Learning to Coordinate Behaviours on a Four-Legged Robot

T J Taylor

MSc Information Technology: Knowledge Based Systems
Department of Artificial Intelligence
University of Edinburgh
1993

Abstract

An algorithm has been developed elsewhere which enabled a six-legged robot to learn how to coordinate its leg movements and walk with a statically stable gait. In the current project, the applicability of this algorithm to a four-legged robot was investigated. A physical machine was constructed, along with a computer simulation. There was insufficient time between the completion of the robot and producing this report to test the algorithm on the physical machine, so the results described apply to the simulation only. The basic finding was that the robot could not learn how to coordinate its legs so that it *never* fell over. Several variations and extensions of the basic algorithm were also investigated, some of which resulted in a marked improvement in performance so that, in some cases, the robot would only fall occasionally. To model the real robot more closely, some trials on the simulation included a degree of noise in the leg movements. This was found to have a (sometimes drastically) detrimental effect on the level of performance achieved. Some reasons for the failure of the algorithm to satisfactorily translate from a six-legged robot to a four-legged robot are discussed, and possible extensions, which may lead to improved performance, are suggested.

Acknowledgements

I would like to thank my supervisor, Dr John Hallam, for his guidance and encouragement throughout the course of this project.

Thanks also to Alasdair MacLean and Sandy Colquhoun for their patience and advice while I was developing the code for the microcontrollers, and for producing the printed circuit board for the robot.

I am very grateful to the members of the robotics laboratory at Forrest Hill who contributed to the construction of the robot, especially David Wyse, Hugh Cameron and Douglas Howie.

Finally, thanks to Dr Gillian Hayes for taking John's role while he was absent, and especially for providing comments on the draft report in addition to John's.

I acknowledge receipt of SERC studentship number 92419031, which provided financial support during my year of study.

Table of Contents

1. Introduction and Background to the Problem 1

1.1 What's Wrong with Wheels? 1

1.2 Previous Work with Walking Robots 2

1.3 Ghengis 3

1.4 Why Use Four Legs? 5

2. The Learning Algorithm 6

2.1 Philosophy behind the Algorithm 6

2.2 Design 7

2.3 Control Strategy 10

2.4 Global Parameters 12

2.5 Differences from Maes and Brooks' Algorithm 12

3. The Robot Design 17

3.1 Requirements 17

3.2 Hardware 18

3.3 Software 20

3.3.1 The Algorithm 20

3.3.2 The Microcontrollers 22

4. The Simulator Program 24

4.1 Why Write a Simulator? 24

4.2 Design 25

4.3 Implementation 26

4.4 The Different Versions of the Simulator 27

4.5 Using the Simulator 27

4.5.1 Getting Started 27
4.5.2 The Canvas 28
4.5.3 The Control Panel 28
4.5.4 The Behaviour Information Panel 29
4.6 Plotting Data from a Simulation Run 30
4.6.1 Feedback Percentages 31
4.6.2 Relevance and Reliability 31
4.6.3 Behaviour Statistics 32
4.6.4 Monitoring Statistics 32
4.6.5 Plotting Other Graphs 32

5. Experimental Design **34**
5.1 'Hard-Wired' Gaits 34
5.1.1 On Smudge 34
5.1.2 In Simulation 35
5.2 Testing the Algorithm in Simulation 35
5.2.1 Basic Trials 35
5.2.2 The Effect of Noise 37

6. Results **38**
6.1 'Hard-Wired' Gaits 38
6.1.1 On Smudge 38
6.1.2 In Simulation 38
6.2 Testing the Algorithm in Simulation 39
6.2.1 Basic Trials 39
6.2.2 Preprogramming the Behaviours 47
6.2.3 Other Extensions 52
6.2.4 The Effect of Noise 60

7. Summary and Discussion **65**
7.1 The Original Goals of the Learning Algorithm 65
7.2 The Results Obtained 66
7.3 Comparison of Preconditions Learned by Best Trials 68

7.4 Some Reasons for Failure to Achieve Perfect Performance 69
7.5 Further Experiments with the Algorithm 70
7.6 General Comments about the Algorithm 71
7.7 Possible Extensions 72
7.8 Relating Results to Other Research 73

Appendices

A. Final Specification of Smudge 77

B. Program Code for the Learning Algorithm 78

C. Program Code for the Simulator 91

D. Program Code for the Microcontrollers 98

List of Figures

2–1 Conceptualised Control Flow 11

2–2 A Quadruped Gait which obeys the Horizontal Balance Constraint . 14

3–1 Illustration of Smudge's Design 19

3–2 Schematic Illustration of Microcontroller and Force Sensing Circuits 21

3–3 Photograph of the Finished Robot 22

4–1 Example Screen from the Simulator 25

6–1 Graph of Feedback *v.* Time for Trial A1 41

6–2 Graph of Feedback *v.* Time for Trial B1 42

6–3 Graph of Feedback *v.* Time for Trial C2S 43

6–4 Graph of Statistics *v.* Time for a Behaviour from Trial C2S 44

6–5 Graph of Statistics *v.* Time for a Behaviour from Trial C12 45

6–6 Graph of Feedback *v.* Time for Trial C12 46

6–7 Situations Where Smudge was Liable to get Stuck During Learning 48

6–8 Graph of Feedback *v.* Time for Trial PA6 50

6–9 Graph of Feedback *v.* Time for Trial PB1S 51

6–10 Graph of Feedback *v.* Time for Trial RA2 56

6–11 Graph of Feedback *v.* Time for Trial RB2 57

6–12 Graph of Feedback *v.* Time for Trial CC1 60

6–13 Noise Distributions Used to Test the Robustness of Learning 61

List of Tables

2–1 Positive Feedback Statistics for each Behaviour 7

2–2 Monitoring Statistics (Positive Feedback) 9

4–1 Summary of the Different Versions of the Simulator 27

4–2 Key to Behaviour Numbers used in the Simulator 30

4–3 Labels used to refer to Data from within MATLAB 33

5–1 Summary of Default Parameter Values Used in the Basic Trials . . 36

5–2 Summary of Basic Trials Performed on Each Version of the Algorithm 37

6–1 Preconditions learned by Version C, trial 2S 44

6–2 Summary of Additional Trials Performed for Version C 45

6–3 Preconditions learned by Version C, trial 12 46

6–4 Preconditions Preprogrammed into Version A, trial PA1S 47

6–5 Preconditions Preprogrammed into Version C, trial PC1 52

6–6 Default Parameter Values Used with Extensions to Algorithm . . . 52

6–7 Preconditions learned by Version A, trial RA2 55

6–8 Preconditions learned by Version B, trial RB2 57

6–9 Preconditions learned by Version C, trial CC1 59

6–10 Preconditions learned by Version C, trial NCC1 63

Chapter 1

Introduction and Background to the Problem

1.1 What's Wrong with Wheels?

The vast majority of work with mobile robots conducted to date has concentrated on machines equipped with wheels. This is not surprising, as wheels and driving motors are readily available to suit almost any size of robot, and wheeled locomotion is relatively easy to control, and entirely adequate for transporting a robot around a laboratory floor.

However, if we want our robots to emerge from their sanitized play-pens and step (or wheel) out into the 'real world', then wheeled locomotion becomes more problematic. In situations involving rough terrain, patches of ground which are unable to support the weight of the vehicle, the need to negotiate stairs and so on, wheeled systems come a poor second to those with legs, i.e. legged animals. (Another advantage of legged systems over those with wheels, often quoted in texts on legged machines (e.g. [Raibert 86]), is that legs provide a form of active suspension, or decoupling of the motion of the legs and body. However, there is no reason why wheeled machines cannot be fitted with active suspension. In fact, over the past few years, much work has been devoted to this topic in the motor car industry.)

1.2 Previous Work with Walking Robots

As early as the 1890s, several designs were proposed for walking machines controlled by mechanical linkages, but there is no evidence that any of these were actually constructed. By the 1950s, a more systematic approach towards the study of legged machines was emerging, leading to the creation of several dozen experimental systems by the 1980s. These systems varied in the number of legs used, but most had one, two, four or six legs. For a review of these machines, see [Song & Waldron 89] or [Todd 85].

In the early 1980s, some fairly sophisticated designs were produced. [Hirose 84] describes work on a four-legged machine which walked with a dynamically stable[1] gait. The robot could negotiate obstacles while maintaining a horizontal body orientation, and was also much more energy efficient than previous walking machines. Raibert and colleagues devoted much research to the area of hopping machines, and successfully constructed examples with one, two and four legs (see [Raibert 86]).

While many machines exhibited competent performance in the laboratory, none can really be said to have been of practical use or to possess all of the abilities that legged animals display.

One reason for this lack of performance is the relatively low level of development that has gone into the mechanical design of the legs—the designs used fall a long way short of achieving the flexibility and robustness of animals' legs.

Another major reason is the complexity of the control systems that these machines were utilising to move their legs. The machines built up to this point were all designed around the classical 'sense-think-act' cycle of control, in other words,

---

[1] A 'dynamically stable' gait is one in which the vehicle is not stable at every point in its cycle, but the body may tip and accelerate for short periods of time. However, over time, tipping motions in one direction are compensated for by tipping in the opposite direction, and an effective base of support is thus provided. All forms of running are examples of dynamically stable gaits.

their architectures were decomposed into functional modules, such as perception, modelling and planning. Such control systems require an explicit internal representation of the mathematical equations relevant for legged locomotion and balance. The more robust the required performance of the robot, the more complex and detailed the equations.

Since the mid-1980s, a paradigm has emerged to rival this classical approach to constructing robot architectures ([Brooks 91b] gives a comprehensive introduction). Pioneered by Rodney Brooks at MIT, this new approach involves breaking down the problem of building a robot into behavioural, rather than functional, modules. The task of designing a robot to achieve a particular task then proceeds by first equipping the machine with a small number of very simple behaviours, and gradually building up new, more complex behaviours on top of the simpler ones until the required level of behavioural competence is reached. Brooks called this approach the subsumption approach, as lower level behaviours are subsumed by higher level ones. Subsumption-based designs lack the central processing pathways of their classical counterparts, tending to evolve with a large number of fairly low-level, localised connections between sensors and actuators.

In 1989, Brooks published the results of applying this new approach to robot design to a robot with six legs, called Ghengis ([Brooks 89]).

1.3 Ghengis

The original Ghengis was a small (25 x 35cm) robot with six legs, each with two degrees of freedom (up/down and forward/backward). Each leg was controllable by two orthogonally mounted model-airplane position-controllable servo motors. The control architecture which originally drove Ghengis, as described in [Brooks 89], allowed the robot to acquire a robust walking behaviour through a process of incrementally expanding its repertoire of behavioural competences (i.e. a subsumption architecture).

The completed system did indeed show a robust walking behaviour, being able to stabilize pitch and roll, walk over obstacles and steer towards infrared sources. However, such a design procedure is, to some extent, unsatisfactory, relying heavily

on the designer having a good insight or 'feel' for what competences to give the robot, and at which level.

In later experiments using essentially the same hardware, Pattie Maes and Brooks ([Maes & Brooks 90]) took a different approach to enabling Ghengis to walk. They designed an algorithm which allowed the robot to learn on the basis of positive and negative feedback when to activate any of its repertoire of behaviours.

For Ghengis, positive feedback was provided by a trailing wheel which detected forward movement, and negative feedback by two touch sensors on the robot's belly. In the simplest case, Ghengis was equipped with just six behaviours which it had to learn to coordinate; a 'move forward' behaviour for each of the six legs (i.e. move leg up, forwards then down), coupled with a global horizontal[2] balance reflex (i.e. if one leg is moved forward, then all the others are moved back a little). The task of the learning algorithm was to decide under which conditions each behaviour was relevant and reliable to produce a stable walking movement.

