Posture determination using a body sensor network

Rednic, R. , Kemp, J. , Brusey, J. and Gaura, E.

COGENT
computing

Posture Determination Using a Body Sensor Network

Ramona Rednic
John Kemp
Elena Gaura
James Brusey

Due of the large number of degrees of freedom of the human body, posture monitoring of human during activity regimes presents many research challenges. Several research groups world wide have engaged with the development of low-power wireless body sensor networks that are capable of providing real-time posture tracking for a variety of applications, such as dance and sports. The work reported here is concerned with the development of a wireless body sensor network that, as opposed to posture tracking, can: a) provide the identification and classification of eight human postures (standing, kneeling, sitting, crawling, walking, laying down on front and back, and laying on one side) in real-time and b) is able to relate this information wirelessly to a remote monitoring point. Posture information is an essential part of monitoring operatives in safety critical missions. The work sits within a larger project aiming to increase general safety of operatives in bomb disposal missions.

The goal of the posture body sensor network developped here is to identify the eight named postures using data from nine accelerometers placed at various sites on the human body. A prototype implementation which fulfills the goal has been produced and evaluated and is reported here.

Technical report number COGENT.006
Cogent Computing Applied Research Centre

Coventry University

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Overview

This project looks at posture identification using integrated sensor nodes that include accelerometers. This will be used as part of a wireless body sensor network which will be applied to wearers of a bomb disposal (EOD) suit in order to help them to feel more comfortable than currently possible [12]. Due of the nature of the missions, thick heavy materials protect the wearer, which means they must be provided with a cooling system. In order to provide good cooling, it is desirable to track the posture of the wearer and only apply cooling when it will be effective, allowing the life of the batteries that power the fan to be extended. The overarching project looks at monitoring a variety of bodily parameters within a bomb disposal suit, including skin temperature and posture, to ensure suitable cooling actuation, of which this report focuses on the posture aspect.

Tracking posture presents many challenges due of the large number of degrees of freedom of the human body. One advantage of the system presented here is that it allows an unconstrained environment for the wearer Due a wireless sensor network is used, which is small and light and doesn't require the level of cabling needed for a traditional wired system. Eight basic positions will be identified by this system: standing, kneeling, sitting, crawling, walking, laying down on front and back, and laying on one side. An example implementation is presented showing the working system identifying the posture in real time.

1.2 Related Work

The work related to the system presented in this report approaches different ways of finding the movement or position of a subject. Most of them use a combination of wireless and miniaturized sensor technologies to monitor a human body. Wireless Sensor Networks are commonly used for monitoring patients in hospitals or at home. In comparison, the work related here is concerned with increasing safety and comfort by integrating a body sensor network for monitoring the subject's condition, as well as visualising the position in real time.

Based on sensor location on a human body, tracking systems can be classified as non-vision based, vision based with markers, vision based without markers, and robot assisted systems. The first category includes the system that is presented here. The second one, vision based with markers, uses optical sensors, e.g. cameras, by placing identifiers upon the human body. As the human skeleton is a highly articulated structure, twists and rotations make the movement fully three-dimensional. This method is used also in movies, medical science, sports science and engineering. The third one, vision based tracking without markers, exploits external sensors like cameras to track the movement of the human body. For this method high speed cameras are required, as conventionally less than sixty frames a second provides an insufficient bandwidth for accurate data representation. The last method, robot assisted tracking, is used in rehabilitation. Human movement is reflected using electromechanical and electromagnetic sensors attached to the body [1, 3, 4, 5].

The most important papers that are related to this work are those presenting methods of finding postures for particular purposes. Posture tracking systems have been implemented for many uses, such as rehabilitation programs and movie graphics production. The system presented by Subir Biswas and Muhannad Quwaider [9] is the closest to the system that we proposed, but implemented through another perspective. This system uses the Mica2Dot (a commercial wireless sensor node) with a two-axis piezoelectric accelerometer incorporated. To determine position, they used a a novel radio frequency based proximity sensing method to monitor the relative movements of body segments, and then processed this using a Hidden Markov Model in order to identify the posture. The system is capable of identifying a limited set of postures: sitting, standing, walking and running. These postures are not determined in real time, which the system presented here is capable of doing, and the set of postures detected is limited compared to the full range which can be encountered. An diagram of their implementation is shown in figure 1.1.

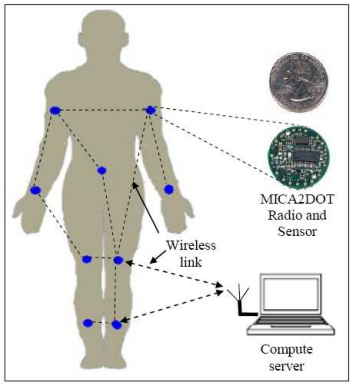


Figure 1.1: System implementation used by S. Biswas and M. Quwaider

The work discussed below is also related to wireless body sensor networks and focuses on patient care using more then one sensing device.

Most of the related work involves patient rehabilitation, for example the system implemented by H Ying *et al*. [10] for automatic step detection (locomotion measurement) for patients with Parkinson's disease. This system provides several methods of detection, such as the Pan-Tompkins Method, Template-Matching Method and Peak-detection method based on combined dual axial signals. The system implemented

consists of a dual axis accelerometer (ADL322), a low drop voltage regulator (TPS77027), and passive low-pass filtering. The Pan-Tompkins method is easy to implement, but fluctuations in the signal can result in false peak-searching intervals. The Template-Matching Method has an advantage: the algorithm is capable of detecting the steps self-adaptively. This, however, depends on the first template, which may be incorrect. The Peak-detection method was determined to be more suitable. This is based on combined dual axial signals and can be easily written in fixed-point algorithms which is suitable for microprocessor of limited computing power. They approach an interesting method of finding the difference between movements, such as walking, and static positions, such as sitting. However, for finding the posture in real-time, a wireless body sensor network is required that will process the data as it is gathered.

Another interesting system is proposed by J Pansiot *et al*. [11]. This provides activity recognition by integrating an ear-worn activity recognition (e-AR) sensor with ambient blob sensors. The system is composed of an e-AR sensor based on the BSN platform that consists of a Texas Instruments MSP430 processor, Chipcon CC2420 radio transceiver, Atmel 512KB EEPROM, MCC ChipOX SpO2 module and a 3-axis accelerometer. The ambient sensor used is a self-contained module consisting of a video sensor, on-board processor, wireless communication facility and a battery. From the e-AR acceleration sensor they extract two types of information: tilt, and a movement frequency spectrum. From the ambient sensor, the derived information includes the aspect ratio and mean velocity of the blob. Sensor fusion is performed based on a Gaussian Bayes EM classifier using the e-AR and blob sensor data, and for the implementation of the classifier they used the Bayes Net Toolkit (BNT). The system differentiates between walking, standing, standing (head tilted), sitting, sitting (sofa), reading, eating, lounging, and laying down (with 100%, 75%, 80%, 47%, 80%, 81%, 90%, 92%, and 100% accuracies respectively). The problem that is not solved in the work presented is the spatial dependency between the ambient sensor and the patient. The system is easy to implement in a home environment but is hard to use in an unknown environment that is not well monitored.

E. Farella *et al*. [6, 7] designed and implemented the WiMoCa, a wireless sensor node based on tri-axial integrated accelerometers used to detected human gestures and postures. The system uses an RF section (transceiver + antenna), LIS3L02DQ accelerometer, ATmega8 microcontroller, and a power supply. It is used to detect seven different postures: sitting, standing, and laying in four different manners. The full system is based on three WiMoCa nodes placed on the trunk, thigh, and shin. A diagram of the system and a sample of the collected data are shown in figure 1.3 and the system design is shown in figure 1.2. The system is not implemented to track dynamic postures such as walking and crawling, it is used only for static postures.

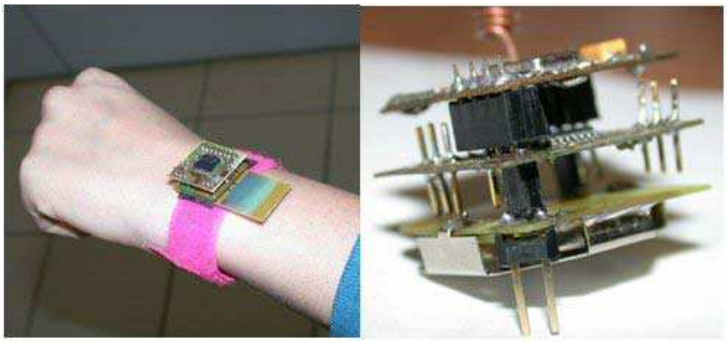


Figure 1.2: System design

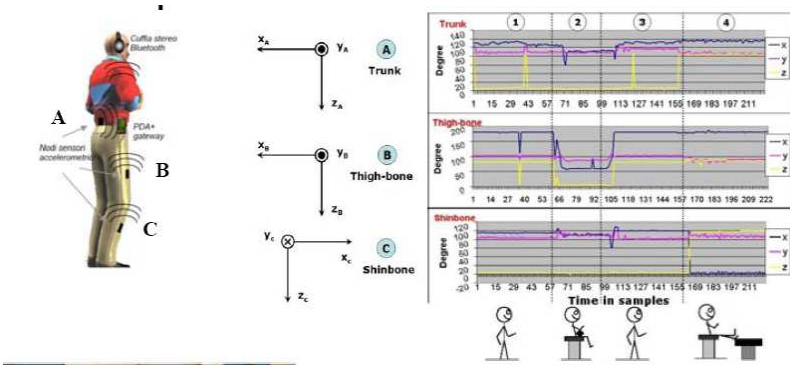


Figure 1.3: System diagram and sample data

Finding human posture with inertial (accelerometer and rate gyroscope) and magnetic (magnetometer) sensors was approached by D Fontaine *et al*. [2]. The system consists of accelerometers and magnetometers mounted on the human body and visualised with Kaydara Filmbox. Postures are not identified by the system, they are only visualized. The calibration of the magnetic sensors must be performed each time the system is used in a different environment.

Chapter 2

System Hardware and Visualisation

2.1 Main system

For data acquisition an integrated sensor board was used, along with Gumstix embedded computers and a custom expansion board, all shown in figure 2.1. The assembled system for the upper body is shown in figure 2.2. The components are each discussed in their own sections below.

2.1.1 Processing unit

Two Connex 400xm-bt Gumstix devices are used as the main processing platform. Nine sensor boards are connected to the Gumstix devices via an expansion board. The Gumstix devices communicate via Bluetooth to a computer that receives the data. The Gumstix devices are single board computers that measure 80mm x 20mm x 6.3mm. I/O options via expansion boards include synchronous and asynchronous serial, USB, Ethernet, Bluetooth and Wifi wireless interfaces. The specific board used here was the Connex 400xm-bt, which offers an Intel XScale PXA255 400MHz processor, 16MB of flash memory, a Bluetooth controller and antenna, and 60-pin and 92-pin connectors for expansion boards. The processor provides enough processing power to be able to deal with a wide range of tasks, and due to the combination of this power and the use of Linux as the onboard OS there is very little restriction on the languages available for use in software development, with both C and Python being routinely used. The 16MB of flash memory allows for storing large programs and data files. The devices are easy to mount or place around a person due to the small dimension. There is also an I^2C bus interface for the attachment of sensor packages. The processing power available is sufficient to allow real-time data modeling and decision making. An example of Gumstix device used is shown in figure 2.3.

Later in the project, the system was upgraded to use a different Gumstix device known as the Verdex, shown in figure 2.4. The specific device used, the Verdex XM4-bt, has a Marvell PXA270 XScale processor running at 400Hz, 64MB of RAM and 16MB of flash memory, a Bluetooth communications facility. It also provides USB host and CCD camera signals, a 60-pin Hirose I/O connector, a 120-pin MOLEX connector, and a 24-pin flex ribbon connector. The device weighs 8g and is 80mm x 20mm x 6.3mm, ideal for application that require mounting the components on to a subject. The advantage that this board has

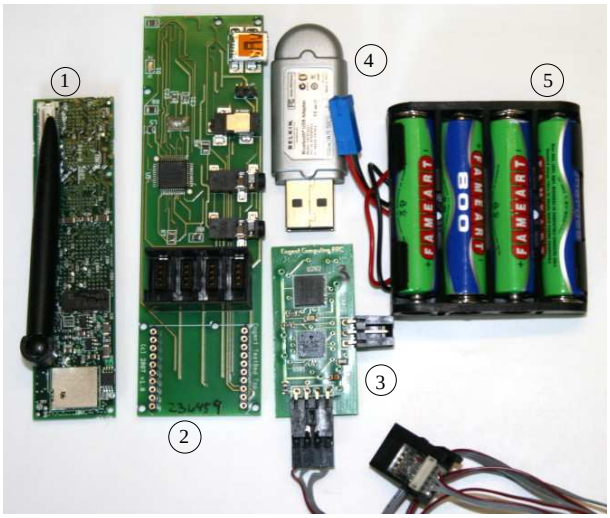


Figure 2.1: System components *1) Gumstix device 2) expansion board 3) sensor board 4) Bluetooth dongle 5) battery pack*

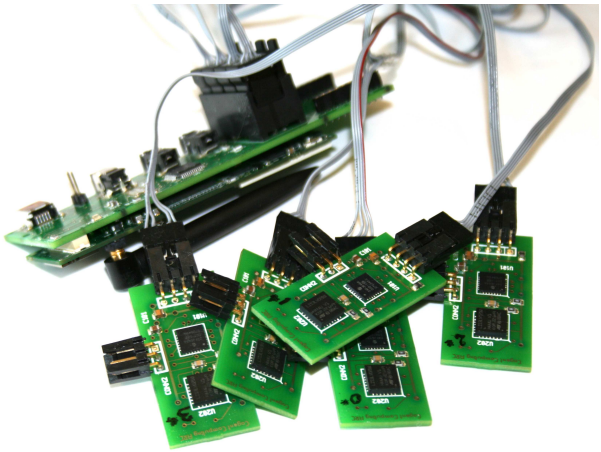


Figure 2.2: Assembled system for the upper body

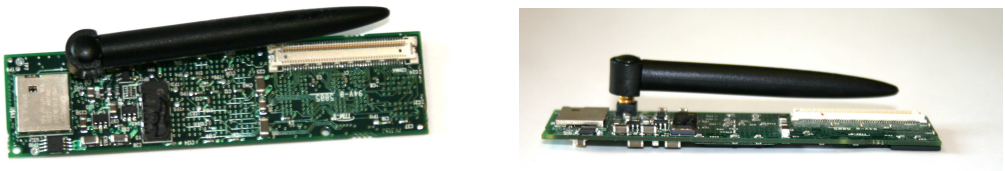


Figure 2.3: Connex 400xm-bt

Figure 2.4: Verdex XM4-bt

over the basix and connex motherboards is that include USB host and the higher RAM and flash memory options. Supply voltage is 3.6V to 5.0V DC.

2.1.2 Sensor Nodes

The board used for gathering data was designed by John Kemp, a PHD student with the Cogent Computing Applied Research Center at Coventry University, England. The board is composed of a PIC microcontroller, temperature sensor, accelerometer, and I^2C bus extender. It is designed to be a low-cost, small size, low-power wearable solution based on commodity components. The microcontroller is a PIC24FJ64GA002, a 28-pin 16-bit flash device provided by Microchip Technology Inc. The PIC uses a 8MHz internal oscillator with a 4x PLL option and multiple clock divide options. These enable it to be used between 31KHz and 32MHz if needed.

Table 2.1 shows a summary of the parameters for this device.

