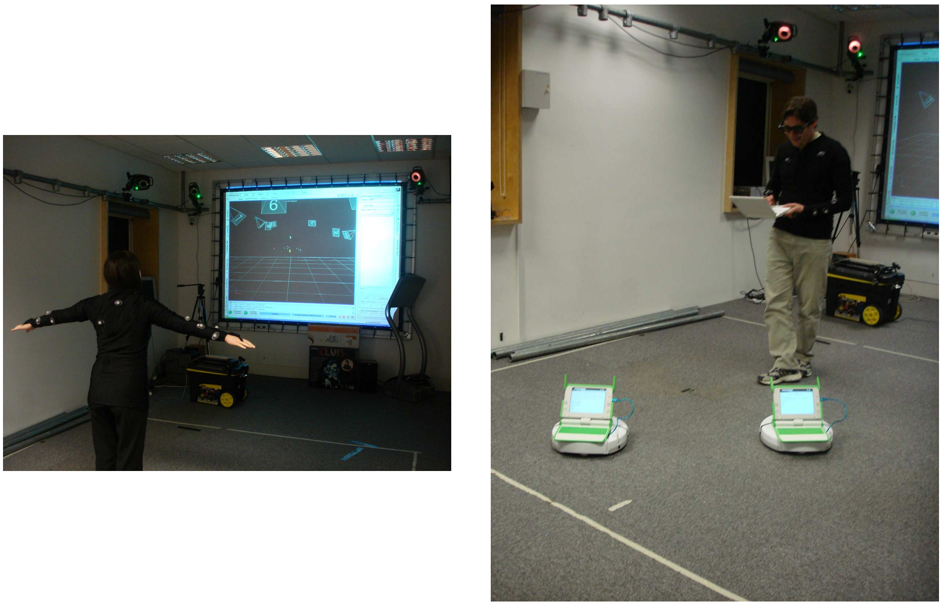

Figure 4.5: Photographs of demonstrations

number of iterations. The Baum-Welch algorithm is beaten by K-Means clustering, which performs equally as well despite taking a fraction of the time to train.

4.5 Real time control

Figure 4.5 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.

4.6 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 per cent. 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.

A point to note is that rather than implementing all the algorithms from scratch as originally planned, I chose to use established libraries (Joone and Jahmm) for my machine learning needs. This came about on advice of my supervisor and has reduced the workload in this case, to give me time to implement a more fully complete system.

Chapter 5

Conclusions

This project implements a working, complete system which has been demonstrated at the Computer Laboratory and tried with multiple users. It takes a multidisciplinary approach, rather than relying on a single technique, and all parts are fully modularised and cleanly separated.

In the upcoming summer, it 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. It is hoped that this will encourage others to experiment and interact with the system, testing its robostness and reliability.

Bibliography

[1] 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.