When implemented, Ghengis was able to learn an alternating tripod gait[3] in between 10 minutes and $1\frac{3}{4}$ minutes, depending on the particular version of the learning algorithm. However, the walking competence learned was not as robust as that produced using the 'hard-wired' subsumption architecture. Ghengis could only walk in a straight line on a smooth surface; it could neither turn nor negotiate obstacles.

Nevertheless, the basic idea of using a learning procedure, rather than a hard-coded architecture, to produce the desired walking behaviour still seemed like a more promising approach in the long run, so I decided to further investigate Maes and Brooks' algorithm in this project.

---

[2]'Horizontal' refers to the angle by which each leg is displaced in the horizontal plane (forward/backward), as opposed to its 'vertical' angle (up/down).

[3]An alternating tripod gait for a hexapod is one where the front and rear legs on one side and the middle leg of the other side move in synchrony, in antiphase with the other group of three legs. There are always at least three legs on the ground, providing a tripod for support. Thus, the robot is stable at all times — it is a 'statically stable' gait.

1.4 Why Use Four Legs?

There are several ways in which the learning algorithm used with Ghengis could have been further explored and expanded. For example, one could have tried to have incorporated some of the other competences that the 'subsumption-based Ghengis' possessed, such as being able to turn corners or climb over obstacles. Rather than this, however, I decided to explore the applicability of the algorithm to a robot with a different number of legs.

Specifically, I chose to investigate whether the algorithm could be applied to a four-legged robot. I chose fewer than six legs because it is theoretically harder to control a robot with fewer legs in such a manner that it is stable at every point in its gait (that is, it is harder to produce a 'statically stable' gait). As explained in [Todd 85] (p. 61), "The alternating tripod gait [*that Maes and Brooks observed emerging on Ghengis*] ... is particularly important for walking robots, and accounts for the prevalence of six-legged machines. Six is the smallest number which always provides a tripod of support even when half the legs are raised. It therefore allows reasonably fast walking, while maintaining static stability at all times."

Note that the development of a quadruped with a statically stable gait is of little interest *per se*, as such a gait is only a very special case of all possible gaits available to a quadruped, and would not allow it to walk particularly fast, let alone run. However, from the perspective of investigating the applicability of the algorithm to a quadruped, such a gait is the easiest it could be expected to learn. Only when this task has been successfully achieved will it be appropriate to look at more complicated gaits.

This chapter has given a brief introduction to my project and some background to the problem that I have tackled. In the next chapter I describe in detail the learning algorithm used.

Chapter 2

The Learning Algorithm

2.1 Philosophy behind the Algorithm

As discussed in the previous chapter, Maes and Brooks wanted to explore ways of developing robot control architectures which did not involve manual design. Even simple systems can require a fairly complicated switching circuitry among selectable behaviours, and it was envisaged that the creation of such circuitry for more complex robotic systems would be too difficult for a human designer.

Maes and Brooks also wanted to produce a system which could continuously monitor its performance and adapt its behaviour should the environmental conditions in which it found itself change. For example, a robot which had learned how to walk on smooth ground should not be stumped when it comes across its first obstacle. Rather, it should be able to learn how to get around this obstacle, if necessary by altering some of the knowledge that it has already acquired. This is in contrast with the types of control system which had been built up to this point, which were hard-wired into the robot by the designer when finally implemented, and were therefore not actively adaptable.

As stated in [Maes & Brooks 90] (p. 796), "In accordance with the philosophy of behaviour-based robots, the learning algorithm is completely distributed. There is no central learning component, but instead each behaviour tries to learn when it should become active." In fact, this is not true for all of the behaviours used; see the comments about the 'horizontal balance behaviour' in the following sections. In

	active	not active
positive feedback	j	k
no positive feedback	l	m

Table 2–1: Positive Feedback Statistics for each Behaviour

the next three sections I will describe the original version of the learning algorithm as used by Maes and Brooks. To be implemented on a four-legged robot, it was necessary to make a number of small changes to the algorithm. These changes are listed in the final section of the chapter.

2.2 Design

The robot is equipped with a set of behavioural primitives, and the task of the algorithm is to determine, on the basis of global positive and negative feedback signals received during learning, under what conditions each behaviour should be activated. The conditions that the algorithm monitors in order to achieve this task are related to the state of each leg. For Ghengis, there were just six conditions involved; namely, whether each of the six legs was up or down.

In the first experiment described in [Maes & Brooks 90], there were only six behaviours to be coordinated; a swing-leg-forward behaviour (i.e. move a leg up, forward, then down again) for each of the six legs. There was also a global 'horizontal balance' reflex hard-wired into the robot, which sums the horizontal angles of the legs and sends a correction to all of the legs so as to reduce that sum to zero. This has the effect that, if one leg is moved forward, then all the other legs are moved back a little. This action was hard-wired into Ghengis rather than having to be learned. For this reason, I will refer to it as the horizontal balance *reflex* rather than behaviour.

Each learnable behaviour has two sets of statistics associated with it, which record how frequently positive and negative feedback signals are received when the behaviour is active and inactive. Table 2–1 shows the statistics for positive feedback — a similar set is maintained for negative feedback.

Each cell in the table of statistics is a count of how many times that particular combination of circumstances has been observed. Each cell is given an initial value, N, at the beginning of a run, and is incremented whenever that particular situation occurs. At each time step, whether the cell has been incremented or not, its value is decayed by $\frac{N}{N+1}$ so that its maximum value is always N. In this way, the effect of more recent experiences dominates, while statistics from less recent experiences die away. The algorithm can therefore cope with changes in its experienced environment; anything that is learned can later become unlearned if no longer appropriate for optimising the expected feedback signals.

Two measures of each behaviour's current performance are calculated from these statistics throughout the execution of the algorithm. One measure is the *relevance* of the behaviour, which is calculated as follows. The Pearson product-moment correlation coefficient between positive feedback and the activity of the behaviour is calculated with the formula

$$corr(P,A) = \frac{j*m - l*k}{\sqrt{(m+l)*(m+k)*(j+k)*(j+l)}}.$$

This gives a number between +1 and -1, indicating how strongly the action of the behaviour is correlated to receiving direct positive feedback. The corresponding coefficient for negative feedback, *corr(N,A)*, is calculated in the same way. The relevance of the behaviour is then defined as

$$relevance = corr(P,A) - corr(N,A)$$

which is therefore a number in the range +2 to −2. The control strategy of the algorithm is such that the more relevant a behaviour is, the more likely it is to be activated.

The other measure calculated from the statistics is the *reliability* of each behaviour. This gives an indication of how consistently positive (or negative) feedback is received when the behaviour is active. Reliability is defined as

$$reliability = min(max(\frac{j_p}{j_p+l_p}, \frac{l_p}{j_p+l_p}), max(\frac{j_n}{j_n+l_n}, \frac{l_n}{j_n+l_n}))$$

where the p and n subscripts refer to the statistics relating to positive and negative feedback respectively. Reliability values range from 0 to +1. A highly relevant behaviour may still have low reliability if neither positive nor negative feedback is

	condition on	condition off
positive feedback	n	o
no positive feedback	p	q

Table 2–2: Monitoring Statistics (Positive Feedback)

received on the majority of occasions when it is active. The algorithm therefore uses the reliability of each behaviour to decide whether the behaviour should try to improve its performance.

Each behaviour which has not reached a certain threshold of reliability will try to improve its performance by changing the list of perceptual preconditions which need to be fulfilled in order for that behaviour to be activated. This is achieved by monitoring a new perceptual condition while the behaviour is active to see whether it is correlated to positive or negative feedback. If a strong correlation is observed, then the condition is adopted on the precondition list.

When a behaviour starts monitoring a condition, it initialises two sets of statistics, one relating to positive feedback (see Table 2–2), the other to negative feedback.

From these statistics, the correlation between a condition being on and positive feedback being received (while the behaviour monitoring the condition is active) may be calculated:

$$corr(P, on) = \frac{n * q - p * o}{\sqrt{(q + p) * (q + o) * (n + o) * (n + p)}}.$$

The correlation between the condition being on and negative feedback, corr(P, off), is calculated similarly.

If, after some time monitoring the condition, a strong positive correlation is observed between that condition being on and receiving positive feedback (or negative feedback), then a new precondition stipulating that this condition must be on (or, for negative feedback, that it must be off), is appended to the existing precondition list for the behaviour. Similarly, if a strong negative correlation with positive (or negative) feedback is observed, then a new precondition stipulating that the condition must be off (or, for negative feedback, that it must be on), is appended to the existing precondition list. Having adopted a new precondition,

the behaviour may still not have reached the criterion reliability target — if this is the case, it will continue monitoring further conditions.

If no correlation is observed after a certain duration of monitoring, then the behaviour will start monitoring another condition. The list of conditions to be monitored is circular, so that conditions already on the precondition list may be re-evaluated, and, if necessary, discarded.

2.3 Control Strategy

The algorithm places the behaviours into groups which control the same legs. At each time step, it is ascertained which behaviours in each group are eligible to become active, i.e. those which are not already active but have all preconditions fulfilled. One or zero of these eligible behaviours from each group are then chosen, probabilistically, to be activated. The likelihood that a behaviour will be chosen depends, in order of importance, upon its

1. relevance relative to other selectable behaviours

2. reliability

3. "interestingness". This relates to the case where the behaviour is monitoring a condition. If the current situation (the condition being on or off) has been experienced less than other situations, then it is deemed to be more interesting.

The control loop of the algorithm is illustrated in Figure 2–1. This process continues indefinitely, and behaviours will continue to monitor new conditions if they are not reliable enough. If the robot falls over at any stage, i.e. negative feedback is received, then all the legs are returned to their initial position.

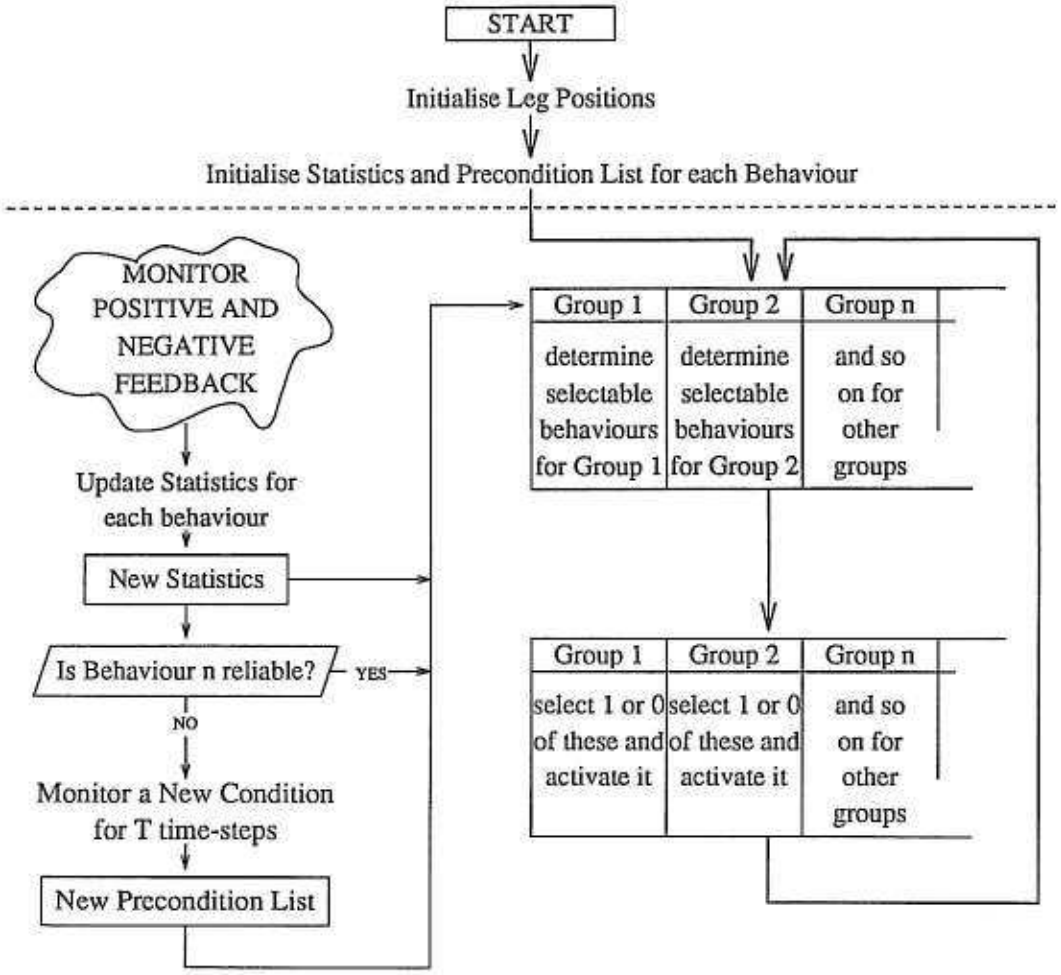


Figure 2–1: Conceptualised Control Flow

2.4 Global Parameters

There are a number of global parameters associated with the learning algorithm:

- how strongly a condition must be correlated to feedback before it is adopted as a new precondition for a monitoring behaviour
- the duration for which a condition is monitored before it is dropped
- how reliable a behaviour should try to become
- how adaptive a behaviour is, i.e. the relative importance of new data versus data from past experiences

These parameters must be fine-tuned for a particular robot and environment in order to optimise the performance of the algorithm.

