Cheryl Hung

Gesture-controlled Robotics using the Vicon Motion Capture System

Computer Science Tripos

King's College

May 9, 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 write a gesture recognition system which converts a stream of coordinates into commands to drive a robot.

Work Completed

Preprocessing Recognition methods: Heuristic, Neural, Markov, Hybrid Robot control Full simulations of coordinates and control

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

Acknowledgements

Acknowledgements

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 we 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 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):

1. accelerate
2. decelerate
3. turn left
4. turn right
5. start/stop - a test case for a more complex gesture, in this case a single clap at chest level.

diagram: gestures

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

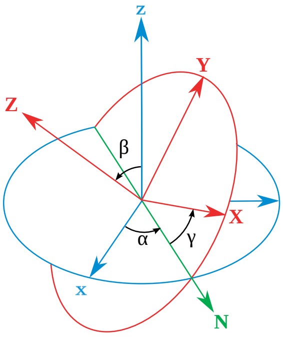

Figure 2.1: Euler Angles

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; it was inspired by the biological neural network. 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:

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

diagram: single neuron

w_{ji} connects node i to node j
a_{ij} is the activation for node, the weighted sum of the inputs
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

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

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

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 $S_{0:t-1} = (S_0, S_1, ...S_{t-1})$

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. The assumption is that the probabilities do not change over time, as the observations are stationary processes.

The forward-backward algorithm solves the task of smoothing; deducing the previous states from the current state, $Pr(S_t|e_{1:t})$ for $0 \leq T < t$.

latex:

The forward pass calculates $f_{1:t} = Pr(S_t|e_{1:t})$, passing $f_{1:t}$ as a forward message:

$$\begin{aligned}
Pr(S_T|e_{1:t}) &= Pr(S_T|e_{1:T}, e_{T+1:t}) \\
&= cPr(S_T|e_{1:T})Pr(e_{T+1:t}|S_T, e_{1:T}) \\
&= cPr(S_T|e_{1:T})Pr(e_{T+1:t}|S_T) \\
&= cf_{1:T}b_{T+1:t}
\end{aligned}$$

where $b_{T+1:t} = Pr(e_{T+1:t}|S_T)$ is the backward message for computing backwards probabilities:

$$\begin{aligned}
\text{Initialisation: } b_{t+1:t} &= Pr(e_{t+1:t}|S_t) = (1, ...1) \\
b_{T+1:t} = Pr(e_{T+1:t}|S_T) &= \sum_{s_{T+1}} Pr(e_{T+1:t}, s_{T+1}|S_T) \\
&= \sum_{s_{T+1}} Pr(e_{T+1:t}|s_{T+1})Pr(S_{T+1}|S_T) \\
&= \sum_{s_{T+1}} Pr(e_{T+1:t}, e_{T+2:t}|s_{T+1}) \\
&= \sum_{s_{T+1}} Pr(e_{T+1}|s_{T+1})Pr(e_{T+2:t}|s_{T+1})Pr(s_{T+1}|S_T) \\
&= \sum_{s_{T+1}} *Sensormodel * b_{T+2:t} * Transitionmodel \\
&= BACKWARDS(e_{T+1:t}, b_{T+2:t})
\end{aligned}$$

This smooths all points $1 : t$ in time $O(t^2)$. By storing the computed results for $f_{1:T}$ and using these in the computation of $b_{T+1:t}$, this is reduced to $O(t)$.

Matrix multiplication to calculate f Re-estimation

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 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

The majority of the system was developed under Ubuntu Linux, using NEdit and GEdit. The XO laptops run a modified version of Fedora Linux, while the PC used during live capture runs Windows XP.

This dissertation was typeset using $\LaTeX$.

2.7 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.