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

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

Appendix A

Neural Network recogniser

```
import java.io.*;
import java.util.*;

import org.joone.log.*;
import org.joone.engine.*;
import org.joone.engine.learning.*;
import org.joone.io.*;
import org.joone.net.NeuralNet;

class Neural extends Recogniser implements NeuralNetListener
{
        static double NEURAL_THRESHOLD = -1;

        static int LEARNING_MODE = -1;
        static int NUM_EPOCHS = -1;
        static int NUM_HIDDEN_NEURONS = -1;
        static double LEARNING_RATE = -1;
        static double MOMENTUM = -1;
        static boolean TRAIN_ON_NEGS = false;

        String output_file;
        double err;

        LinearLayer input;
        SigmoidLayer hidden, output;
        FullSynapse synapse_IH, synapse_HO;
        NeuralNet nnet;
```

```
Monitor monitor;
MemoryInputSynapse inputStream, samples;
TeachingSynapse trainer;

static long timing = System.nanoTime();

public static Gesture recognise(Person person, Features features)
{
        /* Uses:
                features.displacement
                features.leftarm|rightarm.get_delta(0|1|2)
                Total 7 input neurons
        */

        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();

        dump_results(pout);

        if(NEURAL_THRESHOLD < 0)
                NEURAL_THRESHOLD = Double.valueOf(Config.lookup("neuralthresho


        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();
}

private void init_parameters()
{
        if(LEARNING_MODE < 0)
                LEARNING_MODE = Integer.valueOf(Config.lookup("n_learner"));
        if(NUM_EPOCHS < 0)
                NUM_EPOCHS = Integer.valueOf(Config.lookup("n_epochs"));
        if(NUM_HIDDEN_NEURONS < 0)
                NUM_HIDDEN_NEURONS = Integer.valueOf(Config.lookup("n_hidden_n
        if(LEARNING_RATE < 0)
                LEARNING_RATE = Double.valueOf(Config.lookup("n_learning_rate"
        if(MOMENTUM < 0)
                MOMENTUM = Double.valueOf(Config.lookup("n_momentum"));
        if(TRAIN_ON_NEGS == false)
                TRAIN_ON_NEGS = Boolean.valueOf(Config.lookup("n_train_on_negs

        if (LEARNING_MODE == 2)
        {
                LEARNING_RATE = 1.0;
        }
}

private void set_columns(MemoryInputSynapse syn, int first, int last)
{
        String cols = "";

        for(int i = first; i <= last; i++)
        {
                if(i == last)
                        cols = cols + i;
                else
                        cols = cols + i + ",";
        }
        syn.setAdvancedColumnSelector(cols);
}

// NeuralNetListener interface methods follow:
```

```
public void errorChanged(NeuralNetEvent e)
{
        int cycle;

        Monitor mon = (Monitor)e.getSource();
        cycle = mon.getCurrentCicle();
        if(cycle % (NUM_EPOCHS/10) == 0 || cycle >= NUM_EPOCHS - 10)
        {
                err = mon.getGlobalError();
                Utils.log(String.format("%d", cycle) + " epochs remaining; RMS
                    + String.format("%5f", err));
    }
}

public void netStarted(NeuralNetEvent e)
{
        Utils.log("Training started");
        timing = System.nanoTime();
}

public void netStopped(NeuralNetEvent e)
{
        timing = System.nanoTime() - timing;
        Utils.log("Learning-mode Learning-rate Momentum Epochs Hidden-nodes ns
        Utils.results(LEARNING_MODE + " " + LEARNING_RATE + " " + MOMENTUM + "
        saveNeuralNet(nnet, output_file);
}

public void netStoppedError(NeuralNetEvent e, String error)
{
        Utils.log("Net stopped error: " + error);
}

public void cicleTerminated(NeuralNetEvent e) {}

static void saveNeuralNet(NeuralNet nnet, String filename)
{
        try
        {
                FileOutputStream stream = new FileOutputStream(filename);
                ObjectOutputStream out = new ObjectOutputStream(stream);
```

```
			out.writeObject(nnet);
			out.close();
		}
		catch(Exception e)
		{
			Utils.error("Cannot save neural net to <" + filename + ">");
		}
	}

	static NeuralNet restoreNeuralNet(String filename)
	{
		try
		{
			FileInputStream stream = new FileInputStream(filename);
			ObjectInputStream in = new ObjectInputStream(stream);
			return (NeuralNet)in.readObject();
		}
		catch(Exception e)
		{
			Utils.error("Cannot load neural net from <" + filename + ">");
		}
		return null; // Never happens
	}

	static void dump_results(Pattern pat)
	{
		double[] a = pat.getArray();
		Gesture gest = new Gesture(0);

		assert(a.length == Gesture.num_gestures);
		for(int i = 0; i < Gesture.num_gestures; i++)
		{
			gest.command = i;
			if (Utils.verbose)
				System.out.printf(gest.toString() + ": %5f\n", a[i]);
		}
	}

	static void init(Person p)
	{
		Layer input, output;
```

```
                if(p.nnet != null)
                        return; // Already initialised

                p.nnet = restoreNeuralNet(p.neural_file);

                input = p.nnet.getInputLayer();
                input.removeAllInputs();

                output = p.nnet.getOutputLayer();
                output.removeAllOutputs();

                p.netout = new DirectSynapse();
                output.addOutputSynapse(p.netout);

                p.nnet.getMonitor().setLearning(false);
        }
}
```