2.5 Differences from Maes and Brooks' Algorithm

The algorithm as described in the preceding sections is the version used by Maes and Brooks with their Ghengis robot. In order for it to work on a robot with only four legs, it was necessary to make a number of slight alterations. These are listed below.

- *The condition vector.* The algorithm used with Ghengis only dealt with six conditions — whether each of the six legs was raised or not. This was sufficient for a six-legged machine walking with a tripod gait, because the coordination between the two groups of three legs does not particularly require knowledge of whether any of the legs are forwards or backwards — it is a sufficient precondition for one group of three to be raised that the three legs in the other group are all on the ground.

However, for a quadruped, an analysis of the types of statically stable gaits possible reveals that a much tighter coordination is required between the legs. Specifically, if each leg is to learn when it should raise itself and swing forward, it appears essential that it should know not only whether the other legs are up or down, but also whether they are forwards or backwards. With this in mind, I added eight new conditions to the existing four leg up/down conditions; a leg forward and a leg back condition for each of the four legs. The angles by which a leg must be moved forwards or backwards to trigger these two new conditions were specified by two new parameters.

- *Global Horizontal Balance.* As already quoted, [Maes & Brooks 90] (p. 796) claim that "in accordance with the philosophy of behaviour-based robots, the learning algorithm is completely distributed. There is no central learning component, but instead each behaviour tries to learn when it should become active." It therefore seems strange that they should include this 'hard-wired' 'global' horizontal balance reflex in their system.

  On top of this, it is less intuitive that such a behaviour is appropriate for a quadruped than it is for a hexapod—during an alternating tripod gait on a hexapod, three legs move forward as the other three move back, so that the total horizontal angle will always be zero—there is no such obvious symmetry in a statically stable quadruped gait.

  In fact, an analysis of quadruped gaits shows that there *is* a statically stable arrangement where the total horizontal angle is always zero, as illustrated in Figure 2–2. This gait is a special case, however, requiring one leg to be raised as soon as the previous one has been lowered, so that all four legs are on the ground for only a brief portion of the gait cycle.

  It seemed unwise to restrict the system in this manner, so, as well as experimenting with the horizontal balance method, I also decided to look into other ways of moving the legs backwards. Two alternatives were tried:

  - *A Global Move-Leg-Back Reflex.* Like the horizontal balance reflex, this acted globally, and was hard-wired into the algorithm, i.e. it was active all the time rather than having to be learned. Unlike the horizontal balance reflex, there was no summing of horizontal angles. Rather,

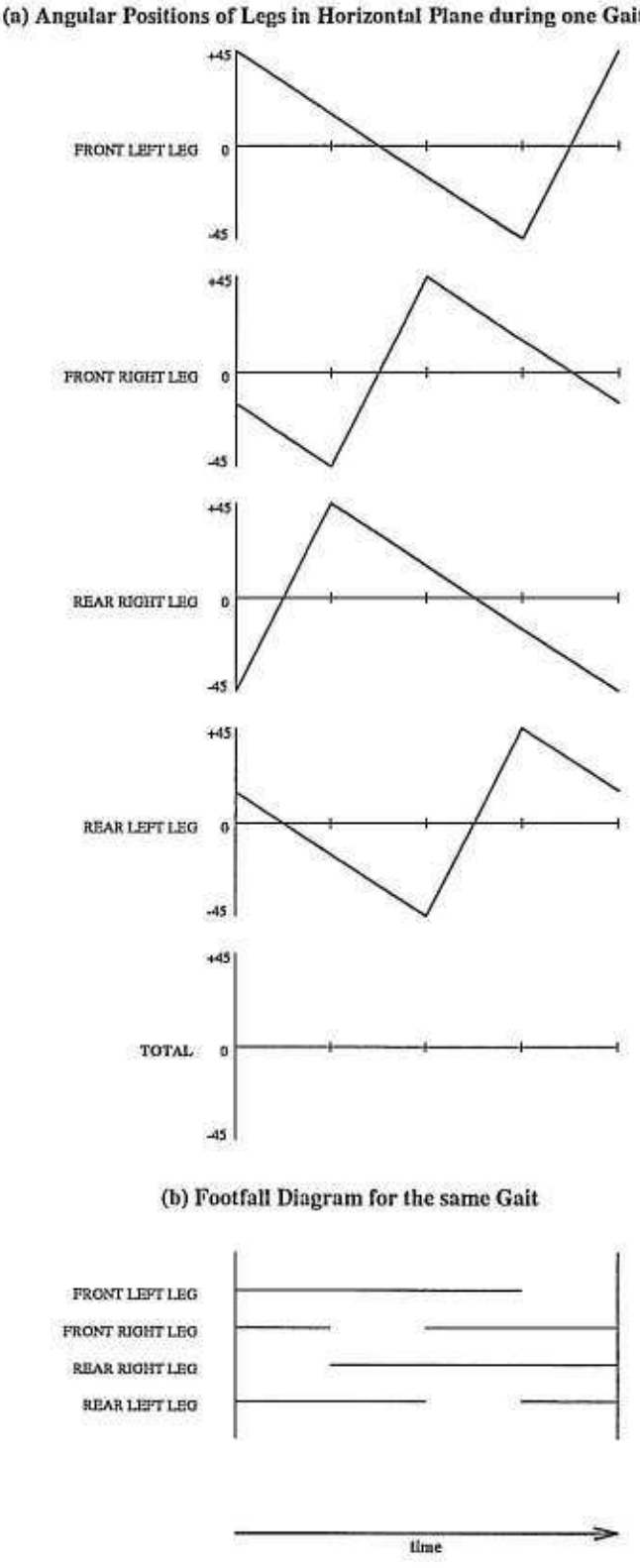


Figure 2–2: A Quadruped Gait which obeys the Horizontal Balance Constraint

each leg which was on the ground was just moved back by some fixed angle, regardless of the positions of the other legs.

- *Learnable Move-Leg-Back Behaviours.* In keeping with the idea of having a truly localised learning algorithm, this method required each individual leg to learn when to move backwards in exactly the same way that it had to learn when to swing forwards. Thus, in this version of the algorithm there were eight learnable behaviours, and no hard-wired reflexes.

- *Definition of 'Interestingness'.* The notion of interestingness was described in [Maes & Brooks 90], but was not formally defined. In my work I adopted the following definition (refer to Table 2–2). The interestingness relating to statistics for positive feedback is defined

$$int_p = \begin{cases} \frac{o+q}{n+o+p+q} & \text{if monitored condition on and } n+p < o+q \\ \frac{n+p}{n+o+p+q} & \text{if monitored condition off and } o+q < n+p \\ 0 & \text{otherwise.} \end{cases}$$

The first case is for situations where the condition is on, but the total number of times that it has previously been on, $(n+p)$, is smaller than the total number of times when it has previously been off, $(o+q)$. Similarly, the second case is for situations where the condition is off, but the total number of times that it has previously been off, $(o+q)$, is smaller than the number of times that it has previously been on, $(n+p)$. All other situations are deemed to have an interestingness value of zero.

The interestingness relating to statistics for negative feedback, int_n, is defined similarly.

The maximum of these two values is taken as the overall interestingness of the situation:

$$interestingness = Max(int_p, int_n)$$

- *Balance between Relevance, Reliability and Interestingness when choosing a which Behaviours to Activate.* Again, this balance was not formally defined

in [Maes & Brooks 90], although it was stated that when choosing a behaviour for a group, the relative relevance of the selectable behaviours had a greater influence than their reliability, which in turn had a greater influence than their interestingness.

In my work, I scored each behaviour according to

$$score = 2 * \{(2 + relevance) + reliability\} + interestingness$$

which was therefore a number between 0 and +11.

To decide which, if any, of n selectable behaviours to pick, a roulette wheel selection method was used, with a wheel size of $11n$[1]. If the wheel stopped in the region $\sum_{i=1}^{n} score_i$ to $11n$, then no behaviour was activated.

The code for the algorithm as used with the simulator is listed in Appendix B.

[1]That is, an imaginary roulette wheel is created, with total area $11n$ (which is the maximum possible total score of n behaviours). Each behaviour is allocated a sector of the wheel of area equal to its score. Thus, the total area of the wheel used for these allocations is $\sum_{i=1}^{n} score_i$.

Chapter 3

The Robot Design

The previous chapter described the learning algorithm used for this study. The idea was that this would be tested using both a simulation of a four-legged robot, and a physical machine. This chapter describes the considerations that went into the design of the latter, and details the hardware and software design.

3.1 Requirements

The robot, called Smudge, was based upon the design of Ghengis. In order to be suitably controlled by the learning algorithm, the following requirements had to be met:

- Each of the four legs must be independently position-controllable in the up/down direction and in the forwards/backwards direction.
- There must be some mechanism for providing a positive feedback signal when the robot is moving forwards.
- There must be some mechanism for providing a negative feedback signal when the robot falls over.

There were also some more practical considerations, such as:

- A low weight, to keep both the torque required from the servos, and the driving power required, to a reasonable level.

- Rigidity. The legs had to be strong enough to support the robot's body without bending.

- Ease of construction. The finished robot was required in a fairly short space of time.

3.2 Hardware

Much of Smudge's design (e.g. the configuration of the servos to drive the legs in two orthogonal directions, the use of a trailing wheel etc.) was adopted directly from Ghengis. The detail of the mechanism for providing negative feedback was slightly modified, as described below. An additional feature was the inclusion of force-sensing circuits — one for each servo — to provide information about how much current each servo is draining. Although this is not required for the present algorithm, it was included for future expansion of the system (see the Software section below). I developed the physical design of Smudge to a level shown in Figure 3–1. The staff of the department's mechanical workshop were largely responsible for more detailed design considerations and for the construction of the robot.

The finished robot weighed approximately 500 grammes, so there was no problem in finding servos which could exert sufficient torque to support the body. However, some care was needed in the decision of which servos to use, as the models available had a variety of weights as well as torques. Thus the choice of servo affected the overall weight of the robot, which affected the torque required. A number of combinations of servo were considered on paper. The two models chosen afforded the greatest excess of torque over what was calculated to be necessary so that two legs could support the total weight of the robot.