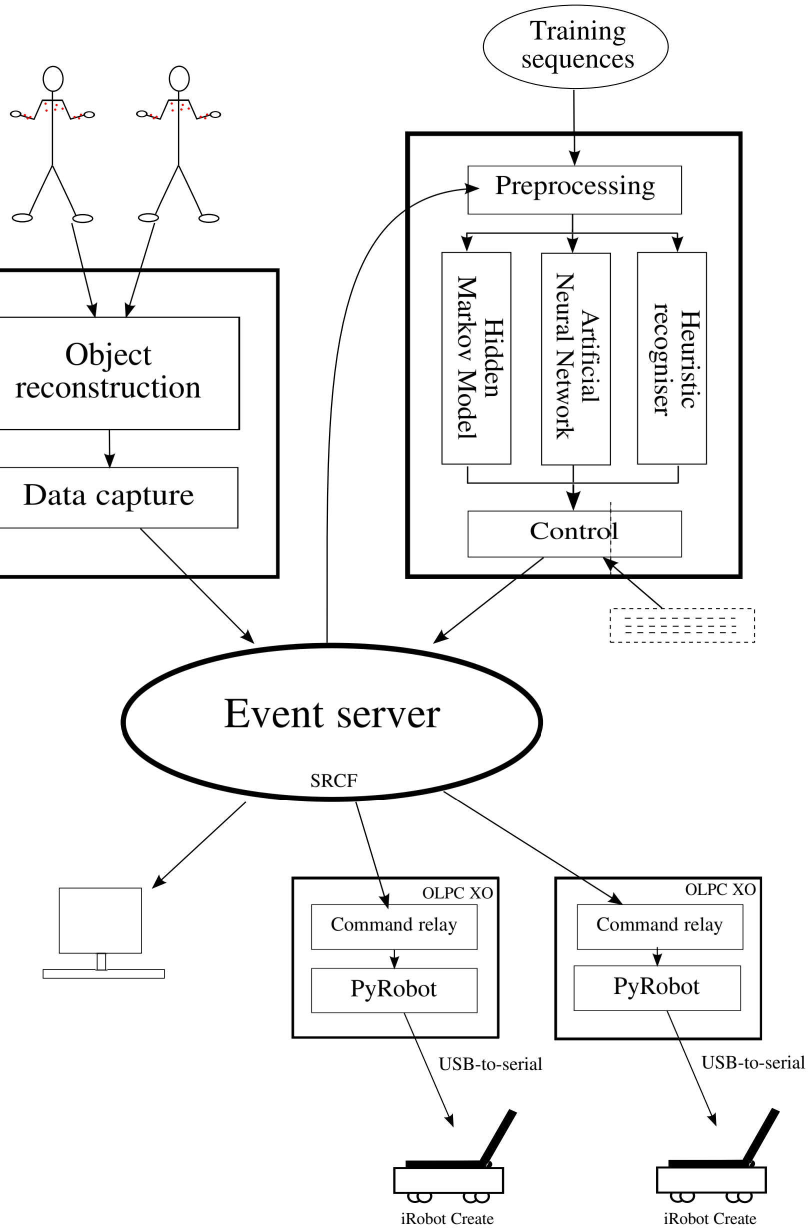

Figure 3.1: System Diagram

3.1.2 Protocols and data formats

<table>
<tr><th></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' 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.</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 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.</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>