Parameter	Value
Operating Voltage	2.0V to 3.6V
I/O Sink/Source Current	18mA
Program Memory/SRAM (bytes)	64K/8K
Timers 16-Bit	5
I^2C/UART/SPI/Comparators	2/2/2/2
10 Bit A/D Channels	10

Table 2.1: PIC Parameters

The temperature sensor used is a ADT75A chip by Analog Device Inc. It is a 12-bit digital temperature sensor that can measure temperature from -55°C to +125°C. It contains a bandgap temperature sensor and a 12-bit ADC to monitor and digitize the temperature to a resolution of 0.0625°C. It operates at supply voltages from 3V to 5.5V and the average supply current is typically 200µA.

Table 2.2 shows a summary of the parameters for this device.

Parameter	Value
Sensing Range	-55°C to 125°C
Resolution	12 bits (0.0625°C)
Conversation Time	60ms
Update Rate	100ms
Supply Voltage	3 - 5.5V
Peak Current Consumption	350 - 525µA (excluding I^2C)
Shutdown Mode Consumption	3 - 8µA

Table 2.2: ADT75A Parameters

The accelerometer used is a LIS3LV02DQ chip by ST Microelectronics. It is a three axis digital output linear accelerometer that includes a sensing element and communicates through an I^2C/SPI digital serial interface. The LIS3LV02DQ has a user selectable scale of ±2g or ±6g and is capable of measuring acceleration over a bandwidth of 640Hz for all axes.

Table 2.3 shows a summary of the parameters for this device

Parameter	Value
Sensing Range	±2g / ±6g
Resolution	1.0mg (@ ±2g range)
Supply Voltage	2.16 - 3.6V
Peak Current Consumption	0.65 - 0.8mA
Shutdown Mode Consumption	1 - 10µA

Table 2.3: LIS3LV02DQ Parameters

The I^2C bus extender used is the P82B715PN chip by NXP Semiconductors. It is a bipolar integrated circuit that permits extension of the distance between components on the I^2C bus up to a maximum of 50m by buffering the data (SDA) and the clock (SCL) lines.

Table 2.4 shows a summary of the parameters for this device.

Parameter	Value
Supply voltage (operating) Vcc	4.5V to 12V
Supply current Icc	16mA
Power dissipation	300mW

Table 2.4: P82B715PN Parameters

The sensor board is showed in figure 2.5.

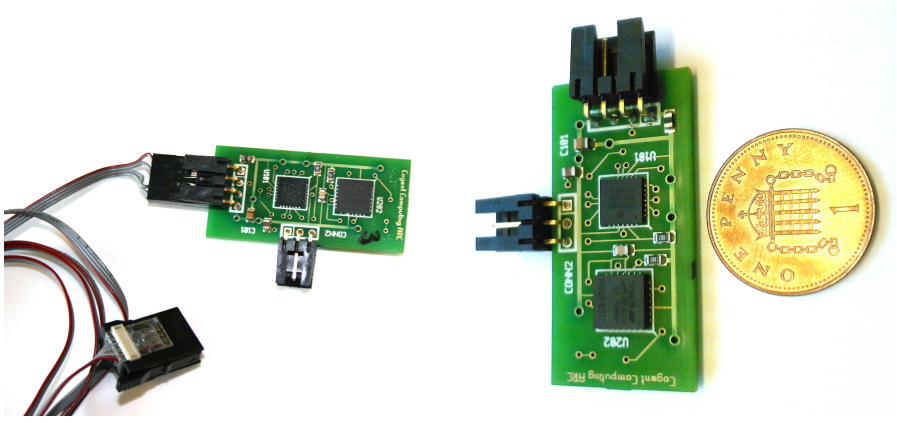


Figure 2.5: Sensor board

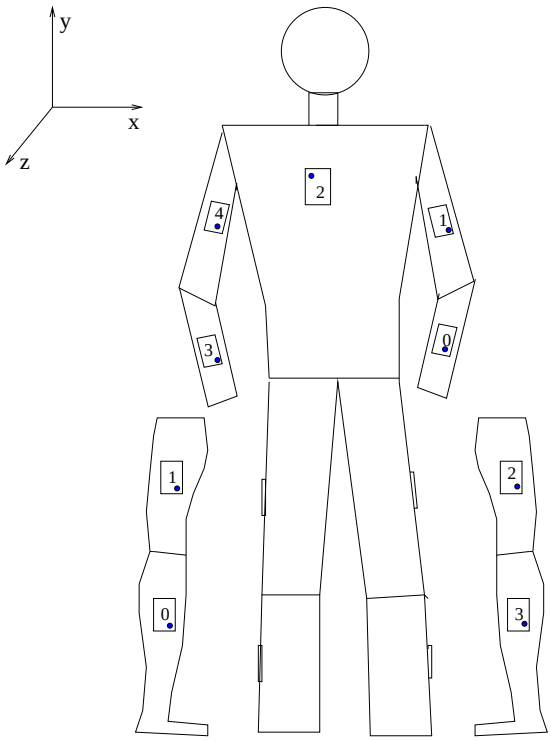


Figure 2.6: Sensor Position

2.1.3 Sensor Node Positioning

The sensors were positioned on the subject's body as shown in figure 2.6. The locations used are the chest, biceps, forearms, calf's and thighs. A single sensor is used per segment in order to increase the practical ease of wearing. Through experimentation it was found that the sensors provided enough information for this choice of positions to be applicable to the posture detection techniques used. The sensor positions chosen cover all of the body segments and thus allow a view of the overall posture of the subject.

2.1.4 Accelerometer Characterization

An accelerometer is a device that measures the acceleration and local gravity that it experiences. There are many types of accelerometers, including single-axis, dual-axis and triple-axis varieties. A triple-axis MEMS (micro electro-mechanical system) accelerometer, LIS3LV02DQ, is used here. This device is suitable for a variety of applications such as: free-fall detection, motion activated functions in portable terminals, anti-theft systems, inertial navigation, gaming and virtual reality input devices, and vibration monitoring and compensation. Acceleration and gravity are expressed in the SI unit of meters/second^2. Specific force and acceleration are related through Newton's laws of motion and Einstein's equivalence principle. The technology used for the LIS3LV02DQ includes suspended silicon structures which are

attached to the substrate at a few points called anchors and are free to move in the direction of the sensed acceleration. When an acceleration is applied to the sensor the proof mass displaces from its resting position, causing an imbalance in the capacitive half-bridge. This imbalance is measured using charge integration in response to a voltage pulse applied to the sense capacitor. This enables the sensor to detect the acceleration motion. When no other accelerations are applied to the device, it will measure 1g acceleration (9.81m/s^2) due to gravity. The LIS3LV02DQ is suitable for use with a microcontroller based system due to an I^{2}C interface, allowing easy connection with minimal wiring.

Directions of the gravitational field are shown in figure 2.7. Methods of finding the data for every axis manually are shown in figure 2.8.

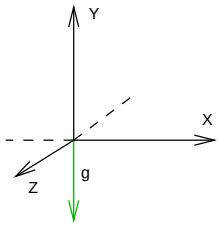


Figure 2.7: Direction of gravitational field

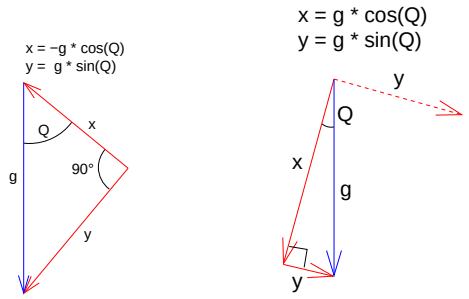


Figure 2.8: Example of axis vs gravitation

Accelerometer characteristics

Resolution: the incremental change in the input signal from a nonzero arbitrary value that would cause a corresponding change in the sensor output. The resolution of the accelerometer is 1mg.

$10^{-3}g * 9.81m/s^2 = 0.00981m/s^2$

Sensitivity: the smallest detectable change in output of the sensor in response to a change in the input. At ±2g scale, sensitivity is quoted as 1024LSb/g.

Offset error: the value that will be output when the ideal value would be zero. At ±2g range, the offset of the device is quoted as ±20mg (X and Y) and ±40mg (Z)

Accuracy: a measurement is more accurate if it is closer to what is defined as the "truth" as compared to a reference standard.

Linearity: a measure of how close the sensor response curve is to being a straight line. A linear response means that the output is proportional to the input signal. Linear response is always a desirable feature. The non-linearity of the accelerometer used is quoted as being ±2% of the full range.

Calibration

The acceleration sensor is factory calibrated and downloads the data (trimming values for sensitivity and zero-g offset) from a non-volatile structure each time it is powered up or reset. The best way of providing additional calibration of the device is by modifying the data once it is received by a host device. This provides more flexibility, as well as avoiding modifications to the internal state of the accelerometer.

The method used to determine the calibrate values for the sensors doesn't require a certain series of pre-defined positions, making it practical in a mobile setting. This can be performed after the system is assembled. The earth's gravity force is used as a known static acceleration when the sensor has no dynamic component applied to it. The equation below is valid for this state:

$$\sqrt{x^2+y^2+z^2} = 1$$

The equation for the offset and scale errors of the accelerometer is:

$$\left(\frac{u-c_x}{m_x}\right)^2 + \left(\frac{v-c_y}{m_y}\right)^2 + \left(\frac{w-c_z}{m_z}\right)^2 = 1$$

In this equation: $\overrightarrow{u} = (u,v,w)$ (the uncalibrated data read from the accelerometer), $\vec{c} = (c_x,c_y,c_z)$(the measurement offset) and $\vec{m} = (m_x,m_y,m_z)$ (the scaling factor). The system would be perfect if $\overrightarrow{c} = (0,0,0)$ and $\vec{m} = (1,1,1)$. Six equations are used to find the offset and scaling factor for all three axes. Six measurements are necessary and should be different from each other in order to guarantee a stable convergence of the non-linear solver.

2.1.5 Expansion board

The expansion board used is a custom board which provides connections for I^2C devices, a battery, a USB client interface, and a Gumstix device. This board is shown in figure 2.9.

2.2 Connectivity

The system implemented has five sensor boards for the upper body that sense the acceleration and pass it through one Gumstix device, and four sensor boards for the lower body passing the sensing acceleration

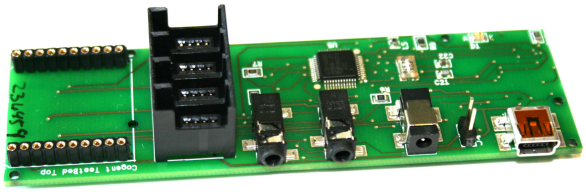


Figure 2.9: Expansion board

through a second Gumstix device. The system is designed to sense the acceleration, pass the data via Bluetooth to a computer, process the data gathered, produce a decision as to posture, and then send the decision to a visualiser. The communication between the Gumstix devices and the monitoring point are carried out via Bluetooth. The Bluetooth radio provides a convenient means of establishing a small network between devices (where the accelerometers are split into groups, each assigned a different master) or between the Gumstix device and a host computer. Bluetooth transfer rates are limited, but more than adequate for transmitting the quantity of readings expected for this system. Wireless connectivity is required for communicating readings to a remote monitoring point. Using a wireless method, we don't have restrictions on the mobility of the subject or the location of the monitoring device. The system connections are shown in figure 2.10.

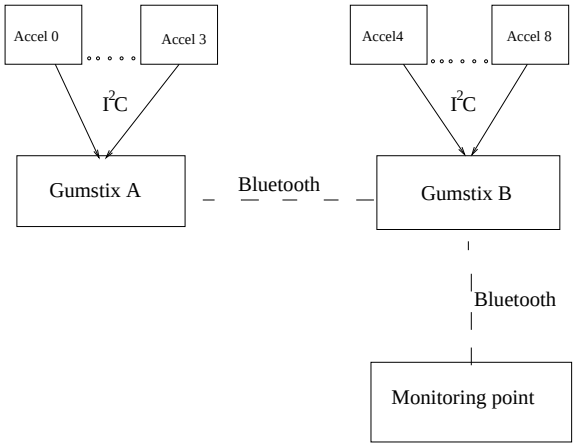


Figure 2.10: Connectivity

I^2C is a 2-wire (SCL – Serial Clock Line or clock and the SDL – Serial Data Line or data wire) serial 8 bit communication protocol. LIS3LV02DQ it can be accessed through I^2C and also SPI serial interface. The sensor is using I^2C that is a slave bus. It can be used to send and to receive data to/from the interface. The communication is beginning with a START bit from the master, a transmission of the slave address (SAD) represented in 7-bits (for LIS3LV02DQ is 0011101b) and 1 single bit representing whether it wishes to write (0) to or read (1) from the slave. If the slave exists on the bus will respond the Master with and acknowledge bit so the Master will know to continue his transmission/receiving data to the slave. The transmission will finish with a STOP bit from the Master. Data is transfer with the Most Significant Bit

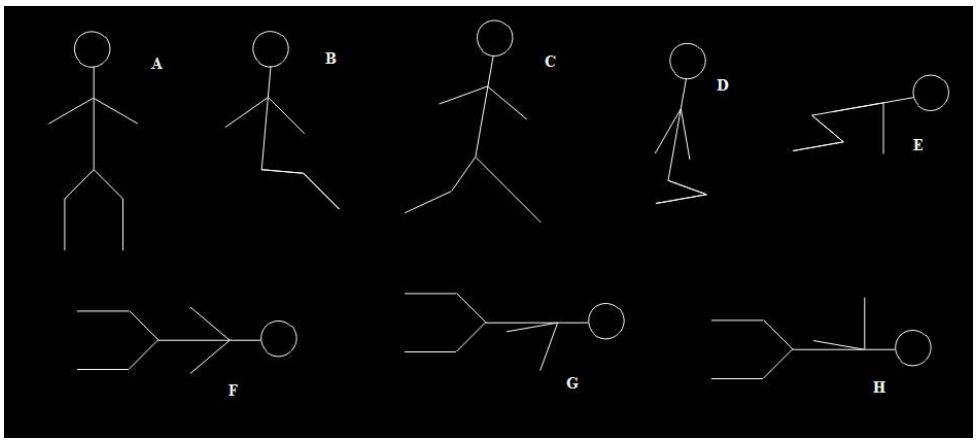


Figure 2.11: Visualisation *A) Standing B) Sitting C) Walking D) Kneeling E) Crawling F) Laying down on one side G) Laying on front H) Laying on back*

(MSB) first. The signal diagram are shown in table 2.5.

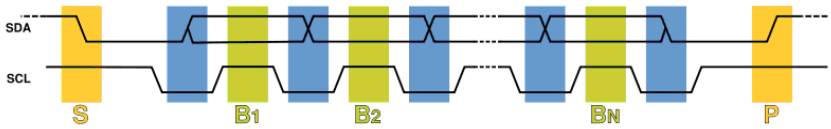


Table 2.5: I^2C functionality

Data transfer is initiated with the START bit (S) when SDA is pulled low while SCL stays high. Then, SDA sets the transferred bit while SCL is low (blue) and the data is sampled (received) when SCL rises (green). When the transfer is complete, a STOP bit (P) is sent by releasing the data line to allow it to be pulled up while SCL is constantly high.

2.3 Visualisation

Data visualisation is shown using stick figures, implemented by the “turtle” drawing method and programed in Python. The visualisation of the various postures appears as shown in figure 2.11. The stick figure corresponding with the result of processing the data is shown in the visualiser in real time. Actual screenshots of these postures as shown in the visualiser are demonstrated in figure 2.12.