The values of the resistors in the force-sensing circuits were chosen so that, when the output of the operational amplifier was fed back into the microcontroller, the analogue-to-digital converter covered a reasonable range of input values.

Positive feedback signals were provided by a Hall Effect sensor attached to a trailing wheel which was dragged along the ground as Smudge advanced. Negative

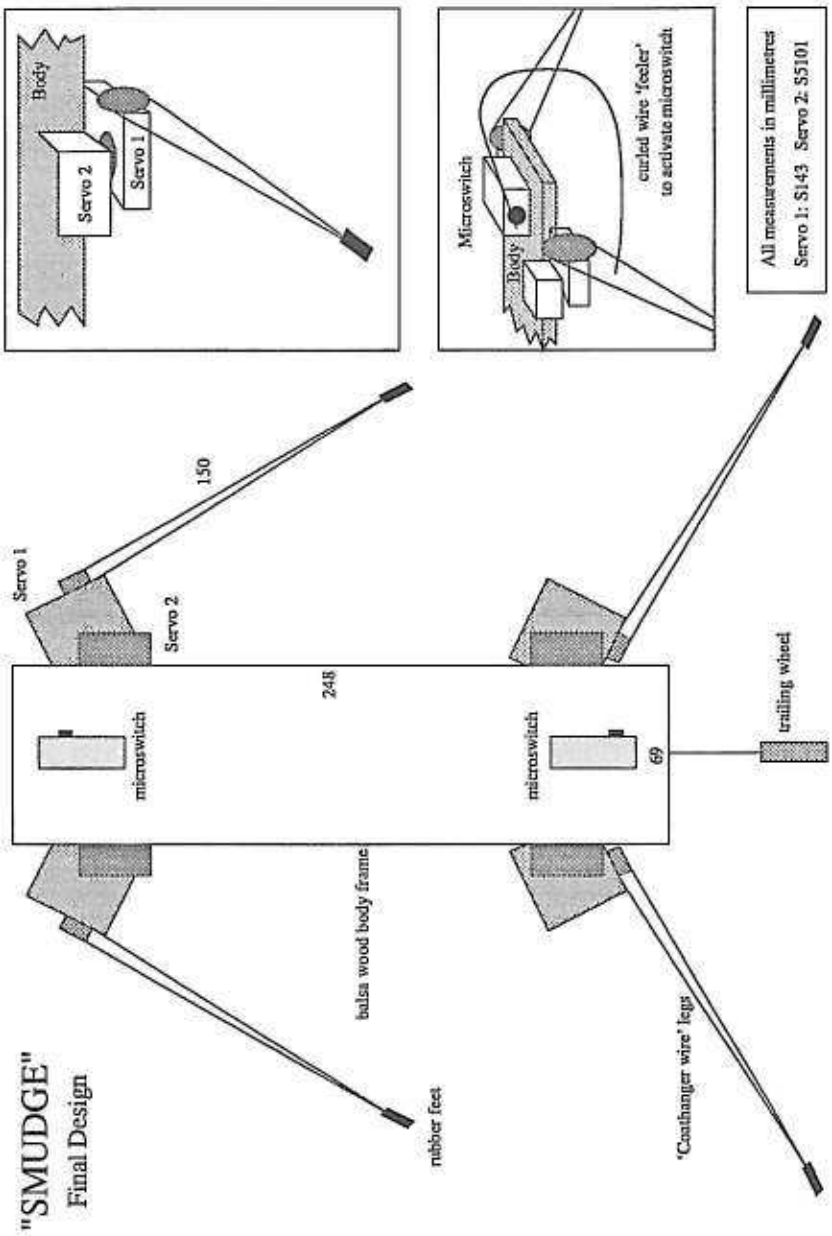


Figure 3–1: Illustration of Smudge's Design

feedback signals were provided by two microswitches, mounted front and rear, which were triggered by curled wire 'feelers' if Smudge fell on its belly (see inset in Figure 3–1 for details). The feelers were to prevent damage to the microswitches when Smudge fell over, by enabling them to be mounted on top of the body rather than on the underside.

The eight servos were controlled in two groups of four by a couple of PIC 16C71 8-bit microcontrollers. The microswitches and Hall Effect sensor were also connected to the microcontrollers. The final circuit is illustrated in Figure 3–2. There is one free pin on Port B of one of the PICs, which may be used for input or output purposes in future versions of the system.

The PICs had three digital channels of communication to the 'Brain Brick' board, which was designed in the AI department in Edinburgh and is usually used to control Lego robots. This board is programmable in C, so that most of the algorithm code developed for the simulator (described in Chapter 4) could be directly transferred onto the real robot.

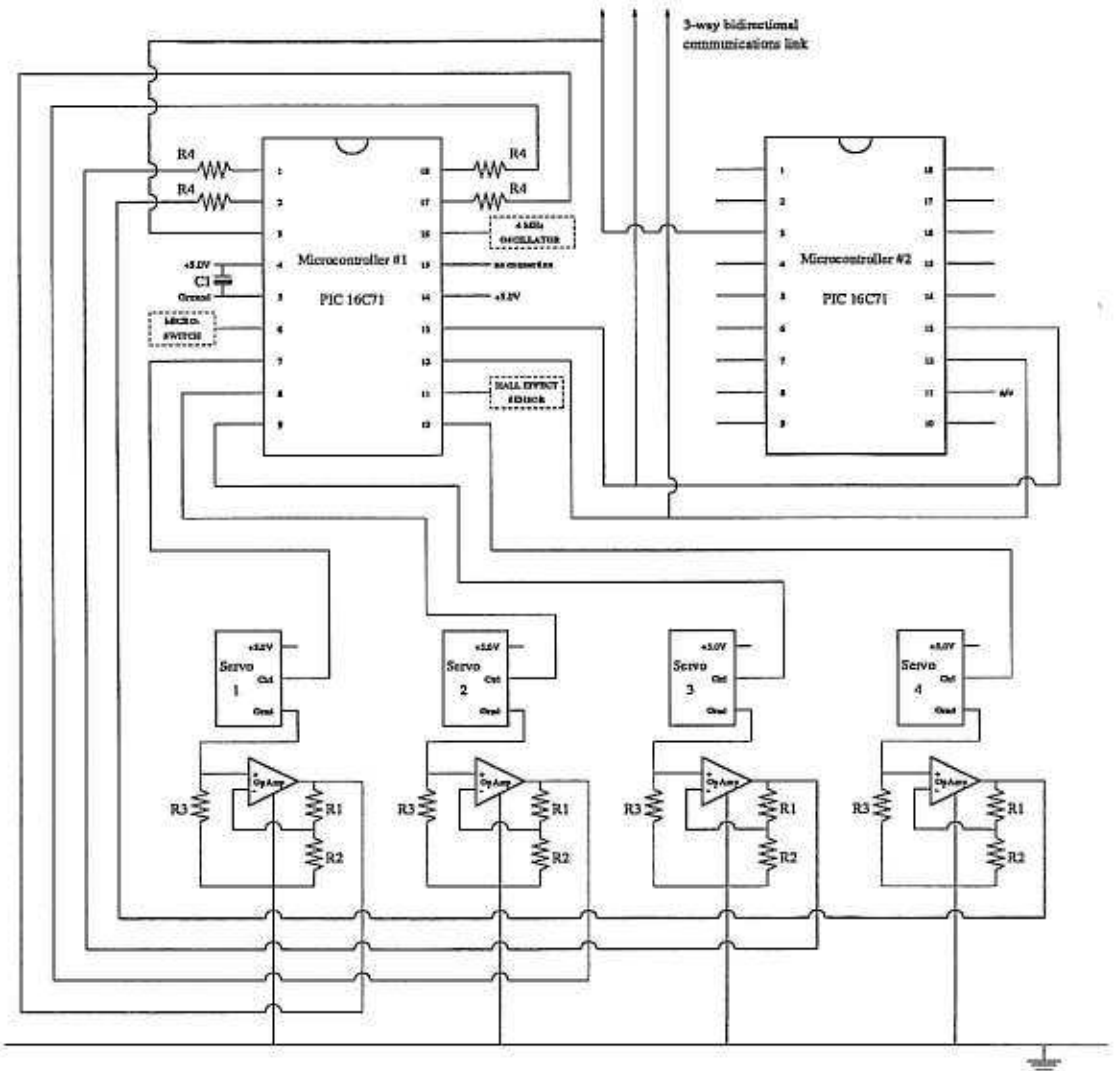
The board containing the microcontrollers was mounted on Smudge's body. An umbilical cable connected this board to the Brain Brick (which was not mounted on Smudge), and also supplied power to the robot. A photograph of the finished robot is shown in Figure 3–3. Note that the microcontroller board is mounted on Smudge's underside in the photograph. This was just a temporary arrangement — it will normally sit on top. Also, the wire feelers attached to the microswitches had yet to be curled under the body.

3.3 Software

3.3.1 The Algorithm

The software to implement the learning algorithm was written in C, which could be used for both the simulator and the real robot. The code for the algorithm, as used with the simulator, is shown in Appendix B.

As Smudge was only completed in late August, there was insufficient time to implement the algorithm on it before the completion of this report. However, as



All Microcontroller #2 connections (including connections to Servos 5, 6, 7 & 8) are identical to those for Microcontroller #1 unless otherwise indicated (note that there is no connection to pin 11 of Microcontroller #2).

Servo 1,2,5,6:	FP-S143
Servo 3,4,7,8:	S5101
R1:	1 MΩ
R2:	1 KΩ
R3:	100 m Ω
R4:	10 KΩ
C1:	100 uF

Figure 3–2: Schematic Illustration of Microcontroller and Force Sensing Circuits

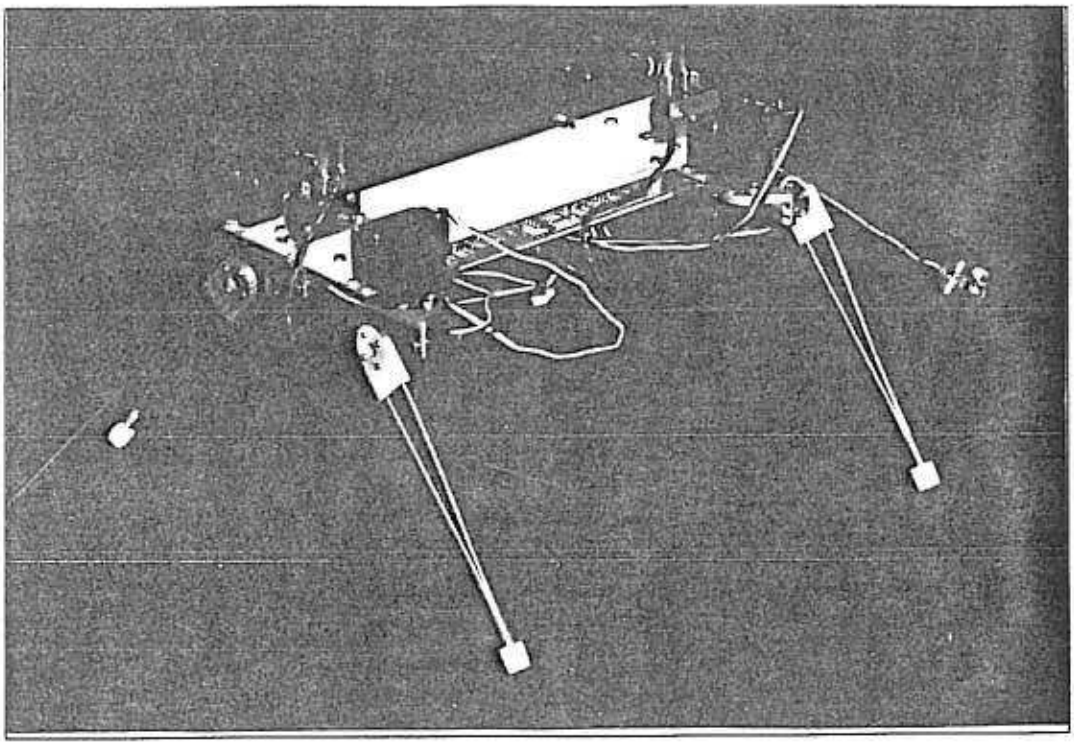


Figure 3–3: Photograph of the Finished Robot

already mentioned, the Brain Brick is programmable in C, so that the majority of the algorithm code for the simulator can be transported directly. The only changes required are concerned with input of data from sensors, and output to the servos.