Appendix B

Hidden Markov model recogniser

```
import java.io.*;
import java.util.*;

import java.util.ArrayList;
import java.util.List;

import be.ac.ulg.montefiore.run.jahmm.*;
import be.ac.ulg.montefiore.run.jahmm.learn.*;
import be.ac.ulg.montefiore.run.jahmm.io.*;

class Markov extends Recogniser
{

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

        static final int NUM_DIMENSIONS = 9;
        static int NUM_STATES = -1;
        static int NUM_ITERATIONS = -1;
        static int LEARNER = -1;

        String output_root;

        static int SCALING_FACTOR = 100;

        public static Gesture recognise(Person person, Features features)
        {
```

```
        String filename;
        double[] probabilities = new double[Gesture.num_gestures];
        ArrayList<ObservationVector> framedata = toObservationVectors(features

        Hmm<ObservationVector> recog_hmm = new Hmm<ObservationVector>(5, new O

        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);
        }

        dump_results(probabilities);

        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);
}

static void dump_results(double[] a)
{
        Gesture gest = new Gesture(0);

        assert(a.length == Gesture.num_gestures);
        for(int i = 0; i < Gesture.num_gestures; i++)
        {
                gest.command = i;
                Utils.log(gest.toString() + ": " + String.format("%5f", a[i]))
        }
```

```
}

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
```

```
	}

	private static ArrayList<ObservationVector> toObservationVectors(ArrayList<Fra
	{
		ArrayList<ObservationVector> ovs = new ArrayList<ObservationVector>();
		double[] values;

		for (int i = 0; i < frames.size(); i++)
		{
			values = frames.get(i).toDoubles();
			for (double v : values)
			{
				v  = v / SCALING_FACTOR;
			}
			ovs.add(new ObservationVector(values));
		}
		//System.out.println("Done sample " + s.pathname);
		return ovs;
	}

	static void save_hmm(Learner learner, String filename)
	{
		try
		{
			FileOutputStream stream = new FileOutputStream(filename);
			ObjectOutputStream out = new ObjectOutputStream(stream);
			out.writeObject(learner.hmm);
			out.close();
		    Utils.log("Hmm saved to " + filename);
		}
		catch (IOException e)
		{
			Utils.error("Cannot save hidden markov model to <" + filename
		}
	}

	static Hmm restore_hmm(String filename)
	{
		try
		{
			FileInputStream stream = new FileInputStream(filename);
```

```
                        ObjectInputStream in = new ObjectInputStream(stream);
                    Utils.log("Hmm loaded from " + filename);
                        return (Hmm)in.readObject();
                }
                catch (IOException e)
                {
                        Utils.error("Cannot load hidden markov model from <" + filenam
                }
                catch (Exception e)
                {}
                return null;
        }

        private void init_parameters()
        {
                if(NUM_STATES < 0)
                        NUM_STATES = Integer.valueOf(Config.lookup("m_hidden_states"))
                if(NUM_ITERATIONS < 0)
                        NUM_ITERATIONS = Integer.valueOf(Config.lookup("m_iterations")
                if(LEARNER < 0)
                        LEARNER = Integer.valueOf(Config.lookup("m_learner"));

                learners = new ArrayList<Learner>(5);

                for (int i = 0; i < Gesture.num_gestures; i++)
                {
                        learners.add(new Learner(NUM_STATES, NUM_DIMENSIONS, NUM_ITERA
                                new Gesture(i)));
                }
        }
};

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, sequen
                hmm = bwl.learn(hmm, sequences);
        }

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

Appendix C

Control.py

```
from Tkinter import *
import scop

class Controls(Frame):
    """This class provides mousebutton and arrow key support for emitting basic contro

    def __init__(self, parent, sock, player, width=800, height=800, **options):
        Frame.__init__(self, parent, **options)
        self.width, self.height = width, height
        parent.title("Player " + str(player))
        self.sock = sock

        Button(self, text='accel', command=self.Accelerate).grid(row=0, column=1)
        Button(self, text='decel', command=self.Decelerate).grid(row=2, column=1)
        Button(self, text='left', command=self.TurnLeft).grid(row=1, column=0)
        Button(self, text='right', command=self.TurnRight).grid(row=1, column=2)
        Button(self, text='startstop', command=self.Pause).grid(row=1, column=1)

        self.running = 0
        self.period = 20 # milliseconds

        parent.bind('<KeyPress-Left>', self.TurnLeft)
        parent.bind('<KeyPress-Right>', self.TurnRight)
        parent.bind('<KeyPress-Up>', self.Accelerate)
        parent.bind('<KeyPress-Down>', self.Decelerate)
        parent.bind('<KeyPress-space>', self.Pause)
```

```
    def Accelerate(self, event=None):
        scop.scop_emit(self.sock, "a")

    def Decelerate(self, event=None):
        scop.scop_emit(self.sock, "d")

    def TurnLeft(self, event=None):
        scop.scop_emit(self.sock, "l")

    def TurnRight(self, event=None):
        scop.scop_emit(self.sock, "r")

    def Pause(self, event=None):
        if self.running:
            self.running = 0
            scop.scop_emit(self.sock, "s")
        else:
            self.running = 1
            scop.scop_emit(self.sock, "s")
```

Appendix D

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.