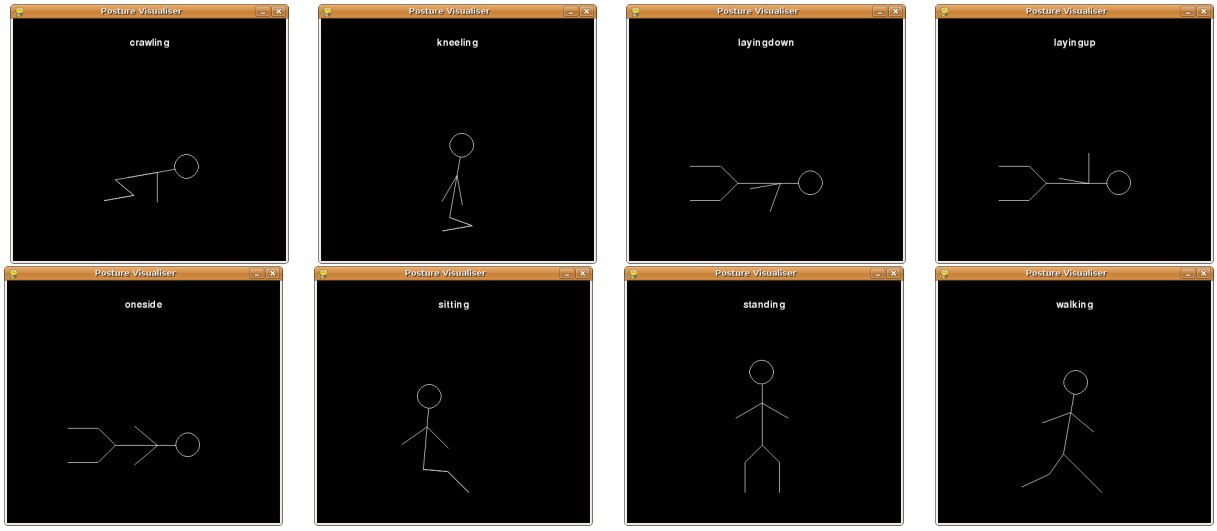


Figure 2.12: Visualiser screenshots

Chapter 3

Data processing

3.1 Postures

All of the postures which the system aims to identify are shown in figure 3.1. This set of postures was predefined for use in the experimentation in order to provide a structured means of processing the gathered data.

3.2 Method of Posture Identification

3.2.1 Training data

Decision trees based on the accelerometer readings were chosen as the method of processing the data. These were chosen as they are readily generated using available tools, are easily converted into a set of rules for real-time processing, and work with numerical values.

Data gathered from an initial testing run using the system was processed using Weka (Waikato Environment for Knowledge Analysis) Explorer generating a J48 tree classifier with 10 fold cross validation. Weka is a suite of machine learning software written in Java, developed at the University of Waikato. WEKA contains a collection of visualisation tools and algorithms for data analysis and predictive modeling, together with graphical interfaces for easy access to this functionality. Weka supports several standard data mining tasks, data preprocessing, clustering, classification, regression, visualisation and feature selection. All of Weka's techniques are predicated on the assumption that the data is available as a single flat file or relation, where each data point is described by a fixed number of attributes. The preprocess panel is the start point for knowledge exploration from Weka. This panel is shown in figure 3.2. The preprocess panel is showing : loading data (four buttons at the top of the preprocess section enable the loading of data into Weka), the current relation box (the "current relation" is the currently loaded data, which can be interpreted as a single relational table in database terminology), attributes (list of attributes), selected attribute, and visualisation.

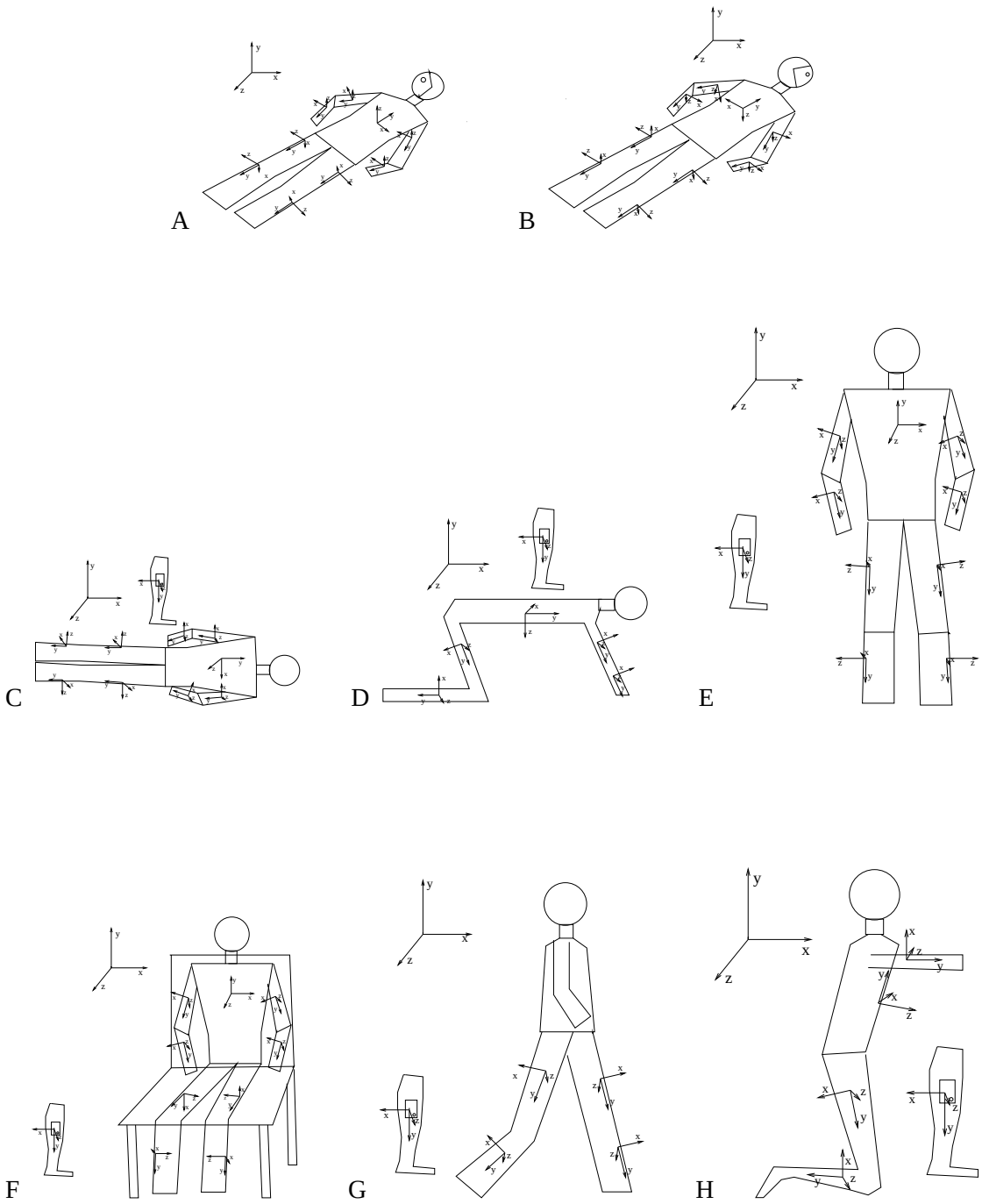


Figure 3.1: All identified postures - *A) Laying on back B) Laying on front C) Laying on one side D) Crawling E) Standing F) Sitting G) Walking H) Kneeling*

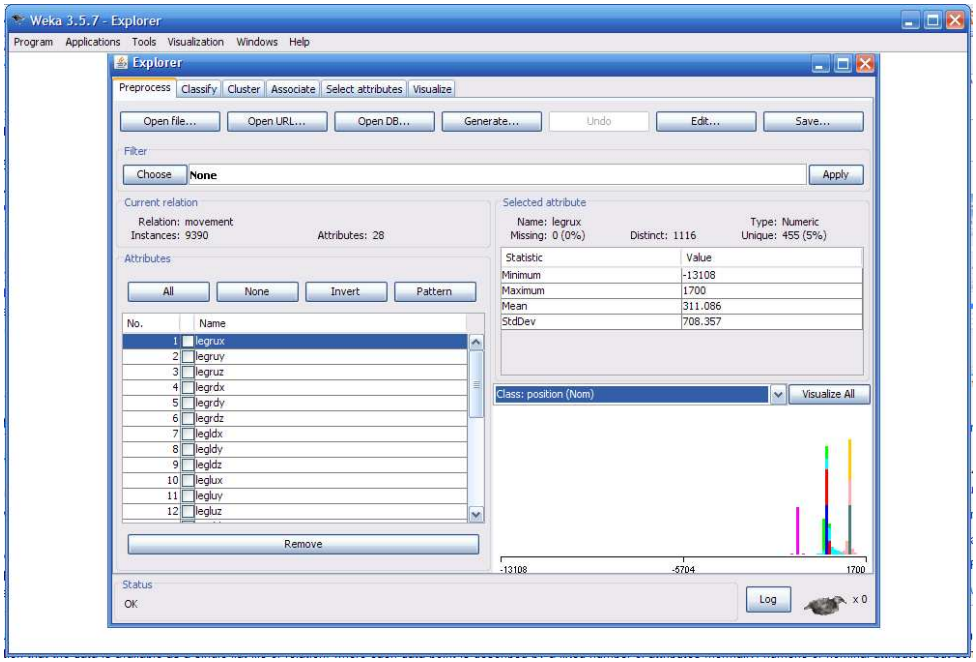


Figure 3.2: Preprocess panel

The classify panel is shown in figure 3.3. This panel is showing: classifier (choose the classifier), test options (there are four test modes: use training set, supplied test set, cross-validation, percentage split), result list (showing which method was chosen and giving options for taking the results out), classifier output (where the results can be browsed).

Weka uses an ASCII text file for processing the data. This is an ARFF (Attribute-Relation File Format) file that describes a list of instances sharing a set of attributes. ARFF files have two distinct sections. The first section is Header information, which is followed the Data information. The header of the ARFF file contains the name of the relation, a list of the attributes (the columns in the data), and their types. An example header on the posture detection data set looks like this:

```
@relation movement

@attribute legrux real
@attribute legruy real
@attribute legruz real
@attribute legrdx real
@attribute legrdy real
@attribute legrdz real
@attribute legldx real
@attribute legldy real
@attribute legldz real
@attribute leglux real
@attribute legluy real
@attribute legluz real
@attribute armldx real
@attribute armldy real
```

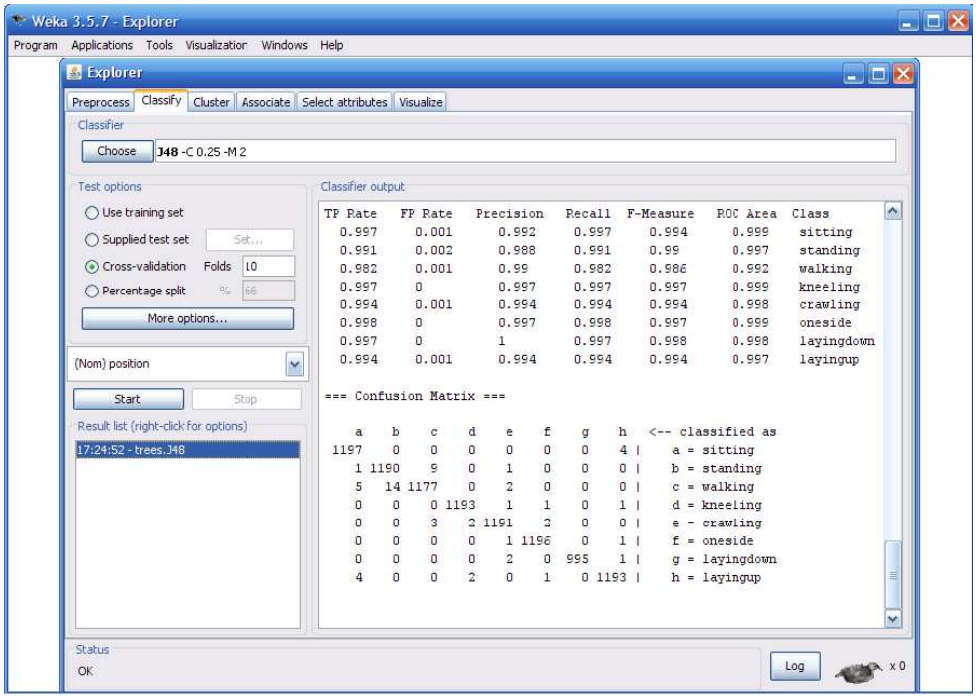


Figure 3.3: Classify panel

```
@attribute armldz real
@attribute armlux real
@attribute armluy real
@attribute armluz real
@attribute bodyx real
@attribute bodyy real
@attribute bodyz real
@attribute armrdx real
@attribute armrdy real
@attribute armrdz real
@attribute armrux real
@attribute armruy real
@attribute armruz real
@attribute position {sitting, standing, walking, kneeling, crawling,
                     oneside, layingdown, layingup}
```

The Data of the ARFF file looks like the following:

```
@data
81,−1032,−59,−1049,69,−245,893,−9,361,−127,−1060,86,1005,−304,134,131,−1042,218,12,992,268,−907,−370,398,−385,−946,226, sitting
161,−1023,61,−258,−1044,−113,137,−1024,−235,−205,−1048,77,186,−1033,72,−74,−1054,186,−28,1001,203,−236,−1036,196,−136,−1056,45, standing
586,−800,47,−371,−977,−40,41,−1007,−327,−148,−1031,−52,162,−1072,63,−383,−789,337,126,1070,75,−218,−1046,191,114,−856,39, walking
1067,128,−25,−200,−1054,−103,61,−1044,−166,−990,98,−167,950,−140,437,905,−558,69,−44,1019,102,−839,−123,627,−925,−412,255, kneeling
1434,119,116,471,−1003,89,561,−890,−352,−778,−27,−651,553,−1004,−51,31,−1038,−184,−106,338,−1161,−55,−1330,−271,681,−1104,−427, crawling
133,−213,1028,−123,−290,958,123,−32,−1120,94,−121,−991,655,−42,−798,367,−132,−976,−1032,103,191,298,−110,1038,−139,15,934, oneside
1007,107,358,903,−251,360,−943,−144,−603,−1023,81,−105,−35,−79,−1028,−755,−48,−695,1,−110,−1062,160,−124,−995,775,−167,−805,layingdown
−1029,−162,−175,−1022,141,−306,985,89,167,1054,−160,107,969,−113,401,1053,−205,79,44,−210,1006,−992,−86,300,−1060,−77,−187, layingup
```

Other classifiers were also examined, for example RandomTree that gave 99.2545% correctly classified

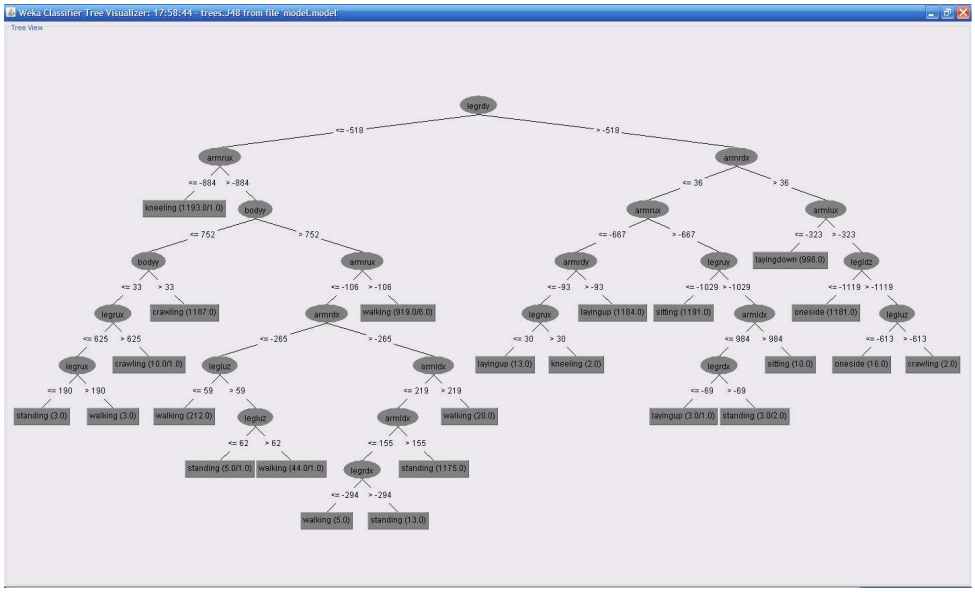


Figure 3.4: Tree visualisation

data items and 0.7455 % incorrectly classified data items. The J48 tree classifier gave 99.3823 % correctly classified data items and 0.6177 % incorrectly classified data items. The tree is shown in figure 3.4.