3.3.2 The Microcontrollers

The PIC 16C71 microcontrollers are programmable in a RISC-like assembler language. At the time of writing, they had just been programmed to control the servos directly, without any connection to the learning algorithm (see Appendix D for code).

Code for reading values from the force-sensing circuits attached to the servos, using the analogue-to-digital conversion facilities of the PICs, has also been tested (see Appendix D). There is no use for this force-sensing information in the present algorithm, but the facility was included should it be necessary, in future work with Smudge, to know when the servos were draining unusually high currents (indicating that their movement was being impeded, perhaps by an obstacle).

In order to be linked up to the learning algorithm, additional code must be written to monitor the microswitch inputs (via pin RB0 on each of the PICs) and the Hall effect sensor (pin RB5 on the rear PIC). This information may be transmitted to the Brain Brick via the three-line communications link.

The communications protocol employed must allow for bidirectional transfer, as the Brain Brick must also send the microcontrollers positional information for each of the servos. The servos require a signal to indicate their desired position every 20ms, whether or not this position has changed. It is envisaged that the algorithm code will communicate to the microcontrollers each time a change in position for a servo is required, and the program on the microcontrollers will maintain the appropriate signal to the servo until it receives the next change of position command for that servo from the algorithm.

It will be a fairly simple task to modify the existing code shown in Appendix D to meet these requirements, and should only require a few days' work.

Chapter 4

The Simulator Program

4.1 Why Write a Simulator?

As no previous practical work has been done in the department concerning robots with legs, a four-legged machine had to be designed and built as part of the project. Because such tasks often take longer to complete than originally planned, and considering the fact that a simple simulation program could be written in a matter of weeks, it was decided that the early work with the algorithm should be done using a program to simulate the robot.

The idea was that work would continue with the simulator until the physical robot was complete, and attention could then be switched to this. As is turned out, the robot was not completed until late August, so that all of the work with the learning algorithm was done using the simulator.

However, an attempt will be made after the completion of this report to program a 'hard-wired' statically stable ripple gait into the robot, in order to demonstrate that the goal of the learning task is achievable on the physical machine.

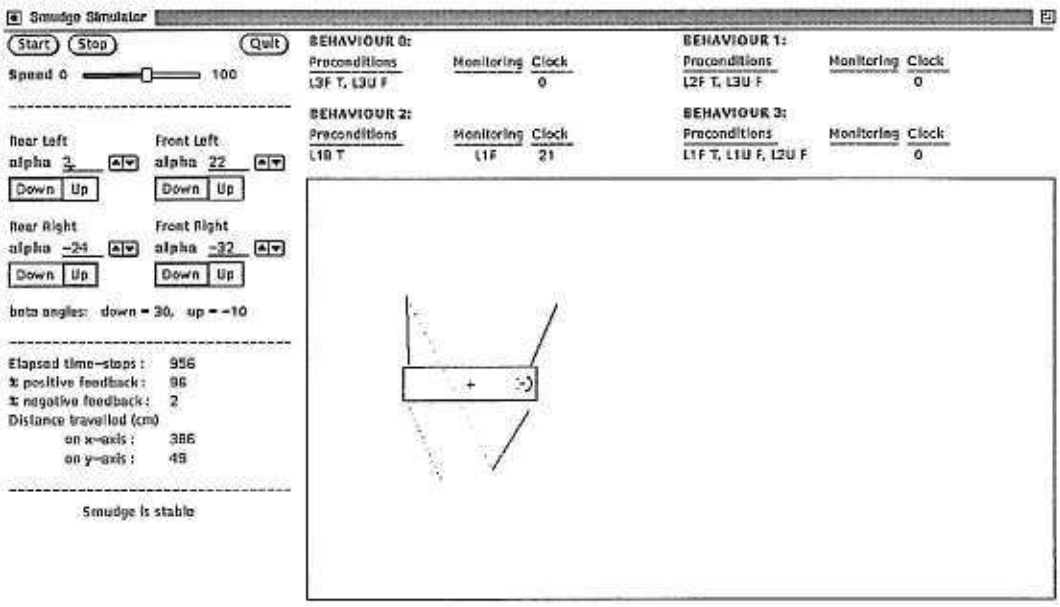


Figure 4–1: Example Screen from the Simulator

4.2 Design

The simulator code was kept as separate as possible from the code for the learning algorithm. This was so that the latter could be transferred onto the physical robot with minimal alterations.

The only interaction between the two programs occurs in the function 'animate' in the simulator code (see Section 4.3 below, and Appendix C). This contains a call to the 'next_time_step' function from the algorithm code (Appendix B), which runs the algorithm for one more time step and returns the robot's new position, together with information regarding its margin of stability.

A sample screen-shot from the simulation is shown in Figure 4–1. Smudge is represented by a two dimensional line drawing of the view looking down onto the robot. Raised legs are represented by dotted lines.

The simulation contains an accurate model of the dimensions and leg–servo offsets for the robot (this is why, on the simulation, the legs do not always seem to originate from the body — the servos are not drawn on the screen). From this information, for any given set of angles for the servos, the program can calculate

whether Smudge is stable. The calculation assumes that the robot would be stable if the projection of its centre of mass (which was assumed to be the centre of the body, represented by a small cross on the simulator) onto the plane of the ground lies within the polygon of support provided by those legs which are on the ground. To enhance the visualisation of the margin of stability afforded, the boundary of this polygon of support which lies closest to the centre of mass is represented on the screen with a dotted line (see Figure 4–1).

Further details about the display are given in Section 4.5.

4.3 Implementation

The simulator was written in C, with a user interface programmed with XView. The code for the simulator is listed in Appendix C. At regular intervals during the execution of the algorithm, summary data relating to the state of the system at that point are output in ASCII flat-file format to a file called 'data.out'. The file is closed when the user quits the simulator. This data file is suitable for use with MATLAB[1], and a suite of M-files have been written to speed the plotting of graphs of the most important data. These files may be found in the directory simulator/data/[2] (the 'data.out' file is also written to this directory). The use of these programs is described in Section 4.6.

---

[1]The first two lines of the file, which contain information about when the file was produced, must first be deleted.

[2]All file paths referred to in this report are relative to the installation directory "project/".

Name	*Description*
sim	Learning algorithm with a global horizontal balance reflex.
sim1	Early version of simulator preprogrammed with the desired walking pattern to demonstrate the goal of the learning algorithm. (The behaviour is hard-wired in this version, and no learning occurs.)
sim2	Hard-coding of a statically stable walking pattern using the global move-leg-back reflex to demonstrate that the desired gait is also possible in this case. As with **sim1**, this version is not connected to the learning algorithm.
sim3	Learning algorithm with the global move-leg-back reflex.
sim4	Learning algorithm with four extra learnable move-leg-back behaviours, and no global or hard-wired reflexes.

Table 4–1: Summary of the Different Versions of the Simulator

4.4 The Different Versions of the Simulator

There are several different versions of the simulator, as summarised in Table 4–1. **sim**, **sim3** and **sim4** are versions which are linked to the learning algorithm, whereas **sim1** and **sim2** contain preprogrammed walking patterns to demonstrate that the goals of the algorithm are achievable. The display for all of the versions involving the learning algorithm is the same, except that **sim4** has information regarding eight behaviours rather than four.

4.5 Using the Simulator

4.5.1 Getting Started

The code for the simulator is in the directory "simulator/". Before running the program, the following command must be given to set some environment variables so that the XView facilities can be accessed:

- **source smudge**

The file 'smudge' contains the necessary export commands. The simulator may then be invoked with the command:

- **sim**

Other versions may be invoked similarly, using the commands **sim1**, **sim2** etc.

4.5.2 The Canvas

The canvas contains a display of Smudge in its current position, as described in Section 4.2. The face indicates which end is the front, and the cross indicates the presumed centre of mass, as used in calculations. At each time step, the distance that Smudge advances along the x and y axes is taken to be the average distance moved in these directions by each of the feet currently on the ground. Whenever Smudge wanders off the edge of the canvas, it is redrawn at its starting position towards the left of the window.

4.5.3 The Control Panel

The control panel is situated on the left-hand side of the window.

The execution of the simulation is controlled by the *Start* and *Stop* buttons in the top left-hand corner of the window. The user may exit the simulation by pressing the *Quit* button. The speed at which the simulation runs can be controlled via the slider just under these buttons. At speed 0, the user may step through the algorithm one time step at a time, by repeatedly pressing the *Start* button.

In the centre of the control panel there is a display of the position of each leg. The *alpha* angle[3], or the angle that a leg is displaced in the horizontal plane, is shown and may be changed by the user at any point to any integer value in the

[3]An alpha angle of zero corresponds to the leg being perpendicular to the side of the body. Because of the way that the servos were arranged, it was decided to use the convention that a positive alpha angle refers to a forward displacement for the front legs, but to a backward displacement for the rear legs.

range −45 to +45. Below this there is a display to show whether the leg is up or down — this may also be changed by the user. At the bottom of this middle section of the panel is a message displaying the *beta* angles for the simulation, i.e. the angles of displacement in the vertical plane which correspond to a leg being up or down. These angles are not adjustable by the user in the current version of the simulation.

Below this section is a set of information about how well the current run is progressing. The first figure is the number of time steps that have elapsed since the start of the present run of the simulation, i.e. how many times that algorithm has repeated its basic control loop (see Figure 2–1). Below this are figures for the percentage of positive and negative feedback received. Note that these figures refer to the *preceding 200 time steps*, and not to the entire run, so that any change in performance may be more readily noticed. Finally, figures are given for the distance travelled along the x-axis (forwards and backwards) and the y-axis (from side to side). These numbers are given in centimetres, as calculated using the dimensions of the real robot.

There is a message at the bottom of the panel to say whether Smudge is stable or not. As explained earlier, when it is standing on three legs, there is also a dotted line drawn between two of the legs to indicate how stable that position is. If Smudge is *unstable* in that position, then the dots are drawn closer together to emphasise that fact.

4.5.4 The Behaviour Information Panel

This is placed at the top right of the window.

The behaviours that individual behaviour numbers refer to are listed in Table 4–2.

For each behaviour, a list of its current preconditions is displayed. Each precondition is represented as a four-character string of the form Lxyz, where x is a leg

Behaviour Number	*In* **sim** *and* **sim3**	*In* **sim4**
0	swing leg 0 forward	swing leg 0 forward
1	swing leg 1 forward	move leg 0 backward
2	swing leg 2 forward	swing leg 1 forward
3	swing leg 3 forward	move leg 1 backward
4	–	swing leg 2 forward
5	–	move leg 2 backward
6	–	swing leg 3 forward
7	–	move leg 3 backward

Table 4–2: Key to Behaviour Numbers used in the Simulator

number between 0 and 3 [4]; y is either U for up, F for forward, or B for backward; and z is either T for true, or F for false.

When a behaviour is monitoring a condition, then this condition is displayed, according to a similar code, under the *Monitoring* column, and the number of time steps for which the monitoring has been in progress is shown under the *Clock* column.