3.1.3 Breakdown of components

Class diagrams diagram: class diagram

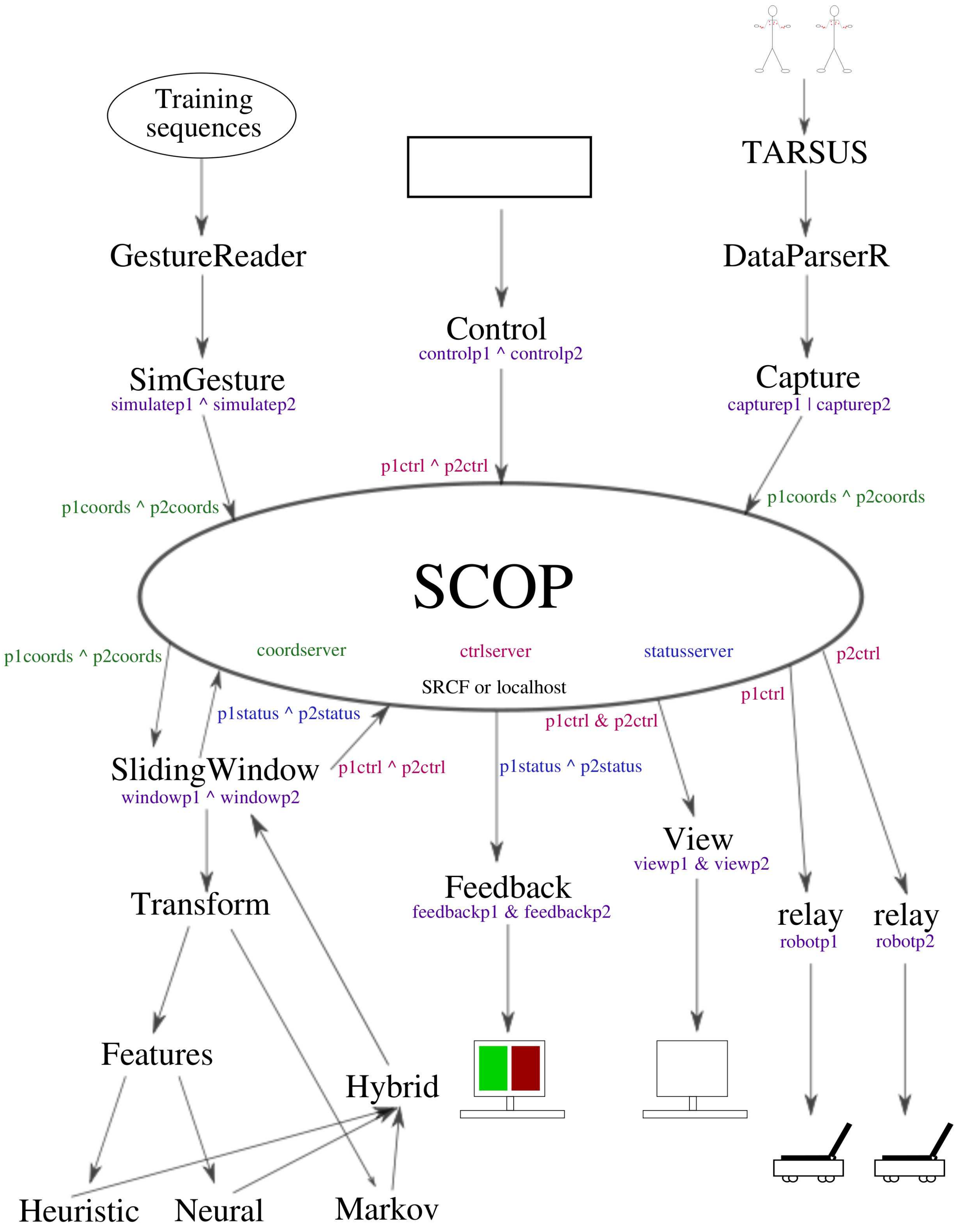

Figure 3.2: Data flow diagram

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 ±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:

```
double angle = 1.0
angle *= Math.sqrt(ax * ax + ay * ay + az * az);
if (angle > EPSILON)
{
    ax /= angle;
    ay /= angle;
    az /= angle;
}
else
{
    //Convert small rotations into a zero rotation about the z-axis
    to avoid floating point errors
    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.

```
while (true)
{
    msg = scop.get_message();
    circbuffer.add(Frame(msg));
    for windowsize = 50; windowsize < 350; windowsize += 10
        {
            if recognised(circbuffer, windowsize)
                break; //recognised emits 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)

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

Training implements NeuralNetListener
Training(double[][] inputdata, double[][] outputdata)
String gesture_dir, output_file
static final String index_filename = "training.dat"
static final int num_epochs = 2000
static final int num_hidden_neurons = 20
LinearLayer input
SigmoidLayer hidden, output
FullSynapse synapse_IH, synapse_HO
NeuralNet nnet
Monitor monitor
MemoryInputSynapse inputStream, samples
TeachingSynapse trainer
static ArrayList<Sample> read_index(String gesture_dir, String index_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)
static void saveNeuralNet(NeuralNet nnet, String filename)
static NeuralNet restoreNeuralNet(String filename)

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).

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.

3.3.4 Hidden Markov Model recogniser

Each gesture is modelled by a different Markov process, so five Hidden Markov Models are created for the five gestures. All the gestures empirically can be modelled by either two states (rising and falling) or four states (including a neutral start and end). This recogniser uses the raw data rather than the extracted data, since it is highly dependant on the temporal characteristics of the input vector.

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

3.3.5 Decision logic

This aggregates the results from each recogniser and uses a majority voting system to decide which, if any, are correct. The output of the stochastic recognisers are probabilities rather than hard binary decisions, as from the heuristic recogniser. Therefore exceeding a minimum threshold is required for a valid gesture to be selected.

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.

3.4 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. The Model-View-Controller design pattern is used 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.

diagram: screenshot of control keys for P1 and P2

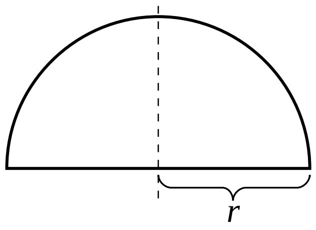

Figure 3.3: Turning radius

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.

diagram: screenshot of turtle graphics showing two turtles

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.

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 3.3.

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 [TODO]

4.1 Training data

Jitter in axis angles

4.2 Comparison of gesture recognition methods

4.3 Accuracy

Body part tagging - hat, body, belt

Stationary vs. moving user

Trained vs. untrained

Accuracy vs. amount of training data (compare HMM and ANN)

Choice of gesture

Optimal number of hidden nodes (ANNs) and hidden states (HMMs)

Chapter 5

Conclusions [TODO]

Chapter 6

Bibliography [TODO]

Appendix A

Appendices

Sample code

Appendix B

Project Proposal