The confusion matrix in table 3.1 shows how good the results of the classifier are (based on the training data only). The confusion matrix is simply a square matrix that shows the various classifications and misclassifications of the model in an easy to read form. The columns of the matrix correspond to the number of data items classified as a particular value and the rows correspond to the number of data items with that actual classification.

a	b	c	d	e	f	g	h	<– classified as
1197	0	0	0	0	0	0	4	a=sitting
1	1190	9	0	1	0	0	0	b=standing
5	14	1177	0	2	0	0	0	c=walking
0	0	0	1193	1	1	0	1	d=kneeling
0	0	3	2	1191	2	0	0	e=crawling
0	0	0	0	1	1196	0	1	f=oneside
0	0	0	0	2	0	995	1	g=layingdown
4	0	0	2	0	1	0	1193	h=layingup

Table 3.1: Confusion matrix

The results of the tree generation procedure are presented in Appendix B, and are explained briefly here. The first three lines of the output from the process is shown below, demonstrating the form of the tree:

legrdy <= -518

| armrux <= -884: kneeling (1193.0/1.0)

| armrux > -884

Each line represents a node in the tree. The second and third lines, those that start with a '|', represent child nodes of the node on the first line. In the general case, a node with one or more '|' characters before the rule is a child node of the node that has one less "|" character and appears most recently before it in the list. The part of the line after the "|" character declares the rule. There are two "sister" nodes for each rule, the first which will be evaluated if the relevant variable is less than or equal to the threshold, and the second which will be evaluated if the variable is greater than the threshold. If the rule is followed by a semicolon and a class designation (such as "kneeling" in the second line above) then a leaf node has been reached and the classification of the data is as given. If not then processing continues with the children of the current node.

Nodes that generate a classification, such as

| armrux <= -884: kneeling (1193.0/1.0)

are followed by a number (sometimes two) in parentheses. The first number represents how many data items in the training set were correctly classified by the node, in this case 1193. The second number (if present) represents the number of data items incorrectly classified by the node, in this case 1.

General error output

Correctly Classified Instances	9332 (99.3823%)
Incorrectly Classified Instances	58 (0.6177%)
Kappa statistic	0.9929
Mean absolute error	0.0017
Root mean squared error	0.0381
Relative absolute error	0.7668%
Root relative squared error	11.5186%
Total Number of Instances	9390

In the list of statistics for the tree presented above, three self-explanatory fields can be seen: correctly classified instances, incorrectly classified instances, and total number of instances. The additional statistics are discussed below.

1. Correlation coefficient

$\frac{S_{PA}}{S_p S_A}$ where $S_{PA} = \frac{\sum_i (p_i - \overline{p})(a_i - \overline{a})}{n-1}$, $S_p = \frac{\sum_i (p_i - \overline{p})^2}{n-1}$, and $S_A = \frac{\sum_i (a_i - \overline{a})^2}{n-1}$

The correlation coefficient measures the statistical correlation between the predicted and actual values. This method does not change when values are scaled for the test cases. A higher number represents a better model, with a value of 1 meaning perfect statistical correlation and a 0 meaning there is no correlation at all.

2. Kappa Statistic

$\frac{P(A)-P(E)}{1-P(E)}$ where $P(A)$ is the proportion of times the modeled value was equal to the actual value and $P(E)$ is the expected proportion by chance.

Whereas the correlation coefficient is used for numerical data, the kappa statistic (also called the kappa coefficient), is used as a means of classifying agreement in categorical data. A kappa coefficient of 1 means a statistically perfect modeling whereas a 0 means every model value was different from the actual value. A kappa statistic of 0.7 or higher is generally regarded as good statistical correlation, but of course, the higher the value, the better the correlation.

3. Root mean-squared error

$\sqrt{\frac{(a_1-c_1)^2+(a_2-c_2)^2+...+(a_n-c_n)^2}{n}}$

The root mean-squared (RMS) error is one of the most commonly used measures of success for numerical prediction. This value is computed by taking the average of the squared differences between each computed value (c_i) and its corresponding correct value (a_i). The root mean-squared error is simply the square root of the mean-squared-error. The root mean-squared error gives the error value the same dimensionality as the actual and predicted values.

4. Mean absolute error

$\frac{|a_1-c_1|+|a_2-c_2|+...+|a_n-c_n|}{n}$

Mean absolute error is the average of the difference between predicted and actual value in all test cases; it is the average prediction error.

5. Root relative squared error

$\sqrt{\frac{(a_1-c_1)^2+(a_2-c_2)^2+...+(a_n-c_n)^2}{(a_1-\bar{a})^2+(a_2-\bar{a})^2+...+(a_n-\bar{a})^2}}$

Relative squared error is the total squared error made relative to what the error would have been if the prediction had been the average of the absolute value. As done with the root mean-squared error, the square root of the relative squared error is taken to give it the same dimensions as the predicted values themselves. Also, just like root mean-squared error, this exaggerates the cases in which the prediction error was significantly greater than the mean error.

6. Relative absolute error

$\frac{|a_1-c_1|+|a_2-c_2|+...+|a_n-c_n|}{|a_1-\bar{a}|+|a_2-\bar{a}|+...+|a_n-\bar{a}|}$

Relative absolute error is the total absolute error made relative to what the error would have been if the prediction simply had been the average of the actual values.

In the four previous equations, c_i is the numerical value of the prediction in the iTh test case, a_i is the actual value of the ith test case,$\bar{a}$ is the average of the actual values of all the test cases, and n is the total number of test cases. In all above error measurements, a lower value means a more precise model, with a value of 0 being the statistically perfect model.

Visualisation of data with weka are shown in figure 3.5.

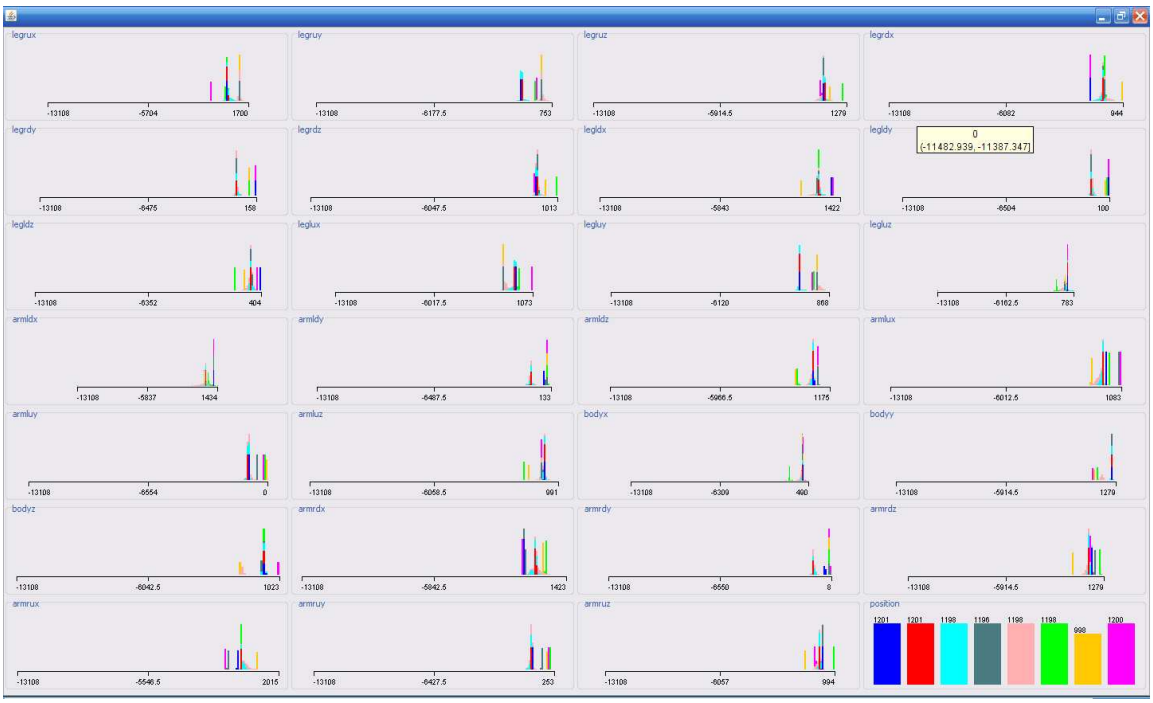


Figure 3.5: Data visualisation Weka

3.2.2 Processing of training data

The aim with this system is to perform the data modeling within the wireless system on the Gumstix nodes, with the processed results being relayed to the application in a usable format. On one of the Gumstix nodes, a program, written in Python, interprets the decision tree generated by Weka. The results of this are transmitted to the visualiser.

An example of the data gathering process is shown in the figure 3.6 where the upper part of the screen shows the data gathered from the accelerometers, the left corner shows the visualiser application, and the right part shows the control terminal. The figure shows the data in the upper portion of the screen from all accelerometers at the same time in a simple array format. The first twelve numbers are data gathered from the accelerometers placed on the lower part of the body (x, y, and z for each of four accelerometers) and the next fifteen numbers are the data gathered from the accelerometers placed on the upper part of the body. The decision is made based on the classifier tree and is shown on the visualiser.

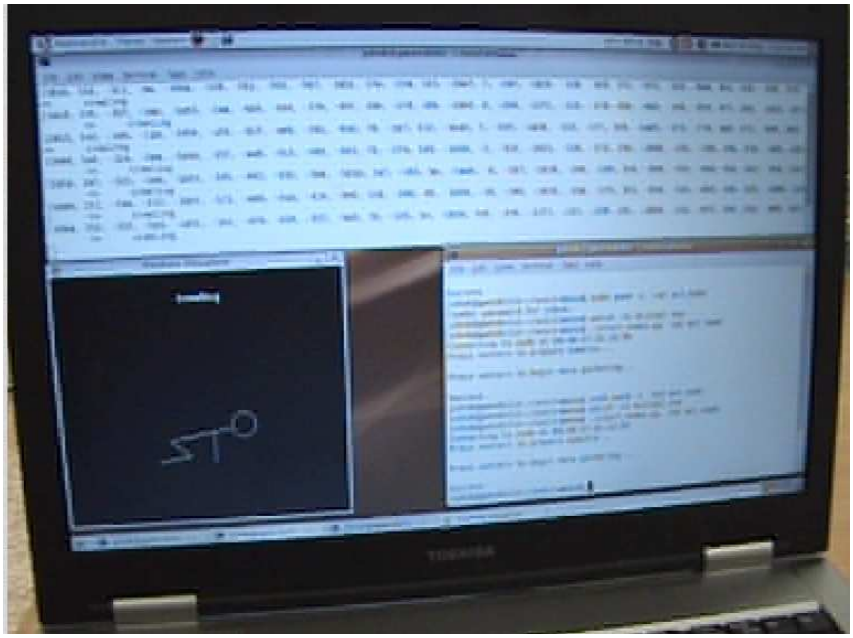


Figure 3.6: Data gathering

Chapter 4

Results and Evaluation

4.1 Experiment setup

For experimentation the postures described previously were recreated by a subject. The sensors were attached using PVC tape in the positions specified. The subject then undertook an activity regime composed of sitting, standing, walking, kneeling, crawling, laying one side, laying down on front, and laying down on back. Each posture was maintained for 2 minutes with a break between them to allow repositioning. Accelerometer data was taken via the prototype wireless sensing system as described previously.

4.2 Initial experimental runs

4.2.1 Single accelerometer example

The graphs in figure 4.1 show the data collected from one accelerometer positioned on the lower leg while the subject moved through the eight postures. The data shows the combination of the actual acceleration of the subject's movement and the gravity that the device is experiencing at that moment.. The data shows considerable difference between postures, showing that this data is suitable for differentiating them. For the walking and crawling postures it can be observed that we have the acceleration from the movement of the subject in addition to the gravitational acceleration. It can be observed that there are greater differences in the acceleration when the subject is walking than when crawling.

The size of data was up to 22 bytes/sample. At 2 samples per second this gives up to 2640 bytes/minute.

4.2.2 All accelerometers

The graphs in figure 4.2 show the acceleration readings taken at 10 samples/second on the lower body using four accelerometers. The graphs are presented in the order right lower leg (node 0), right upper leg (node 1), left upper leg (node 2), and left lower leg (node 3). The graphs presented in figure 4.3 are

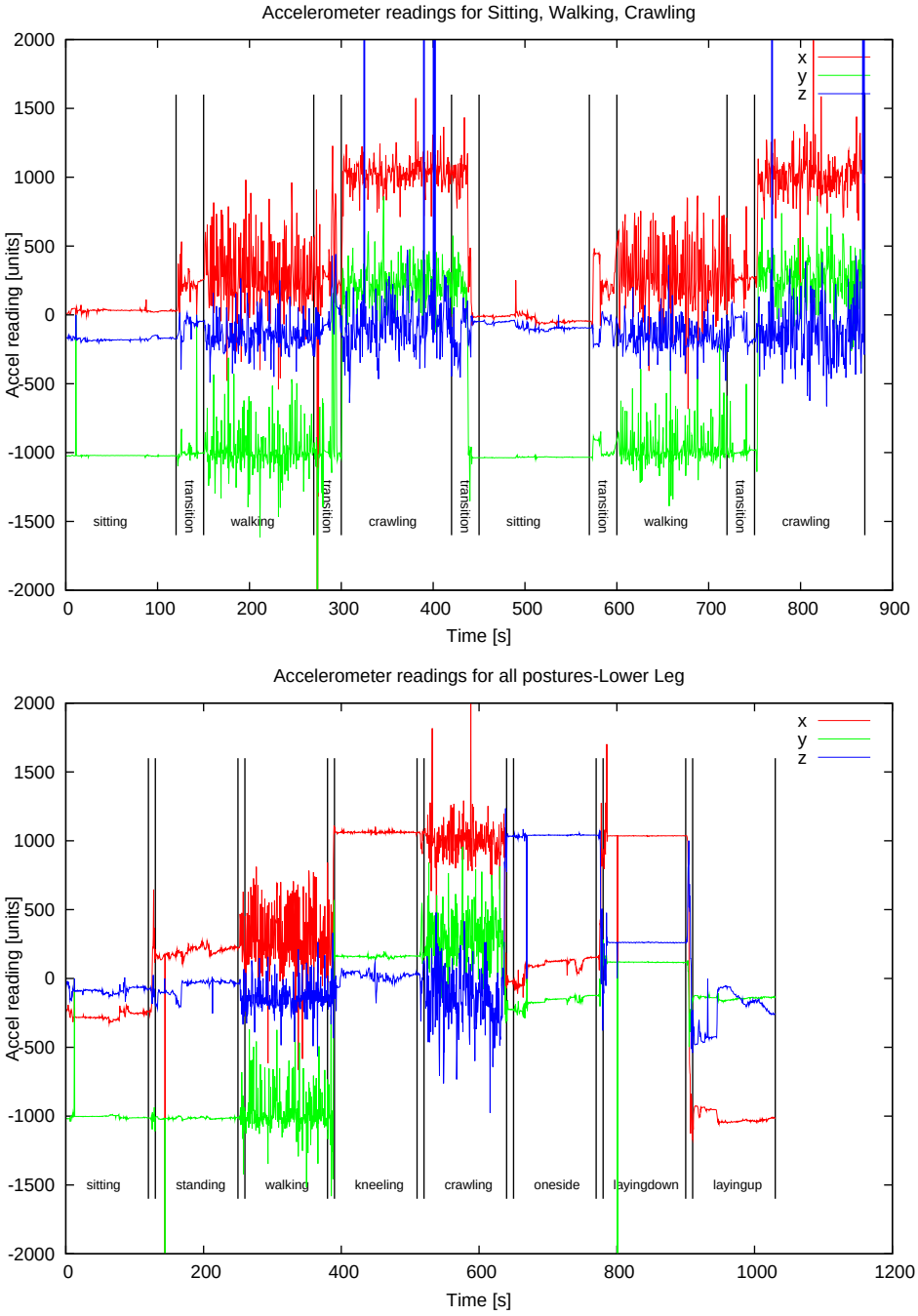


Figure 4.1: x, y and z axis data for various postures

for the upper body taken from five accels at 10 samples/second. The graph are presented in the order left forearm (node 0), left upper arm (node 1), chest (node 2), right forearm (node 3), right upper arm (node 4). From the graphs it can be seen that the accelerometers give readings that are different enough between postures to support the decision-making process. This experiments were proceed to be used in the decision making tree and data analysis.