4.6 Plotting Data from a Simulation Run

As mentioned earlier, the data file from a run of the simulation, called 'data.out', together with MATLAB M-files for plotting the data, can be found in directory simulator/data/. Before using MATLAB, the first two lines should be stripped from the data file. Once in MATLAB, the command

- **load data.out**

will load in the data. The columns of data must then be tagged with names so that they can be referred to by the plotting utilities or by the user.

---

[4]Leg 0 is the front left leg, Leg 1 the front right, Leg 2 the rear right, and Leg 3 the rear left.

For **sim** and **sim3**, which both have four learnable behaviours, data is written to the output file every ten time steps. To name the columns of data produced by either of these, type

- **simplot**

This runs an M-file which contains the necessary MATLAB commands.

If **sim4** has been used, then eight learnable behaviours are involved, so the data file is larger than in the other cases. As this is so, the data is only written every 25 time steps, and it should be named using the command

- **simplot8**

When the data has been labelled in this way, a number of programs to plot graphs are provided, as described below.

4.6.1 Feedback Percentages

To plot the percentage of time that positive feedback was received against time steps, type

- **plotppfb**

Similarly, for negative feedback, type

- **plotpnfb**

4.6.2 Relevance and Reliability

The relevance and reliability of a behaviour may be plotted against time steps with the command

- **plotrel**n

where n is a behaviour number between 0 and 3 (or 0 and 7 for **sim4**).

4.6.3 Behaviour Statistics

The positive and negative statistics relating to each behaviour, from which are calculated the relevance and reliability for that behaviour, may be plotted for data from **sim** and **sim3** with the command

- **plot**x**s**$n$$y$

where x is **p** for positive feedback or **n** for negative feedback; n is a behaviour number between 0 and 3; and y is **a** for the statistics referring to when the behaviour is active (cells j and l in Table 2–1), or **i** for inactive (cells k and m).

4.6.4 Monitoring Statistics

The statistics relating to when a behaviour is monitoring a condition may be plotted for data from **sim** and **sim3** with the command

- **plotm**x**s**$n$$z$

where x is **p** for positive feedback or **n** for negative feedback; n is a behaviour number between 0 and 3; and z is **on** for the statistics referring to when the condition was on (cells n and p in Table 2–2), or **off** for when it was off (cells o and q).

4.6.5 Plotting Other Graphs

The labels given to each of the columns of the data file are listed in Table 4–3. The user may use these labels to plot any other graph that may be of interest, either by entering commands on-line from within MATLAB, or by creating M-files similar to those provided for the existing utilities.

Label	*Description*
tsc	time step counter
lnu$n$	leg n up condition set (1=true, 0=false)
lnf$n$	leg n forward condition set (1=true, 0=false)
lnb$n$	leg n backward condition set (1=true, 0=false)
prec$xyyz$	precondition number yy for behaviour x with truth value z (1=true, 0=false)
ba$n$	behaviour n active (1=true, 0=false)
bm$n$	behaviour n currently monitoring (1=true, 0=false)
psxyz	positive statistics for behaviour x, row y, column z of statistics table
nsxyz	negative statistics for behaviour x, row y, column z of statistics table
bnmp$n$	behaviour n monitoring precondition number (preconditions are numbered from 0 to 11, such that LOU is 0, LOF is 1, LOB is 2, L1U is 3 ... and L3B is 11)
mc$n$	monitoring clock for behaviour n
bmppsxyz	positive statistics for monitoring a condition by behaviour x, row y, column z of statistics table
bmpnsxyz	negative statistics for monitoring a condition by behaviour x, row y, column z of statistics table
ppfb	percentage of time receiving positive feedback
pnfb	percentage of time receiving negative feedback
rele$n$	relevance of behaviour n
reli$n$	reliability of behaviour n

Table 4–3: Labels used to refer to Data from within MATLAB

Chapter 5

Experimental Design

The preceding chapters have described the learning algorithm, the construction of the robot, and the details of the simulator. With all of these components in place, experiments were conducted to test the performance of the algorithm under a variety of conditions. It has already been mentioned that the robot was not completed in time to allow testing of the algorithm on it, so all the experiments concerning learning were performed in simulation only. However, an attempt will be made to equip Smudge with a hard-wired walking behaviour by directly programming the microcontrollers.

5.1 'Hard-Wired' Gaits

5.1.1 On Smudge

Although there was insufficient time to test the algorithm on the physical robot, it was still important to get some idea of how accurately the simulation modelled the real situation. It was therefore decided that the microcontrollers should be programmed to attempt to produce a statically stable gait. If this attempt was successful, then it would prove that the ultimate goal of the learning algorithm was achievable in reality.

Refer to Section 6.1.1 of the Results chapter.

5.1.2 In Simulation

As well as attempting to equip the physical robot with a hard-wired walking behaviour, it was equally important to do the same thing on the simulation, in order to have some confidence that the goal of the learning algorithm was at least possible.

The first attempt involved programming the legs to swing forwards and move backwards at suitable speeds and with such coordination that the simulated robot was stable at every point in the gait.

A second attempt involved trying to achieve the same goal by incorporating the global horizontal balance reflex, as used by one version of the learning algorithm. In other words, the timing and speed of the swing-leg-forwards behaviours were preprogrammed, but the movement of the legs on the ground was controlled by the horizontal balance reflex.

The outcome of these attempts is described in Section 6.1.2.

5.2 Testing the Algorithm in Simulation

5.2.1 Basic Trials

The most basic question that can be asked of the algorithm is "Does it work?". To answer this, a number of trials were devised for each of the three versions of the algorithm (i.e. one with the global horizontal balance reflex, one with the global move-leg-back reflex, and the other with the four learnable move-leg-back behaviours). The purpose of the trials was to explore some of the parameter space of the algorithm in order to determine what parameter values led to the best performance. Each trial differed from the others in the value that either one of the global parameters of the algorithm, or one of the thresholds relating to leg movements, took. Details of the default parameter values for each trial are given in Table 5–1, and the changed parameter values for each trial are listed in Table 5–2.

An additional consideration concerning these trials is where to place the legs at the start of a run. This is an important decision, because the initial leg positions

Parameter / Threshold	*Version* **A** *(global horizontal balance reflex)*	*Version* **B** *(global move-leg-back reflex)*	*Version* **C** *(learnable move-leg-back behaviours)*
correlation threshold	0.65	0.65	0.65
monitor duration	50	50	50
reliability target	0.95	0.95	0.95
statistics decay rate	0.90	0.90	0.90
maximum alpha angles for legs	40	40	40
swing angle for raised legs at each time step	10	10	10
threshold for triggering forward and backward conditions	15	15	15

Table 5–1: Summary of Default Parameter Values Used in the Basic Trials

are also those to which the legs are returned each time that the robot falls over — different initial positions will presumably result in markedly different performance. The obvious choice is to have all legs on the ground at the start of a run, all perpendicular to the side of Smudge's body (i.e. alpha = 0°). However, a look at Figure 2–2 reveals that there is no one point in this gait at which all four legs are actually in this position. Therefore, it was decided to run each trial twice, with different initial positions for the legs; in one run, they were all on the ground, perpendicular to the side of the robot's body (*Position 'A'*); in the other run, all legs were on the ground, but this time, their alpha angles were set to values which had been observed in the analysis of a statically stable gait at a point where all four legs *were* on the ground[1] (*Position 'B'*).

From the performance of the three versions of the algorithm in these trials, the relative suitability of each one may also be assessed.

The results obtained from these trials are described in Section 6.2.1.

[1]Specifically, the alpha angles for the four legs were: Leg 0 = −38°, Leg 1 = 8°, Leg 2 = 8°, Leg 3 = −38°. The gait from which these angles were obtained is demonstrated in program **sim1** — see Section 6.1.2.

Trial No.	*Version* **A**	*Version* **B**	*Version* **C**
1	default values	default values	default values
2	maximum alpha = 45	maximum alpha = 45	maximum alpha = 45
3	threshold for f/b conds = 30	threshold for f/b conds = 30	threshold for f/b conds = 30
4	correlation threshold = 0.85	swing angle = 15	swing angle = 15
5	monitor duration = 200	correction[a] = 3	crawl angle[b] = 3
6	reliability target = 0.75	correlation threshold = 0.85	correlation threshold = 0.85
7	statistics decay rate= 0.7	monitor duration = 200	monitor duration = 200
8	—	reliability target = 0.75	reliability target = 0.75
9	—	statistics decay rate = 0.7	statistics decay rate = 0.7

[a] the angle by which a leg on the ground is moved backwards at each time step by the move-leg-back reflex

[b] the angle by which a move-leg-back behaviour moves a leg at each time step

Table 5–2: Summary of Basic Trials Performed on Each Version of the Algorithm

5.2.2 The Effect of Noise

In the standard version of the simulation, all movements are completely deterministic — for a given set of parameter values, the performance on one run will be exactly the same as that on the previous run. Assuming that some versions of the algorithm displayed reasonable performance with this simplification, then it would be of interest to see how their performance is affected by introducing noise into the simulation. This noise would take the form of a degree of randomness in the movement of each servo.

The results of these experiments are described in Section 6.2.4.

Chapter 6

Results

The previous chapter gave a brief description of the experiments that were conducted, and also some explanation of why they were conducted. The results of these trials are now described.

6.1 'Hard-Wired' Gaits

6.1.1 On Smudge

Unfortunately, the construction of Smudge was not completed in time to allow any testing. However, some programs have been written to drive the servos via the microcontrollers, and it is hoped to be able to demonstrate a hard-wired walking behaviour on the robot in the week after the completion of this report.

6.1.2 In Simulation

It turned out that this goal was indeed possible — a statically stable walking gait was demonstrated for both attempts at producing a hard-wired behaviour on the simulation.

The result of the first attempt, involving hard-wiring of both the swing-leg-forwards and the move-leg-backwards behaviours, is demonstrated in program sim1.

The speed at which the legs swung forward, relative to that at which they pushed backwards on the ground, affected the margin of stability[1] of the robot throughout the gait; a high-speed swing produced a more stable gait than a low-speed swing.

The result of the second attempt, using the global horizontal balance reflex, is demonstrated in program **sim2**.

The gait shown in **sim1** is an example of the general case for statically stable gaits on quadrupeds where there are periods when all four legs are on the ground. That shown in **sim2** is the limiting case where one leg is raised as soon as the previously raised leg is lowered — this is the gait discussed in Section 2.5 and illustrated in Figure 2–2.

6.2 Testing the Algorithm in Simulation

6.2.1 Basic Trials

It soon became apparent that most trials were not producing a learned pattern of behaviour which could be sustained for long periods — in many cases, new preconditions were constantly being monitored, adopted or dropped by each behaviour, so that the performance of the robot as a complete system varied over time.

This presents a problem with the classification of how good individual trials were, and how good they were relative to other trials. Therefore, the method of studying the trials was to let them run for between 5000 and 15000 time steps (depending on how promising each one was looking), and noting the qualitative performance of the robot over this time, together with some quantitative information at interesting phases of the run (e.g. the percentage of time receiving positive and negative feedback during the phase).

---

[1]The margin of stability is the minimum distance between the projection of the robot's centre of mass onto the ground plane and the boundary of the polygon of support provided by those legs which are on the ground.

After studying the graphs from the first few trials, a marked difference was observed between the activity over time of the front legs compared to the rear legs (i.e. there was a large difference in the statistics that were emerging for the front legs compared to those for the rear legs). On investigation of the simulation, it was found that this asymmetry in the statistics was due to a corresponding asymmetry in the design of the robot on the simulation. The asymmetry was such that, when all four legs were on the ground in initial position A, and one of the rear legs was raised with the other legs remaining stationary, then the robot was (just) stable. However, when one of the front legs was raised in a similar manner from the same initial position, then the robot was (just) unstable. Because a robot which had a perfectly symmetrical arrangement of legs front and rear would be theoretically (just) stable if any one of the legs were raised from this initial position, it was decided to overcome this asymmetry in the simulation by allowing a small leeway in the margin of stability before the robot was deemed to have fallen over. Specifically, the projection of the robot's centre of mass was allowed to be up to 1cm *outside* the polygon of support provided by the legs on the ground before Smudge actually fell over in the simulation[2].

All of the trials reported below incorporated this fix.

Version A — Global Horizontal Balance Reflex

The observed performance did not vary significantly between any of the seven basic trials, and none of them produced particularly good results.

For initial leg position A (all legs on ground, alpha angle = 0°), the percentage of time in which positive feedback was received had typically reached 15–20% by 5000 time steps. The corresponding figure for negative feedback was 35–40% (see Figure 6–1). In other words, Smudge was falling over about twice as frequently as it was actually moving forward. The behaviours generally continued to monitor new conditions, and no stable sets of preconditions were found which survived for more than a couple of hundred time steps.

---

[2]That is, 1cm as measured on the physical robot.

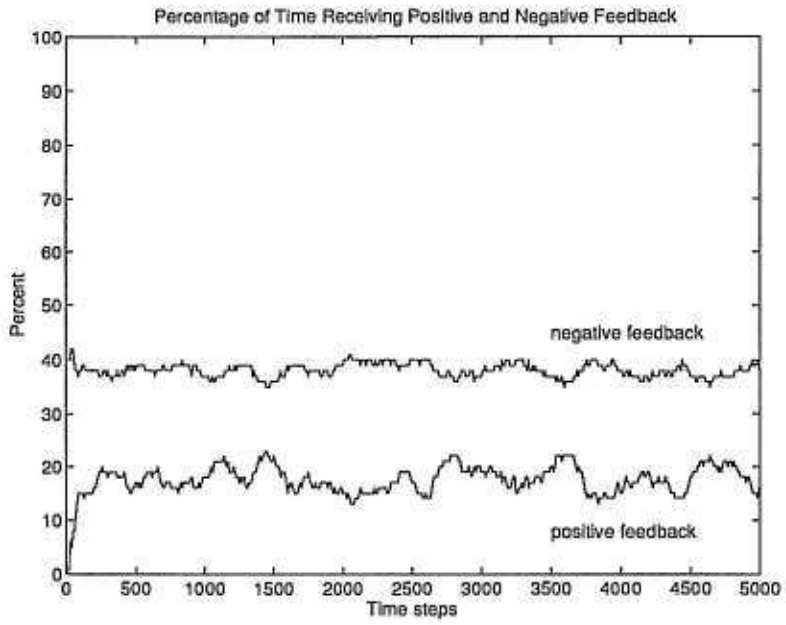


Figure 6–1: Graph of Feedback *v.* Time for Trial A1

For initial leg position B (staggered legs), similar qualitative results were observed, but the overall feedback figures were slightly worse — generally between 10–20% for positive feedback, and 35–40% for negative feedback after 5000 time steps.

Version B — Global Move-Leg-Back Reflex

Generally, the overall performance of this version of the algorithm was better than for Version A, but the results were still poor. At some points in most of the trials, the positive feedback figure exceeded the negative feedback figure, but only by a maximum of 10–15% (for example, see Figure 6–2).

The best performance was observed in Trial 1 (using all of the default values) using initial leg position A (unstaggered). In this run, after about 5000 time steps, behaviour 3 (swing leg 3 forward) adopted the precondition that leg 3 should be raised. This appeared strange at first, as, in effect, it meant that the behaviour would never become activated (as leg 3 could only be raised by the action of behaviour 3). On studying the behaviour of the whole robot, however, this precondition made more sense, as it meant that the robot was stable for more of the

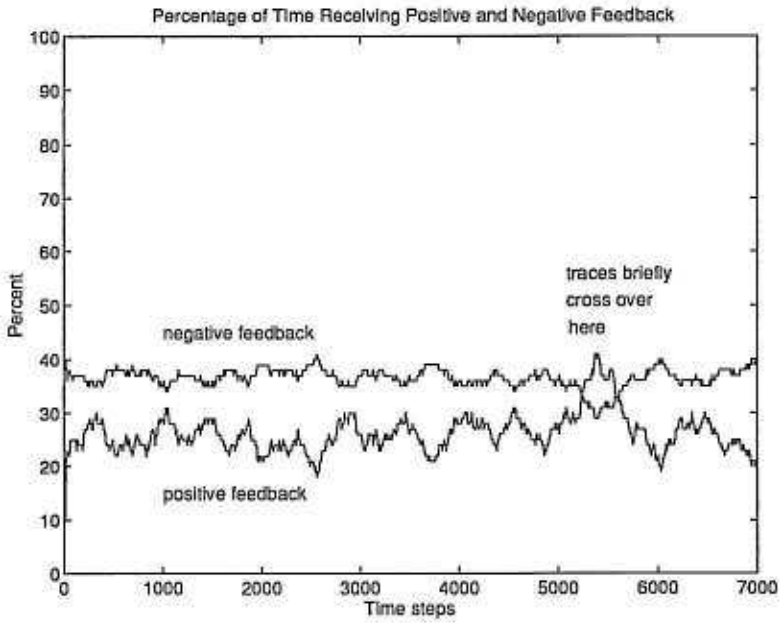


Figure 6–2: Graph of Feedback *v.* Time for Trial B1

time (as leg 3 was always on the ground). Thus, the algorithm was able to explore preconditions for other legs under more stable conditions. The balance between positive and negative feedback reached 40% to 25% at this point. Unfortunately, it appeared that the preconditions learned by other behaviours while leg 3 was constantly on the ground were not suitable for when it was able to be raised — eventually, behaviour 3 dropped the aforementioned precondition, and the overall performance of the robot deteriorated to roughly 20% positive feedback and 40% negative feedback after 10000 time steps.

The feedback percentages for the other trials reached similar values (they were typically in the range 20–30% for positive feedback, and 35–40% for negative feedback), with the trials using initial leg position A (unstaggered) marginally outperforming those using position B (staggered).

Version C — Learnable Move-Leg-Back Behaviours

The trials with this version produced substantially better performance than those of the preceding versions.

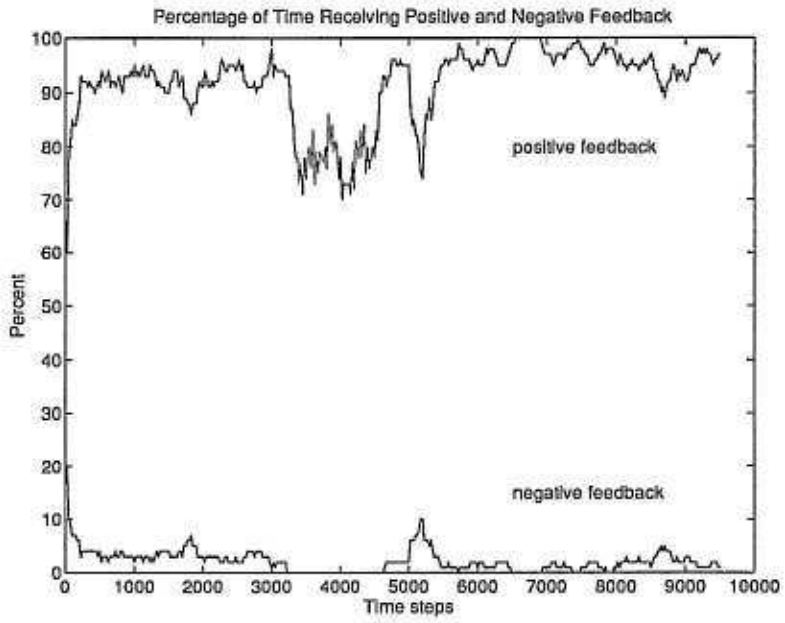


Figure 6–3: Graph of Feedback *v.* Time for Trial C2S

Although none of the trials reached a state where a set of preconditions had been learned and no further monitoring took place, there were periods of up to 2000 time steps in some of the trials when no existing preconditions were dropped, and no new ones were adopted. For example, in trial 2S[3], the preconditions shown in Table 6–1 were learned by 6500 time steps. These were retained for a further 2000 time steps, during which time, the percentage of time receiving positive feedback was in the range 95–100%, and the corresponding figure for negative feedback was 0–2% (see Figure 6–3). However, new conditions were still being monitored occasionally during this time; eventually some of the existing preconditions were dropped and new ones were adopted, which led to a deterioration in performance.

With this version of the algorithm, the trials with initial position B performed marginally better than those with position A. However, there were still no trials in which a good combination of preconditions was found *and retained indefinitely.* In order to investigate why these apparently good solutions were not remaining stable, various graphs were plotted of the data produced from trial 2S. It was

[3]'S' denotes the staggered initial leg position (B)

Behaviour	*Precondition*	*Behaviour*	*Precondition*
Swing leg 0 forward	L3F T	Move leg 0 back	—
Swing leg 1 forward	L2F T	Move leg 1 back	—
Swing leg 2 forward	L3F F	Move leg 2 back	—
Swing leg 3 forward	—	Move leg 3 back	L3F T

Table 6–1: Preconditions learned by Version C, trial 2S

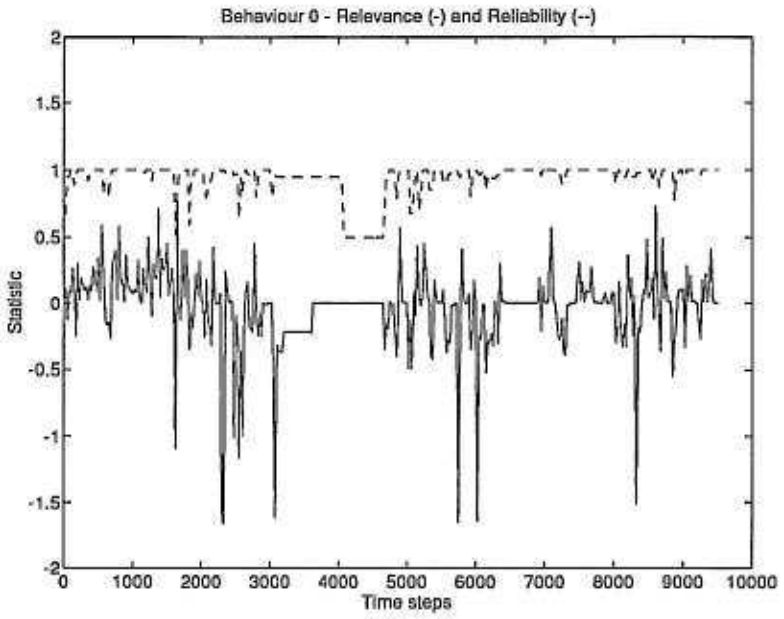


Figure 6–4: Graph of Statistics *v.* Time for a Behaviour from Trial C2S

found that the statistics were far too transitory for a stable state to emerge — even with a decay rate of 0.90. This is illustrated by the graph of the relevance and reliability of Behaviour 0, shown in Figure 6–4.

A new set of trials was therefore conducted, all using a slower statistics decay rate (0.99). The values of most of the other parameters were the same as for the original trial 2S, but the monitor duration, correlation threshold and reliability target were varied between these new trials. A summary of the new trials is shown in Table 6–2.