4.3 System in use

For the evaluation the sensors were mounted on the subject using PVC tape as described previously. The subject performed various tasks in order to evaluate the system stability, the communication range, the collection of data, and the correspondence of the visualizer with the data taken. The subject also performed the routine discussed previously, composed of sitting, standing, walking, kneeling, crawling, laying on one side (oneside), laying on front (layingdown), and laying on back (layingup) for the purpose of data gathering and the evaluation of the effectiveness of the data processing. The routine was sustained for 2 minutes for each position with a small break between them to allow for repositioning, though due to technical problems only one minute of data was collected for the layingdown posture.

The gathered data was processed as described in chapter 3 and the results are shown below. Figure 4.5 shows the placement of the sensors on the arms and legs. Figure 4.6 shows a close look at the sensors placed on various body segments. In figure 4.4 it is shown a processing node that was placed on the subject body.

4.3.1 Data gathering results

The confusion matrix in table 4.2 shows the results of the data gathering for the evaluation experiment. The processing step is shown to be largely accurate, detecting six of the postures correctly in real time. The confusion matrix is also demonstrated graphically in figure 4.7, with lighter squares showing the concentration of results. Due to the range of values that walking takes during the data gathering, it is very hard to determine postures very close to walking, such as standing. The incorrect results for kneeling are expected to be an artifact of decision tree generation and should not be present if larger quantities of data are made available for the generation of future trees. While it is not shown in the confusion matrix for this experiment, the decision tree was seen to be very sensitive to limb positioning when the subject was in the sitting posture in previous experimentation. This is also considered to be an artifact of the data collected for tree generation: the subject was very still while sitting, meaning that the data stayed within very narrow bounds with the result that deviation from this was not expected by the tree.

The percentages of correct and incorrect classifications for the data gathered is shown in table 4.1. The data was 72.06% correctly classified and 27.94% incorrectly classified

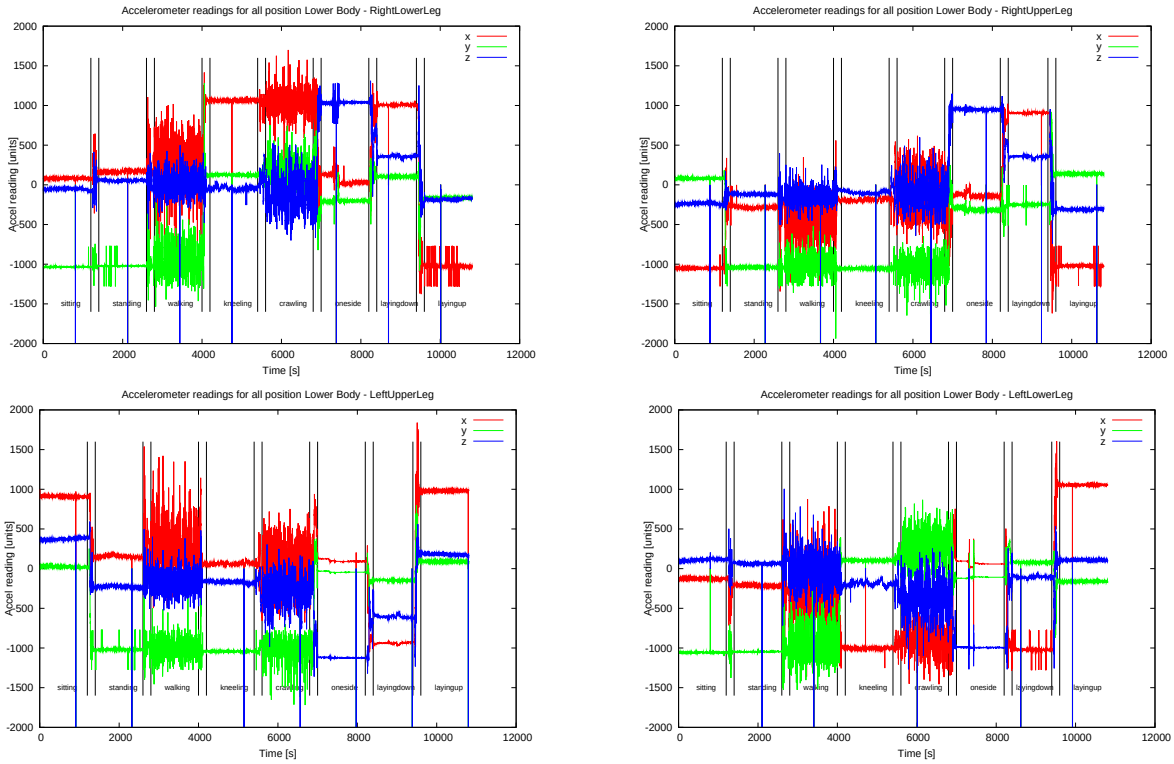


Figure 4.2: Accelerometer readings for all the lower body segments

Postures	Correct classified	Incorrect classified
Sitting	93.97%	6.03%
Standing	0%	100%
Walking	99.58%	0.42%
Kneeling	0.09%	99.01%
Crawling	99.91%	0.09%
Onseside	98.08%	1.92%
Layingdown	97.67%	2.33%
Layingup	99.83%	0.17%
All postures	72.06%	27.94%

Table 4.1: Classification percentage

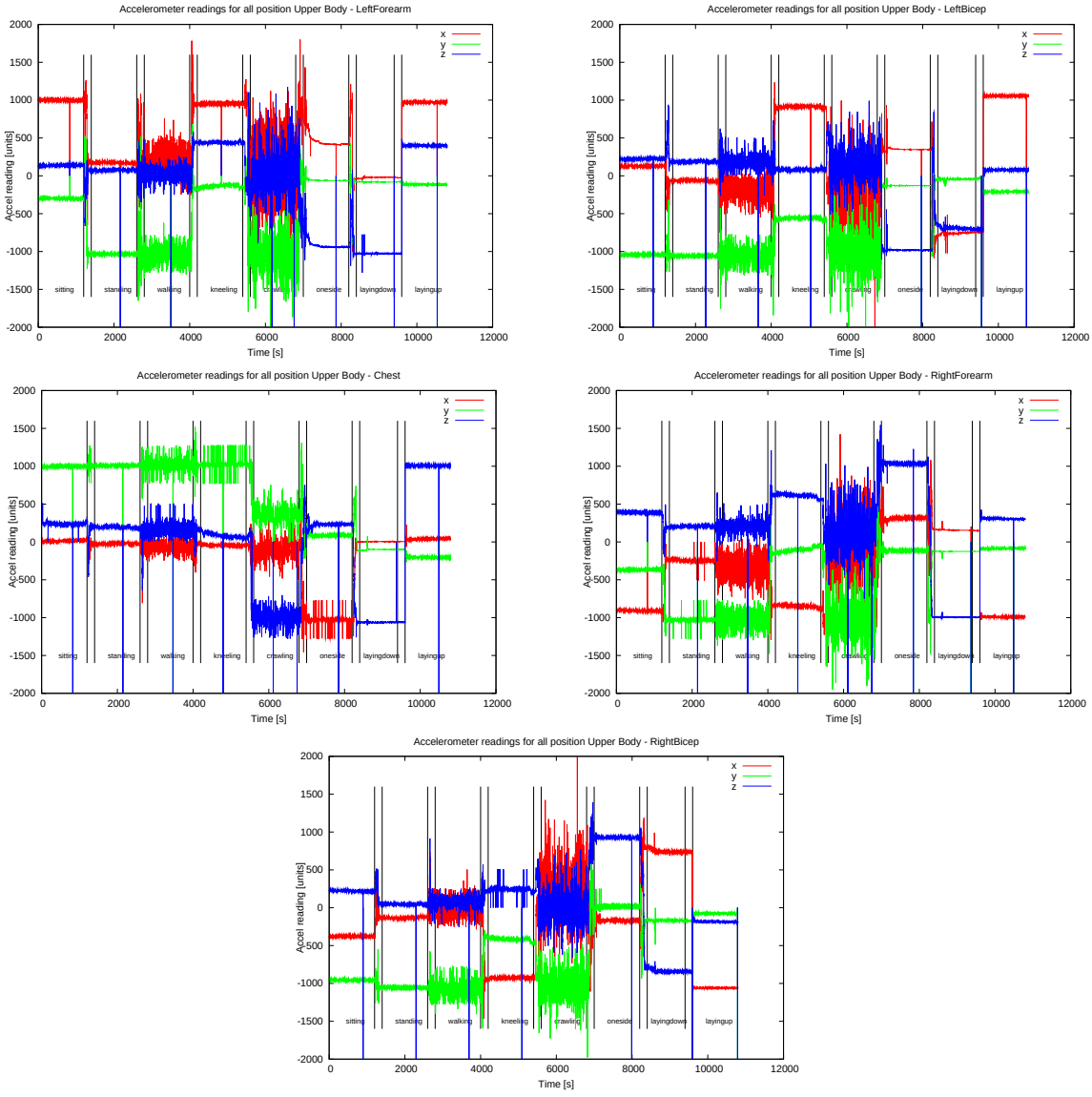


Figure 4.3: Accel reading for all the upperbody part

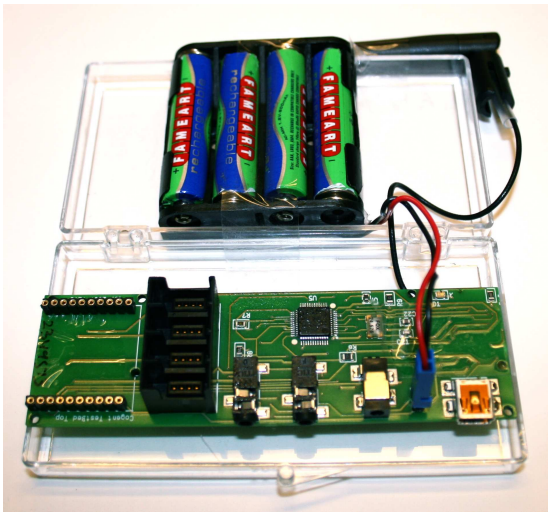


Figure 4.4: Processing node

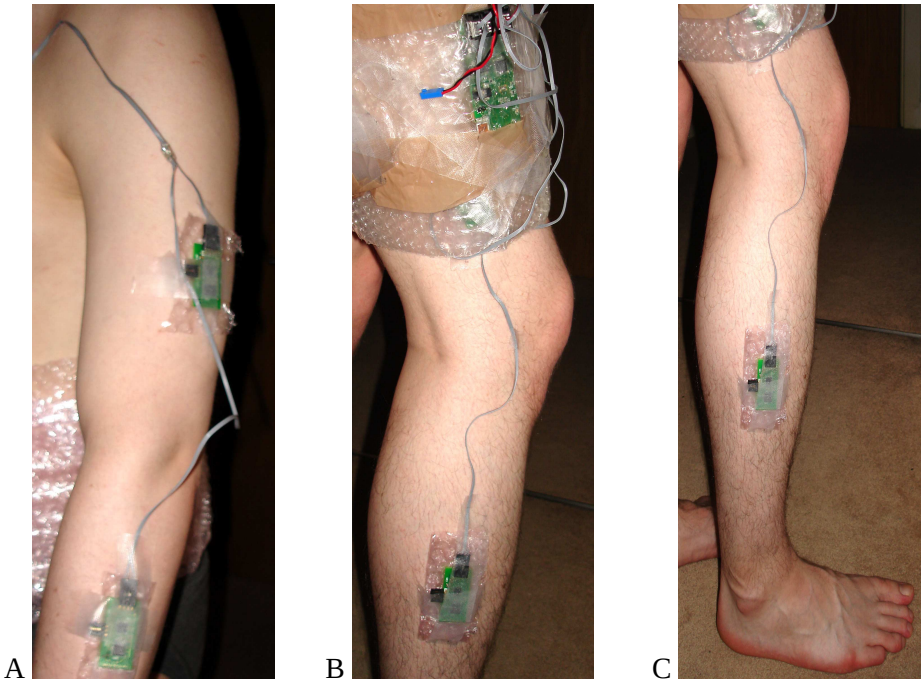


Figure 4.5: Sensor placement for: A) arms, B+C) legs

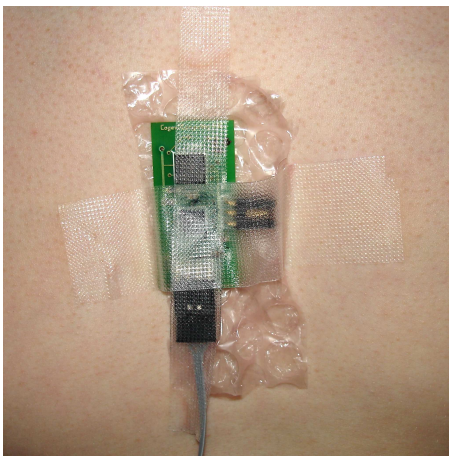
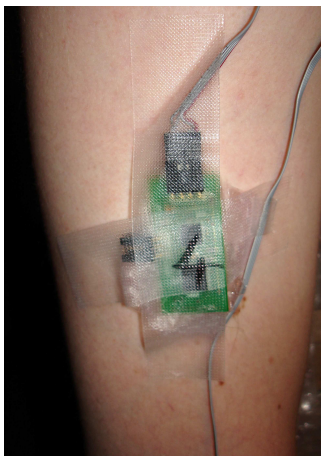
A  B  C 

Figure 4.6: Placing the sensor on the subject - *A) leg B) chest C) arm*

Posture was...

sitting	standing	walking	kneeling	crawling	oneside	layingdown	layingup	
1123	0	1	0	0	0	0	1	sitting
0	0	3	3	0	19	0	0	standing
0	1194	1195	1194	1	0	0	0	walking
0	0	1	1	0	1	0	0	kneeling
0	0	0	2	1199	2	14	1	crawling
0	0	0	0	0	1176	0	0	oneside
0	0	0	0	0	1	586	0	layingdown
72	0	0	0	0	0	0	1198	layingup

Posture was identified as...

Table 4.2: Confusion matrix for evaluation data

Graphs of the evaluation data for the lower part of the body (left and right legs) are shown in figure 4.8. The graphs for the upper part of the body (chest and arms) are shown in figure 4.9.

4.3.2 Tree regeneration

As noted previously, there are several results generated by the tree which do not reflect the actual posture of the subject during the experimentation. Due to this, a new tree was generated from the data gathered during the new experimentation, as it represented a wider range of possible data values. For instance, the subject no longer maintained their arms in a fixed position in certain postures. This should give the arms less priority in the decision where they are not quite as relevant (such as sitting). The new tree can

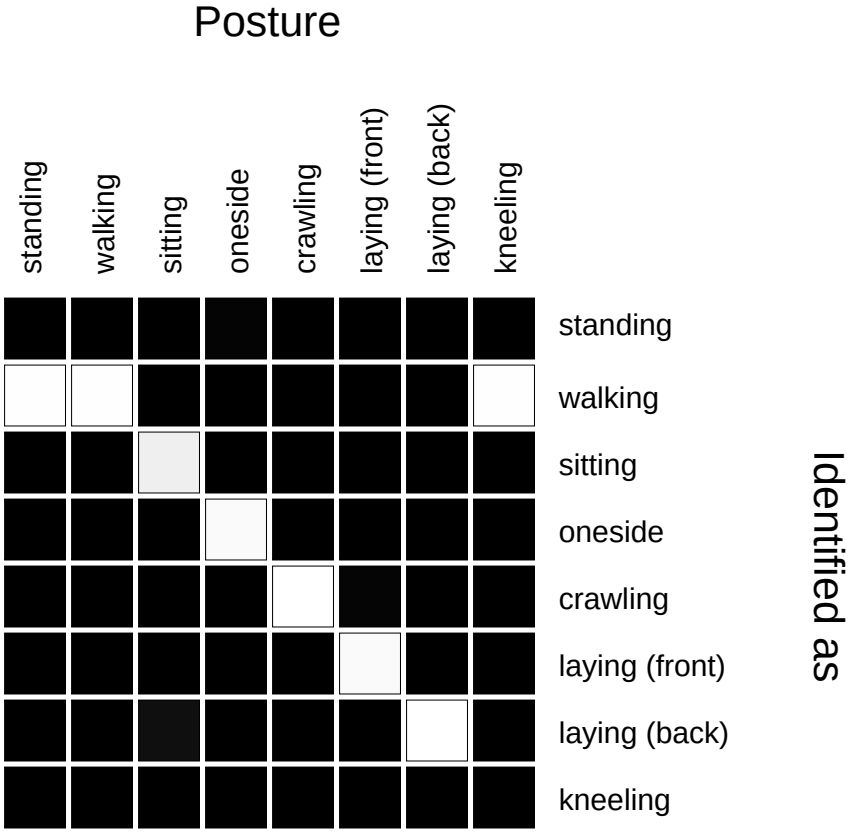


Figure 4.7: Confusion matrix of the system evaluation

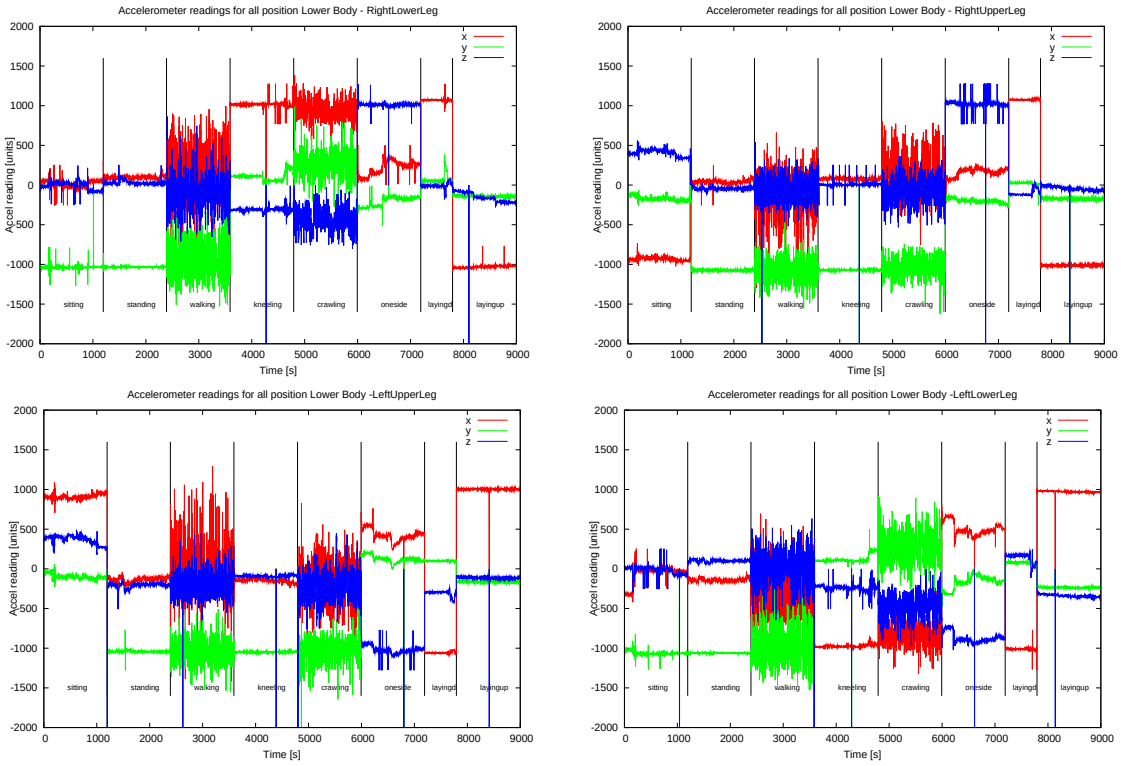


Figure 4.8: Accelerometer reading for the lower body segments for system evaluation

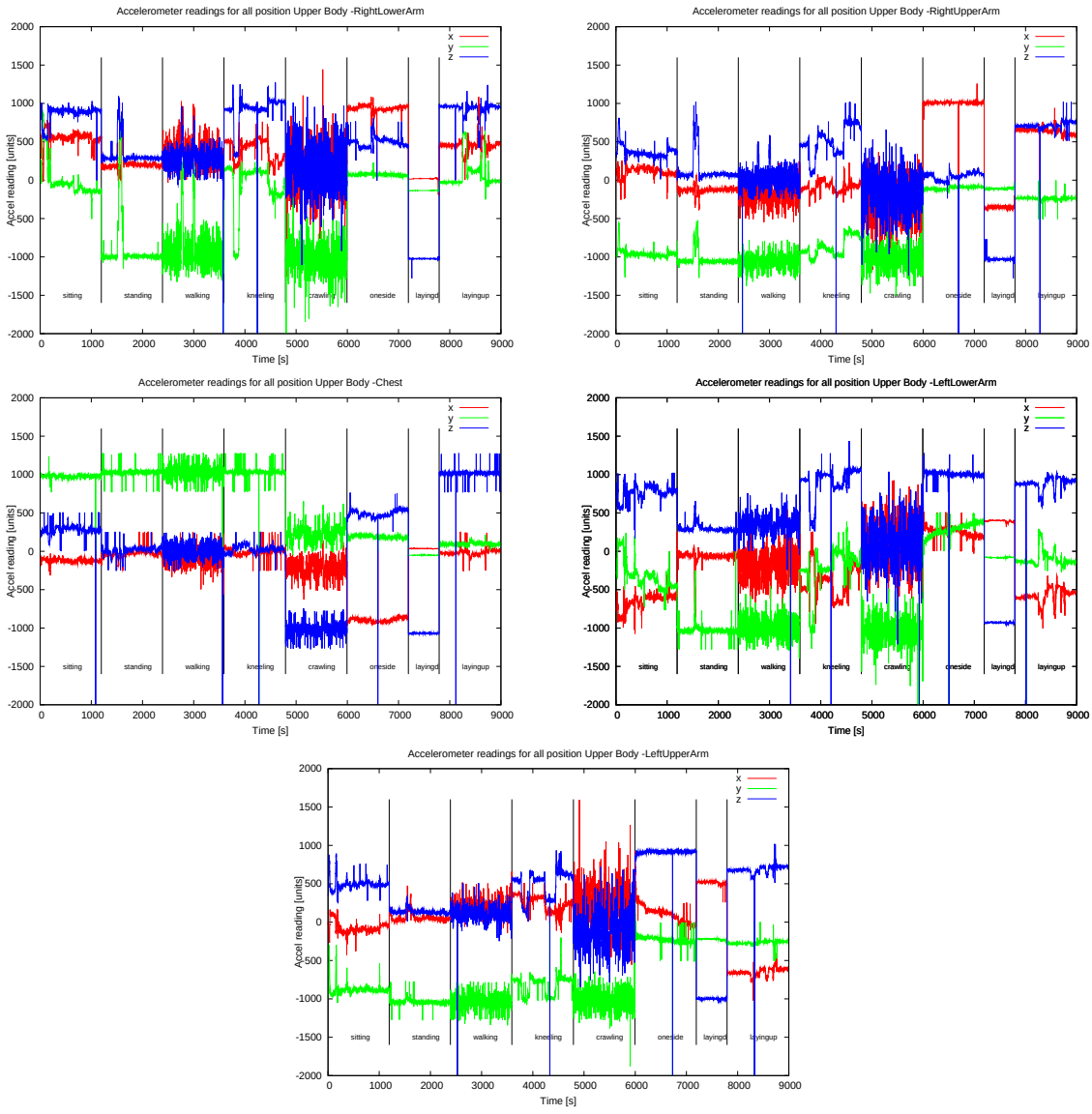


Figure 4.9: Accelerometer reading for the upper body segments for system evaluation

be seen in listing 4.1, and it can be seen that a lot of the upper-level decisions are made based more on the legs and chest than on the arms. The results of the decisions made by this tree for the data collected previously are shown in table 4.3. These results show that it does not only make correct decisions for the training data used. The problem differentiating between walking and standing is still present, though the problem with identifying kneeling has been solved. There does appear to be a problem with identifying a subject laying on one side. Training a new tree on all available data would seem to provide a compromise between the two trees presented here.

Listing 4.1: New tree based on evaluation data

```
J48 pruned tree for the new data
------------------------------


legrdy <= -473
|   bodyz <= -743: crawling (1202.0/2.0)
|   bodyz > -743
|   |   legruy <= -1
|   |   |   legruz <= -1
|   |   |   |   legruz <= -5: walking (982.0/2.0)
|   |   |   |   legruz > -5
|   |   |   |   |   armruz <= 229
|   |   |   |   |   |   armrux <= 59: standing (2.0)
|   |   |   |   |   |   armrux > 59: walking (11.0/1.0)
|   |   |   |   |   armruz > 229: kneeling (2.0)
|   |   |   legruz > -1
|   |   |   |   legluy <= -1070
|   |   |   |   |   legluy <= -1074: walking (124.0)
|   |   |   |   |   legluy > -1074
|   |   |   |   |   |   armldx <= 247: standing (2.0)
|   |   |   |   |   |   armldx > 247: walking (7.0)
|   |   |   |   legluy > -1070
|   |   |   |   |   legluy <= -1054
|   |   |   |   |   |   legrdy <= -1060
|   |   |   |   |   |   |   legrdy <= -1082
|   |   |   |   |   |   |   |   legrdy <= -1085: walking (9.0/1.0)
|   |   |   |   |   |   |   |   legrdy > -1085: standing (9.0)
|   |   |   |   |   |   |   legrdy > -1082
|   |   |   |   |   |   |   |   legrdz <= -82
|   |   |   |   |   |   |   |   |   legluy <= -1063: walking (2.0)
|   |   |   |   |   |   |   |   |   legluy > -1063: standing (13.0)
|   |   |   |   |   |   |   |   legrdz > -82: standing (1165.0)
|   |   |   |   |   |   legrdy > -1060
|   |   |   |   |   |   |   armrdx <= -28: walking (21.0)
|   |   |   |   |   |   |   armrdx > -28
|   |   |   |   |   |   |   |   legrdx <= -173: walking (2.0)
|   |   |   |   |   |   |   |   legrdx > -173: standing (2.0)
|   |   |   |   |   legluy > -1054: walking (44.0)
|   |   legruy > -1: kneeling (1195.0)
legrdy > -473
|   legldx <= 763
```

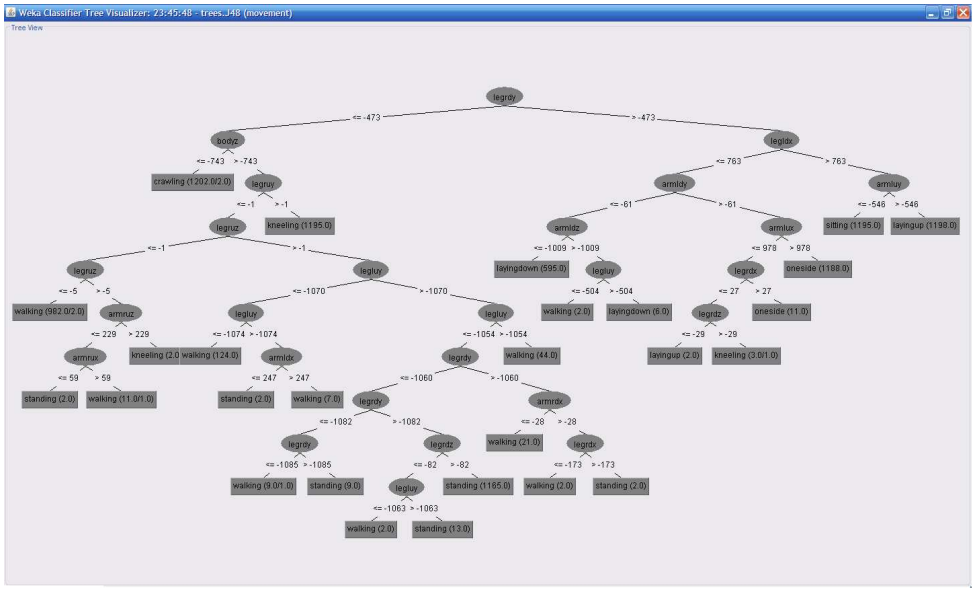


Figure 4.10: New tree generated with Weka

```
|   |   armldy <= -61
|   |   |   armldz <= -1009: layingdown (595.0)
|   |   |   armldz > -1009
|   |   |   |   legluy <= -504: walking (2.0)
|   |   |   |   legluy > -504: layingdown (6.0)
|   |   armldy > -61
|   |   |   armlux <= 978
|   |   |   |   legrdx <= 27
|   |   |   |   |   legrdz <= -29: layingup (2.0)
|   |   |   |   |   legrdz > -29: kneeling (3.0/1.0)
|   |   |   |   legrdx > 27: oneside (11.0)
|   |   |   armlux > 978: oneside (1188.0)
|   legldx > 763
|   |   armluy <= -546: sitting (1195.0)
|   |   armluy > -546: layingup (1198.0)
```

The new tree is shown in figure 4.10.

Posture was...								
sitting	standing	walking	kneeling	crawling	oneside	layingdown	layingup	Posture was identified as...
1197	0	0	0	0	0	0	0	sitting
0	0	24	0	0	0	0	0	standing
2	1198	1164	0	1	0	0	0	walking
0	2	3	1192	8	407	0	0	kneeling
0	1	2	2	1187	0	2	2	crawling
0	0	0	0	0	0	0	0	oneside
0	0	0	2	2	791	996	2	layingdown
2	0	0	0	0	0	0	1196	layingup

Table 4.3: Confusion matrix for older data using new tree

4.4 System evaluation

4.4.1 Network evaluation

Communication range

The communication range of the system is important due to the ability to communicate gathered data to a remote point is often useful where an observer is unable to be in the same physical location as the subject. This also means the the subject is not constrained within a particular location. In order to test this parameter, a subject wearing the prototype system was instructed to walk slowly away from the base station, and the distance was recorded at which communication was noticeably affected.

The communication range for a Bluetooth class 2 device is approximately 10 meters working at a maximum permitted power of 2.5mW. The results for this test were that the system communicated successfully at a range of up to 11.5m.

Bandwidth

Bandwidth refers to a data rate measured in bits per second (sometimes bytes), and is used often to describe network throughput. It is important to find the bandwidth due this can restrict the data transfer between nodes. A low bandwidth may result in data buffering due to the new data not being transmitted fast enough. This will in turn result in delays receiving new data for processing. Bandwidth is calculated using:

$$\text{bandwidth (bits/s)} = \frac{\text{data sent (bits)}}{\text{time required (s)}}$$

The method used here of measuring the throughput is to transfer a large (5MB) file and record the time required to do that.

The theoretical maximum transmission data rates for Bluetooth are shown in table 4.4. The results found are shown in table 4.5.

Version	Data Rate	Effective transfer rate
Version 1.2	1Mbit/s	721Kb/s
Version 2.0 + EDR	3Mbit/s	2.1Mb/s

Table 4.4: Theoretical transmission rate

Transfer time	1 min 15 secs
Transfer rate	546.13Kb/s

Table 4.5: Discovered transmission rate

Data loss

Data loss is a measure of the quantity of data received compared with the quantity of data sent.

$$\text{data loss} = \frac{\text{data received}}{\text{data sent}}$$

This will be measured by trying to gather data at various rates starting at 10 samples/seconds. This is due to the faster the data is gathered by the system, the greater the stress it is placed under (retrieving data from the sensors at a faster rate, transmitting more data, etc). The results of this test are shown in table **??**. Across the range of sample rates measured there was no observed data loss. Using the current system, data cannot be gathered at more than approximately 65 samples/sec. No data loss is sustained at this rate, and any faster attempt at data gathering results in only approximately 65 samples/sec being retrieved.

Latency

This measurement refers to the latency of the connection between the node and the base station. Checking this will be carried out by performing 10 pings of each node. A ping measures the round-trip time for a packet of data to the remote node and back again, meaning that a synchronized time source between nodes is not required. The results are shown in table 4.6.

	Minimum	Average	Maximum
Round-trip time	15.68ms	26.84ms	35.75ms

Table 4.6: Latency

4.4.2 Computing evaluation

Code size

The sizes of the various pieces of code for each node in the system are shown in table 4.7. Note that as Python was used as the development language, space is required for the interpreter and additional libraries.

Component	Code size
Master node	5.4KB + 1.9KB of configuration data
Slave node	1.8KB
Base station	4.5KB

Table 4.7: Code size

Processor usage

The processor usage for the base station is negligible as the vast majority of its time is spent waiting for data to be received. The processor usage for the master and slave nodes is approximately 100% due to the use of busy waiting loops in the code. While this will in theory shorten battery life, it produces easier to understand code and the battery life (see section 4.4.3) is far in excess of that required for experimentation currently.

4.4.3 Ease of use

Comfort

The subject can move freely without any restriction within the range of the Bluetooth connection, and assuming the current posture does not place pressure upon the processing nodes (the prototype system does not include sufficient protection for them to withstand this). The sensors are easy to place as they are largely flat and this allows for them to be taped to a subject with minimal effort. There are some slight sharp points on the underside of the sensor nodes, though these can be removed and do not pose a major problem.

Setup time

Experimentation showed that the system can be set up in 18 minutes. This time includes the sensors being placed on the subject's body and wired to the processing nodes. Data gathering can being within 20 minutes.

Battery life

The results for the current draw of the system are shown in table 4.8. This test was carried out using a 4.8v source. If four 800mAh batteries are used (3.2Ah total) and if the system is activated and fully working constantly (drawing 0.376A), the batteries should last for approximately 8 and a half hours.

State	Devices	Current draw
System stopped	Gumstix + expansion board	0.198A
System stopped	Gumstix + expansion board + 5 sensor boards	0.286A
System activated	Gumstix + expansion board + 5 sensor boards	0.376A

Table 4.8: Current draw

4.5 Future work

Future work for this project includes establishing a firm difference between the dynamic postures (such as walking) and the static postures (such as standing). One method of this is using a Fast Fourier Transform (FFT), as discussed in section 4.5.1. Another optimization would be to build a better decision tree by the considering of more data over a wider range of natural stances.

In practical cases, the maximum frequency of human movement is 30Hz. This means that the gathering of 60 data points per second (time between samples = 16.7msec) is a sufficient rate for performing full motion capture without losing information. This would also potentially allow for filtering techniques which require greater quantities of readings. The data so far was gathered only at 10 sample/seconds and in the future it may be beneficial to gather at the 60 sample/seconds rate.

4.5.1 FFT

Fast Fourier Transforms (FFTs) are a mathematical tool used for processing a signal that varies with respect to time. The transform accomplishes this by breaking down the original time-based waveform into a series of sinusoidal terms, each with unique magnitude, frequency, and phase. This process, in effect, converts a waveform in the time domain that is difficult to describe mathematically into a more manageable series of sinusoidal functions that when added together, exactly reproduce the original waveform. Plotting the amplitude of each sinusoidal term versus its frequency creates a power spectrum, which is the response of the original waveform in the frequency domain.

The use of FFTs was briefly investigated for distinguishing between similar static and dynamic postures types (such as standing and walking). Testing code was implemented in Python. The results for the data collected are shown in figure 4.11 for walking, sitting and crawling using an accelerometer placed on the lower right leg. As can be seen, the dynamic movement produces an immediately apparent spike on the relevant graph meaning that this method could be useful for additional processing where appropriate. Future work could integrate this with the decision trees in order to provide greater accuracy in determining the posture of a subject.

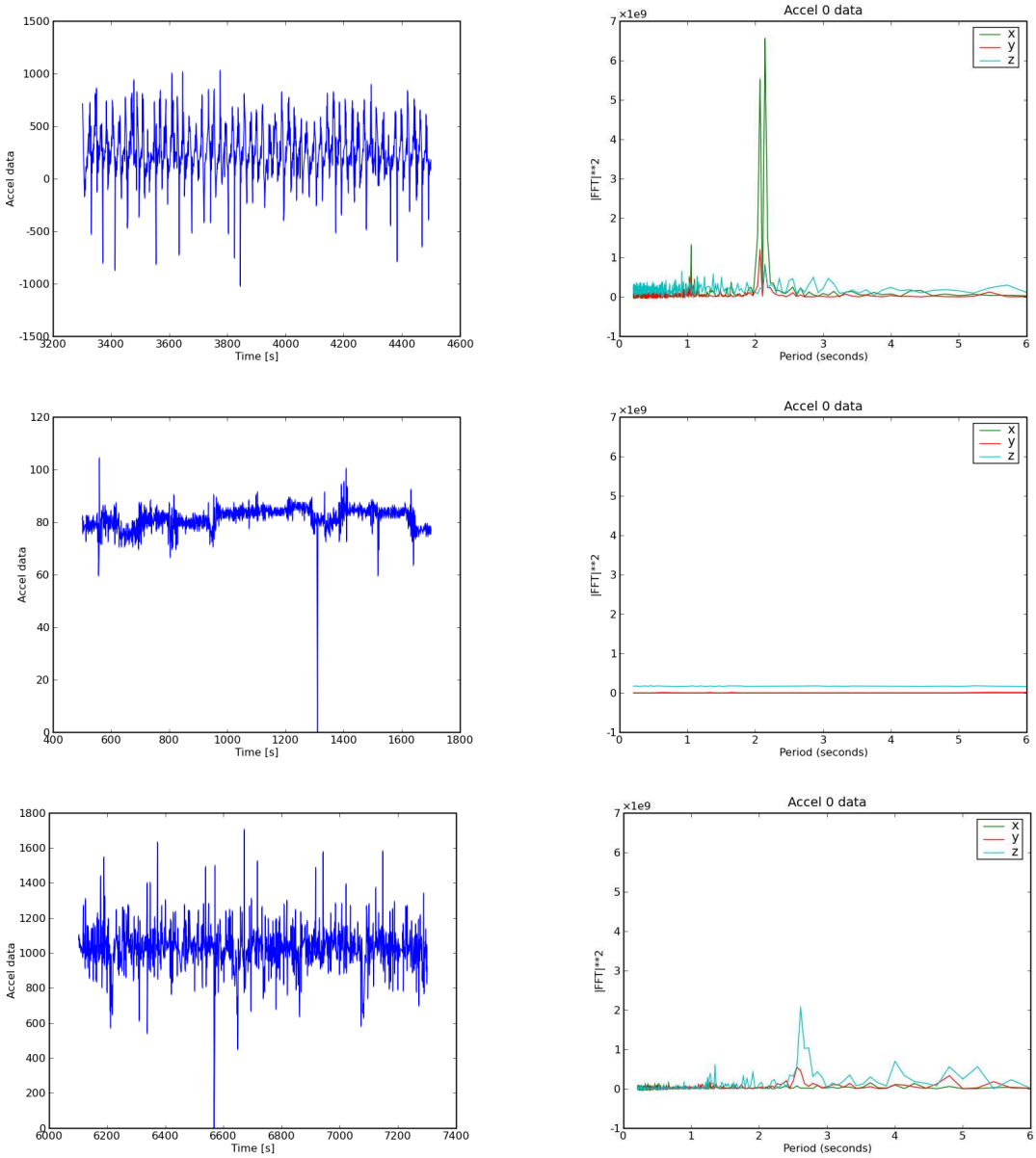


Figure 4.11: FFT output for walking, sitting and crawling

Bibliography

[1] A. Gallagher, YokyMatsuoka, Wei-Tech Ang, The Robotic Institute Carnegie Mellon University: "An efficient Real-Time Human Posture Tracking Algorithm Using Low Cost Inertial and Magnetic Sensors" Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendal, Japan, September28-Octomber2, 2004 [Online] http://www.ri.cmu.edu/pub_files/pub4/gallagher_anthony_t_2004_1/ gallagher_anthony_t_2004_1.pdf

[2] David Fontaine, Dominique David, Yanis Caritu: “Sourceless Human Body Motion Capture” [Online] http://www.minatec.com/grenoble-soc/proceedings03/Pdf/30-fontaine.pdf

[3] Huiyu Zhou, Huosheng Hu: “A Survey-Human Movement Tracking and Stroke Rehabilitation” Technical Report: CSM-420 ISSN 1744-8050, 8 December 2004 [Online] http://cswww.essex.ac.uk/staff/hhu/Papers/CSM-420.pdf

[4] Alex Young, University if Edinburgh: “Distributed Posture Tracking” [Online] http://www.specknet.org/publications/Alex_Young.pdf

[5] Bodo Rosenhahn, Uwe G. Kersting, Lei He, Andrew W. Smith, Thomas Brox, Reinhard Klette, Hans-Peter Seidel: “A Silhouette Based Human Motion Tracking System” [Online] http://www.mpi-inf.mpg.de/~rosenhahn/publications/CITR-TR-164.pdf

[6] Elisabetta Farella, Augusto Pieracci, Davide Brunelli, Andre Acquaviva, Luca Benini, Bruno Ricco, DEIS-University of Bologna, Italy: “Design and Implementation of WiMoCA Node for Body Area Wireless Sensor Network” [Online] http://www.vs.inf.ethz.ch/events/dag2005/program/ws/farella-2.pdf

[7] Elisabetta Farella, Augusto Pieracci, Davide Brunelli, Luca Benini : ”A Wireless Body Area Sensor Network for Posture Detection” Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC’06)

[8] The Gumstix-Platform Sensor I/O System (SIOS) Acc/Mag Module [Online] http://svn.v2.nl/embed/sios/trunk/Documentation/Acc-Mag_Module.rtf

[9] Subir Biswas1, Muhannad Quwaider : “Body Posture Identification using Hidden Markov Model with Wearable Sensor Networks” Proocedings of the Third International Conference on Body Area Networks, Tempe, Arizona, March 13-17, 2008

[10] Hong Ying, C. Silex, A. Schnitzer, S. Leonhardt and M. Schiek: “Automatic Step Detection in the Accelerometer Signal” Proceedings of 4th International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2007) March 26 – 28, 2007 RWTH Aachen University, Germany

[11] Julien Pansiot, Danail Stoyanov, Douglas McIlwraith, Benny P.L. Lo and G. Z. Yang: “Ambient and Wearable Sensor Fusion for Activity Recognition in Healthcare Monitoring Systems” Proocedings of the Third International Conference on Body Area Networks, Tempe, Arizona, March 13-17, 2008

[12] Kemp, J., Gaura, E.I., Brusey, J. (2008) ’Instrumenting Bomb Disposal Suits with Wireless Sensor Networks.’ In Proceedings of the 5th International Conference on Informatics in Control, Automation and Robotics (ICINCO2008). Held May 11-15 2008 at Funchal, Madeira - Portugal.

Appendix A

Calibration for one accelerometer

Calibration done in Mathcad

The results are:

$c_x := 1 \qquad m_x := 1$

$c_y := 1 \qquad m_y := 1$

$c_z := 1 \qquad m_z := 1$

Given

$$\left(\frac{1072-c_x}{m_x}\right)^2+\left(\frac{-13-c_y}{m_y}\right)^2+\left(\frac{-37-c_z}{m_z}\right)^2=1$$

$$\left(\frac{-1056-c_x}{m_x}\right)^2+\left(\frac{-20-c_y}{m_y}\right)^2+\left(\frac{1-c_z}{m_z}\right)^2=1$$

$$\left(\frac{54-c_x}{m_x}\right)^2+\left(\frac{1029-c_y}{m_y}\right)^2+\left(\frac{4-c_z}{m_z}\right)^2=1$$

$$\left(\frac{19-c_x}{m_x}\right)^2+\left(\frac{-1040-c_y}{m_y}\right)^2+\left(\frac{42-c_z}{m_z}\right)^2=1$$

$$\left(\frac{49-c_x}{m_x}\right)^2+\left(\frac{15-c_y}{m_y}\right)^2+\left(\frac{1060-c_z}{m_z}\right)^2=1$$

$$\left(\frac{12-c_x}{m_x}\right)^2+\left(\frac{-35-c_y}{m_y}\right)^2+\left(\frac{-1041-c_z}{m_z}\right)^2=1$$

$$\begin{pmatrix} c_{xval} \\ c_{yval} \\ c_{zval} \\ m_{xval} \\ m_{yval} \\ m_{zval} \end{pmatrix} := Find(c_x, c_y, c_z, m_x, m_y, m_z)$$

$c_{xval} = 8.469 \qquad m_{xval} = 1.065 * 10^3$

$c_{yval} = -5.287 \qquad m_{yval} = 1.035 * 10^3$

$c_{zval} = 9.763$ $m_{xzval} = 1.051 * 10^3$

Evaluation of results for g and g/2:

$x_{cal} := \left(\frac{1072 - o_{xval}}{s_{xval}}\right)$ $x_{cal} = 0.999$

$y_{cal} := \left(\frac{1029 - o_{ycal}}{s_{yval}}\right)$ $y_{cal} = 0.999$

$z_{cal} := \left(\frac{1060 - o_{zcal}}{s_{zval}}\right)$ $z_{cal} = 0.999$

$x_{calhalf} := \left(\frac{538 - o_{xval}}{s_{xval}}\right)$ $x_{calhalf} = 0.497$

$y_{calhalf} := \left(\frac{545 - o_{yval}}{s_{yval}}\right)$ $y_{calhalf} = 0.532$

$z_{calhalf} := \left(\frac{518 - o_{zval}}{s_{zval}}\right)$ $z_{calhalf} = 0.483$

Appendix B

Training data Weka

B.1 Run information

Scheme: weka.classifiers.trees.J48 -C 0.25 -M 2

Relation: movement

Instances: 9390

Attributes: 28

legrux

legruy

legruz

legrdx

legrdy

legrdz

legldx

legldy

legldz

leglux

legluy

legluz

armldx

armldy

armldz

armlux

armluy

armluz

bodyx

bodyy

bodyz

armrdx

armrdy

armrdz

armrux

armruy

armruz

position

Test mode: 10-fold cross-validation

B.2 Classifier model (full training set)

J48 pruned tree

```
legrdy <= -518
|   armrux <= -884: kneeling (1193.0/1.0)
|   armrux > -884
|   |   bodyy <= 752
|   |   |   bodyy <= 33
|   |   |   |   legrux <= 625
|   |   |   |   |   legrux <= 190: standing (3.0)
|   |   |   |   |   legrux > 190: walking (3.0)
|   |   |   |   legrux > 625: crawling (10.0/1.0)
|   |   |   bodyy > 33: crawling (1187.0)
|   |   bodyy > 752
|   |   |   armrux <= -106
|   |   |   |   armrdx <= -265
|   |   |   |   |   legluz <= 59: walking (212.0)
|   |   |   |   |   legluz > 59
|   |   |   |   |   |   legluz <= 62: standing (5.0/1.0)
|   |   |   |   |   |   legluz > 62: walking (44.0/1.0)
|   |   |   |   armrdx > -265
|   |   |   |   |   armldx <= 219
|   |   |   |   |   |   armldx <= 155
```

```
|   |   |   |   |   |   |   legrdx <= -294: walking (5.0)
|   |   |   |   |   |   |   legrdx > -294: standing (13.0)
|   |   |   |   |   |   armldx > 155: standing (1175.0)
|   |   |   |   |   armldx > 219: walking (20.0)
|   |   |   armrux > -106: walking (919.0/6.0)
legrdy > -518
|   armrdx <= 36
|   |   armrux <= -667
|   |   |   armrdy <= -93
|   |   |   |   legrux <= 30: layingup (13.0)
|   |   |   |   legrux > 30: kneeling (2.0)
|   |   |   armrdy > -93: layingup (1184.0)
|   |   armrux > -667
|   |   |   legruy <= -1029: sitting (1191.0)
|   |   |   legruy > -1029
|   |   |   |   armldx <= 984
|   |   |   |   |   legrdx <= -69: layingup (3.0/1.0)
|   |   |   |   |   legrdx > -69: standing (3.0/2.0)
|   |   |   |   armldx > 984: sitting (10.0)
|   armrdx > 36
|   |   armlux <= -323: layingdown (996.0)
|   |   armlux > -323
|   |   |   legldz <= -1119: oneside (1181.0)
|   |   |   legldz > -1119
|   |   |   |   legluz <= -613: oneside (16.0)
|   |   |   |   legluz > -613: crawling (2.0)
```

Number of Leaves : 24

Size of the tree : 47

Time taken to build model: 1.53 seconds

B.3 Stratified cross-validation

B.3.1 Summary

Correctly Classified Instances	9332 (99.3823%)
Incorrectly Classified Instances	58 (0.6177%)
Kappa statistic	0.9929
Mean absolute error	0.0017
Root mean squared error	0.0381
Relative absolute error	0.7668%
Root relative squared error	11.5186%
Total Number of Instances	9390

B.3.2 Detailed Accuracy By Class

TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area	Class
0.997	0.001	0.992	0.997	0.994	0.999	sitting
0.991	0.002	0.988	0.991	0.99	0.997	standing
0.982	0.001	0.99	0.982	0.986	0.992	walking
0.997	0	0.997	0.997	0.997	0.999	kneeling
0.994	0.001	0.994	0.994	0.994	0.998	crawling
0.998	0	0.997	0.998	0.997	0.999	oneside
0.997	0	1	0.997	0.998	0.998	layingdown
0.994	0.001	0.994	0.994	0.994	0.997	layingup

B.3.3 Confusion Matrix

a	b	c	d	e	f	g	h	<– classified as
1197	0	0	0	0	0	0	4	a=sitting
1	1190	9	0	1	0	0	0	b=standing
5	14	1177	0	2	0	0	0	c=walking
0	0	0	1193	1	1	0	1	d=kneeling
0	0	3	2	1191	2	0	0	e=crawling
0	0	0	0	1	1196	0	1	f=oneside
0	0	0	0	2	0	995	1	g=layingdown
4	0	0	2	0	1	0	1193	h=layingup

Appendix C

Fast Fourier Transform (Python)

```
import pylab
import scipy
import scipy.io.array_import
import sys
import scipy.fftpack


"""
    Generate an FFT (in terms of both frequency and period)
              using the supplied data.
    Inputs
        data            The data to be transformed via FFT
        sample_freq     The sampling frequency of the supplied data
                        (default = 1 sample per second)
    Outputs
        A tuple containing the frequency, period, and power components

    Based on example code from http://linuxgazette.net/115/andreasen.html
"""
def fft(data, sample_freq = 1.):
    fft = scipy.fft(data)
    fft_len = len(fft)
    power = abs(fft[1:(fft_len / 2)])**2
    nyquist = sample_freq / 2
    freq = scipy.array(range(fft_len / 2)) / (fft_len / (2.0 * sample_freq)) * nyquist
    period = sample_freq / freq
    return (freq, period, power)


"""
    Add a generated FFT to the current graph.
    Inputs
        fftdata         Data for the FFT to be plotted (as returned from the
                        fft function)
        name            A name to use in the graph legend
"""
```

```
def plotfft(fftdata, name = ""):
    freq = fftdata[0]
    period = fftdata[1]
    power = fftdata[2]
    pylab.plot(period[1:len(period)], power, label = name)


"""
    Output the current graph to a file
       (series are added to the graph via the plotfft function).
    Inputs
        filename        The file to output the graph to
        title           The title of the graph
        axis            Limits for the axes (default = None [auto-selected])
        legend          Whether to output a legend on the graph
                        (default = False)
"""
def outputgraph(filename, title = "", axes = None, legend = False):
    pylab.xlabel("Period (seconds)")
    pylab.ylabel("|FFT|**2")
    if title != "": pylab.title(title)
    if axes != None: pylab.axis(axes)
    if legend: pylab.legend()
    pylab.savefig(filename)


"""
    Clear the current graph, removing all plotted ffts.
"""
def cleargraph():
    pylab.clf()



if __name__ == "__main__":

    cleargraph()

    tempdata = scipy.io.array_import.read_array('walking.dat')

    time = tempdata[:,0]
    datax = tempdata[:,1]
    datay = tempdata[:,2]
    dataz = tempdata[:,3]
    pylab.xlabel("Time [s]")
    pylab.ylabel("Accel data")
    pylab.plot(time, datax)
    pylab.savefig('original.png')

    plotfft(fft(datax, 10.), "x")
    plotfft(fft(datay, 10.), "y")
    plotfft(fft(dataz, 10.), "z")
    outputgraph("fft.png", "Accel 0 data", [0, 6, -1e9, 7e9], True)
```

Appendix D

Tree implementation for Gumstix

```
#!/usr/bin/python


"""
        This module provides classification functionality based on a decision
        tree generated by the Weka tool.

        The functionality available is provided by the Tree class.
"""


import sys


# Recursively print a node and all of the tree descending from it.
def PrintNode(node, depth = 0):
        print "%s%s" % ("| " * depth, node)
        for child in node.children:
                if child != None:
                    PrintNode(child, depth + 1)


# Print an entire tree (this is an alias for calling PrintNode with the
# root node of a tree).
def PrintTree(tree):
        PrintNode(tree[0])


# This class represents a single node within a tree.
class TreeNode:
    def __init__(self, parent):
        self.parent = parent
        self.children = [None, None]
        self.position = ""

    def __repr__(self):
```

```
        if self.position != "":
            return "%s" % (self.position)
        else:
            return "%s %u" % (self.variable, self.value)

    def AddChild(self, node):
        self.children.append(node)


# This class represents an entire tree made up of TreeNode objects.
class Tree:
        def __init__(self):
                self.nodes = []

                # Maximum depth of tree = 20 nodes
                self.last_nodes = [None] * 20

                # Create a root node with no parents
                self.nodes.append(TreeNode(None))

                # Read in a configuration file specifying the variables used
                # in the tree (this also determines the ordering of data for
                # classification)
                conf_filename = "tree_conf/use_tree.conf"
                conf_file = open(conf_filename, "r")
                self.cols = {}
                index = 0
                for line in conf_file:
                        line_s = line.split()
                        if (len(line_s) > 0):
                                self.cols[line_s[0]] = index
                                index += 1
                conf_file.close()


        # Helper function to discover the current depth in a tree represented
        # with text
        def NumPipes(self, line):
                pipes = 0
                for element in line:
                    if element == '|': pipes += 1
                return pipes


        # Read in a tree from a file (only trees generated by Weka are currently
        # supported)
        def InterpretText(self, filename):
                in_file = open(filename, "r")

                for line in in_file:
                        line_s = line.split()

                        curr_depth = self.NumPipes(line_s)
```

```
                if curr_depth != 0:
                        # Most nodes have the last one we saw at the previous
                        # depth as the parent
                        parent_node = self.last_nodes[curr_depth - 1]
                else:
                        # Except the top-level one, which has our root node
                        # as its parent
                        parent_node = self.nodes[0]

                variable = line_s[curr_depth]
                condition = line_s[curr_depth + 1]

                value = line_s[curr_depth + 2]
                # Are we at a leaf node?
                if value[-1:] == ':': value = int(value[:-1])
                else: value = int(value)

                self.nodes.append(TreeNode(parent_node))

                this_node = self.nodes[len(self.nodes) - 1]

                if len(line_s) > curr_depth + 3:
                        this_node.position = line_s[curr_depth + 3]

                # Which child are we looking at?
                if condition == "<=":
                        parent_node.children[0] = this_node
                if condition == ">":
                        parent_node.children[1] = this_node

                parent_node.value = value
                parent_node.variable = variable

                self.last_nodes[curr_depth] = this_node

        in_file.close()


def Print(self):
        PrintTree(self.nodes)


# Classify a set of data according to the tree read in
def Classify(self, data):
        # Start at the root node
        curr_node = self.nodes[0]

        # Non-leaf nodes have position == ""
        while curr_node.position == "":
                pos = self.cols[curr_node.variable]

                # If the data received has a value for this variable of less
                # than the threshold in the tree then move to the first child
                # else move to the second
```

```
                        if data[pos] <= curr_node.value:
                                curr_node = curr_node.children[0]
                        else:
                                curr_node = curr_node.children[1]

                return curr_node.position


if __name__ == "__main__":
        tree = Tree()
        tree.InterpretText("tree_conf/ramona_tree_2")
        tree.Print
```

Appendix E

Visualiser code

E.1 Posture visualisation module

```
#!/usr/bin/python


"""
    Posture visualisation module.

    Code calling this module should call init_vis() to initialise the
    visualiser (including creating the display window), and then use
    draw_posture(name) to display specific postures. Other functions
    and variables are for internal use.
"""


import pygame, pygame.transform, pygame.image,socket
from pygame.locals import *
import math


foreground = (255,255,255)      # foreground colour (text and posture display)
background = (0,0,0)            # background colour
w = 450                         # window width
h = 400                         # window height


# Postures are defined by a set of angles, as well as two values for x and
# y starting offset (to allow centering)
postures = {
        "standing"   : [  0,  45,  90,   45,  -45, 120, 120, 250, 350],
        "sitting"    : [ 45,  40, 180,  -40,  -45, 130, 130, 250, 350],
        "walking"    : [ 45,   0, 100,  -30,  -10, 120, 120, 270, 350],
```

```
        "kneeling"    : [-80, 150, 180, -150, -160, 160, 160, 200, 350],
        "crawling"    : [-80, 130, 180, -130, -210, 180, 100, 150, 300],
        "oneside"     : [-90,  45,  90,   45,  225, 140, 140, 100, 300],
        "layingdown"  : [-90,  45,  90,   45,  225, 190, 110, 100, 300],
        "layingup"    : [-90,  45,  90,   45,  225, 170, 270, 100, 300]
}

t = None
screen = None
font = None



"""
    This class implements the "turtle" method of drawing. This is used to
    simplify the posture drawing process in the person(t, angles) function.
"""
class Turtle(object):

    def __init__(self,screen,startpos=(250,350)):
        self.pos=startpos
        self.angle=0
        self.screen=screen
        self.stack=[]
        self.pen = 1

    def forward(self,d):
        newpos=[0,0]
        newpos[0]=self.pos[0]+math.sin(math.radians(self.angle))*d
        newpos[1]=self.pos[1]+math.cos(math.radians(self.angle))*d
        if self.pen:
                pygame.draw.aaline(self.screen,foreground,self.pos,newpos)
        self.pos=newpos

    def rotate(self,angle):
        self.angle+=angle

    def push(self):
        self.stack.append((self.pos,self.angle))

    def pop(self):
        if len(self.stack)>0:
            self.pos,self.angle=self.stack[-1]
            self.stack=self.stack[:-1]

    def circle(self, radius):
        pygame.draw.circle(self.screen, foreground, self.pos, radius, 1)

    def pen_up(self):
```

```
        self.pen = 0

    def pen_down(self):
        self.pen = 1



"""
    This function draws a stick figure using the supplied set of angles.

    Inputs
        t               A object of type Turtle to use for drawing
        angles          The set of angles defining the posture to draw
"""
def person(t, angles):
    t.push()

    t.pos = (angles[7], angles[8])

    #leg1
    t.rotate(angles[0])
    t.forward(50)
    t.rotate(angles[1])
    t.forward(40)

    #leg2
    t.push()
    t.rotate(angles[2])
    t.forward(40)
    t.rotate(angles[3])
    t.forward(50)
    t.pop()

    t.rotate(-angles[0])

    #trunk
    t.rotate(angles[4])
    t.forward(70)
    t.push()
    t.rotate(angles[5])         #arm 1
    t.forward(50)
    t.pop()
    t.push()
    t.rotate(-angles[6])        #arm 2
    t.forward(50)
    t.pop()
    t.forward(30)

    #head
```

```
    t.pen_up()
    t.forward(20)
    t.pen_down()
    t.circle(20)
    t.pop()



"""
    Initialise the visualiser, including setting up the display window. This
    should be called before any other functions in this module.
"""
def init_vis():
        global t
        global screen
        global font

    # Create and set up display window
        pygame.init()
        window = pygame.display.set_mode((w,h))
        pygame.display.set_caption('Posture Visualiser')
        screen = pygame.display.get_surface()
        screen.fill(background)
        font = pygame.font.Font(None, 24)

        # Display initial message
        text = font.render("No data available", True, foreground, background)
        textRect = text.get_rect()
        textRect.centerx = screen.get_rect().centerx
        textRect.centery = 40
        screen.blit(text, textRect)

        pygame.display.update()

        t = Turtle(screen)
        t.rotate(180)


"""
    Draw a named posture. If no posture with that name is known then nothing
    will be done.

    Inputs
        name            The name of the posture to draw
"""
def draw_posture(name):
        global t
        global screen
        global font
```

```
        if postures.has_key(name):

        # Clear display and draw new posture
            screen.fill(background)
            person(t, postures[name])

            # Display name of posture
            text = font.render(name, True, foreground, background)
            textRect = text.get_rect()
            textRect.centerx = screen.get_rect().centerx
            textRect.centery = 40
            screen.blit(text, textRect)

            pygame.display.update()
```

E.2 Communication via Bluetooth

```
#!/usr/bin/python


"""
    This code simply accepts data from a Bluetooth socket and passes it back
    out via a TCP socket.

    This is used to route posture names sent by a remote node to the
    visualiser. To use this, start the visualiser, start this code, and then
    start the data gathering/classification process on the remote node.
"""


import bluetooth
import socket


ethc_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ethc_sock.connect(("127.0.0.1", 8557))

bts_sock = bluetooth.BluetoothSocket(bluetooth.L2CAP)
bts_sock.bind(("", 0x1001))
bts_sock.listen(1)

btc_sock,address = bts_sock.accept()

data = btc_sock.recv(1024)
while data != "":
        ethc_sock.send(data)
        data = btc_sock.recv(1024)
```

```
bts_sock.close()
ethc_sock.close()
```

E.3 Interface to the posture visualisation module

```
#!/usr/bin/python


"""
    This is a simple interface to the posture visualisation module. Posture
    names are accepted via a TCP socket and these are passed straight to
    the visualisation module.
"""


import socket
import posture as p
import sys
import time


p.init_vis()

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('', 8557))
sock.listen(5)

while 1:
        client_sock, client_addr = sock.accept()

        data = client_sock.recv(1024)
        while data != "":
                p.draw_posture(data)
                data = client_sock.recv(1024)
```

Cogent Computing Applied Research Centre
Coventry University
Priory Street, Coventry CV1 5FB
www.cogentcomputing.org