The best performance was observed in trial 12. The plot of relevance and reliability for this trial is shown in Figure 6–5. It can be seen that these statistics are much smoother than for the original trial illustrated in Figure 6–4. A graph of

Trial No.	Stats Decay Rate	Monitor Duration	Reliability Target	Correlation Threshold	Initial Leg Position
10	0.99	50	0.95	0.65	staggered
11	0.99	120	0.95	0.65	staggered
12	0.99	65	0.75	0.65	staggered
13	0.99	70	0.83	0.60	staggered
14	0.99	65	0.75	0.65	unstaggered

Table 6–2: Summary of Additional Trials Performed for Version C

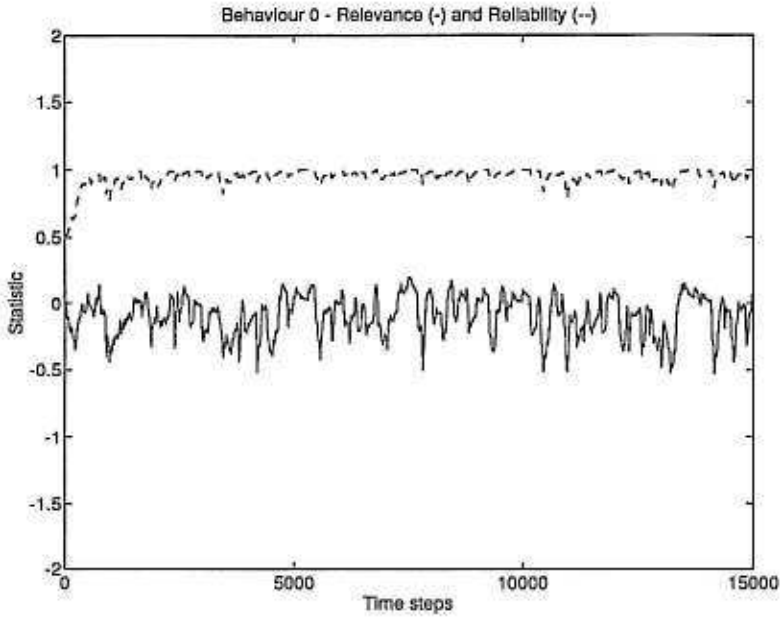


Figure 6–5: Graph of Statistics *v.* Time for a Behaviour from Trial C12

the percentage of time receiving positive and negative feedback for a run of 15000 time steps is shown is Figure 6–6.

The preconditions learned on trial 12, shown in Table 6–3, are different to those learned in the best phase of trial 2. They were learned rapidly, and remained for the duration of the trial. Some behaviours did occasionally monitor new conditions, but no other ones were actually adopted. Note that no preconditions were learned concerning the movement of leg 3. This is an artifact of the staggered leg position to which Smudge returned whenever it fell over; leg 3 was returned to more or less the position where it should have landed after a swing-leg-forward movement anyway, so that it was not necessary to learn any preconditions to produce near-optimal (but not optimal) performance.

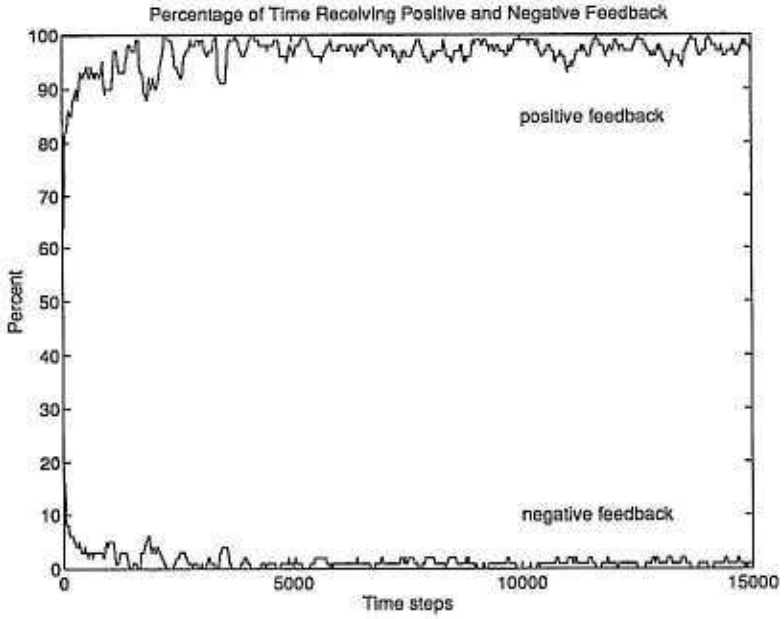


Figure 6–6: Graph of Feedback *v.* Time for Trial C12

Behaviour	*Precondition*	*Behaviour*	*Precondition*
Swing leg 0 forward	—	Move leg 0 back	L0F T
Swing leg 1 forward	L2F T	Move leg 1 back	—
Swing leg 2 forward	L2B T	Move leg 2 back	—
Swing leg 3 forward	—	Move leg 3 back	—

Table 6–3: Preconditions learned by Version C, trial 12

Behaviour	*Preconditions*
Swing leg 0 forward	L1U F, L2U F, L3U F, L3F T
Swing leg 1 forward	L0U F, L2U F, L3U F, L2F T
Swing leg 2 forward	L0U F, L1U F, L3U F, L1B T
Swing leg 3 forward	L0U F, L1U F, L2U F, L1F T

Table 6–4: Preconditions Preprogrammed into Version A, trial PA1S

6.2.2 Preprogramming the Behaviours

As none of the trials described in the last section actually achieved the target of learning a gait where negative feedback was completely eliminated, it was decided to try to preprogram a suitable set of preconditions into each behaviour for each version of the algorithm. If the desired gait could be produced by this method, preconditions could then be stripped away from the behaviours one by one in order to find the point where the algorithm fails to re-learn the solution set.

Version A — Global Horizontal Balance Reflex

The set of preconditions that were programmed into the behaviours on the first attempt are shown in Table 6–4. The first three preconditions for each behaviour were chosen to specify that a leg could be raised only if all other legs were on the ground, and the final precondition for each behaviour specified the required coordination between the four legs.

Default parameter values (Table 5–1) were used in the trials unless otherwise stated.

The first trial (PA1S)[4] was with a staggered initial leg position. Performance was initially perfect. However, after a very brief period (about 80 time steps), the robot got into a position where all four legs were on the ground, the horizontal angles summed to zero and none of the behaviours' precondition lists was fulfilled (see Figure 6–7(a)). Hence Smudge remained inactive until the statistics eventually

[4]'P' denotes a trial with preprogrammed preconditions.

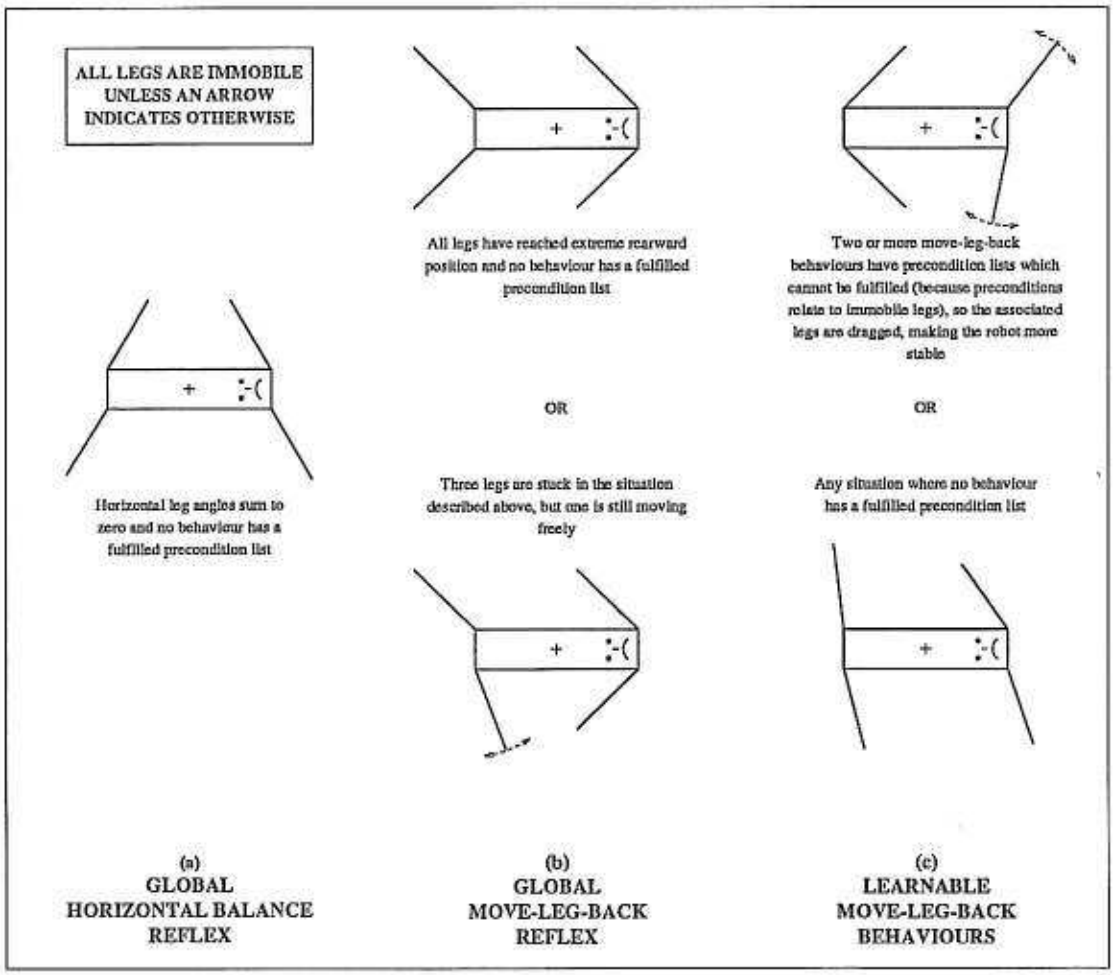


Figure 6–7: Situations Where Smudge was Liable to get Stuck During Learning

decayed to a point where the behaviours' reliability values were below threshold (after about 1000 time steps). At this point, the behaviours started to monitor new conditions, some of the preprogrammed preconditions were dropped and the stable walking gait was lost.

Further trials using different parameter values and/or the unstaggered initial leg position were not successful either — some would work perfectly for up to 2000 time steps, but eventually some of the preconditions would be dropped and the stable gait lost in all trials.

The gait illustrated in Figure 2–2, which satisfies the horizontal balance constraint, was peculiar in that there was no period during the gait in which all four legs were on the ground (apart from the four instantaneous moments when one

leg was just being lowered and another was just being raised). Therefore, a number of trials were performed where the initial leg positions involved one leg being raised. Specifically, all legs had an alpha angle of 0°, legs 0, 1 and 3 were on the ground, but leg 2 was raised (this will be referred to as initial leg position C). Note that Smudge is only just stable in this position, so it would not be a good initial position for the physical robot. However, the trials were done to investigate the performance of the algorithm a little further.

The first trial with this new initial leg position used the default parameter values except for the following; 'leg forward' threshold = 5°, monitor duration = 150, reliability target = 0.75, and statistics decay rate = 0.90. (This was felt to be a fairly good set of values from the results of the preceding trials). The preconditions were preprogrammed as in Figure 6–4.

The results of this trial were very good — Smudge did not fall over and no other conditions were monitored for the entire duration of the run (3000 time steps).

Two similar trials were run, one with a statistics decay rate of 0.96, and one with a rate of 0.99. Both of these performed just as well as the first run. With a decay rate of 0.99, a few conditions were monitored in the first 300 time steps of the run, but they were not adopted, and no more monitoring occurred after that point. The plot of feedback percentages against time for the latter trial (PA6) is shown in Figure 6–8.

Having found this good configuration, it was of interest to see how many of the preprogrammed preconditions may be removed so that either the performance is not affected, or the algorithm re-adopts them. A number of trials were therefore performed where one or more of the preconditions was omitted. In trials where one of the preconditions of one behaviour which specified that another leg should be on the ground (e.g. the precondition L1U F for Behaviour 0) was omitted, the performance of the robot was not affected and the omitted precondition was not re-adopted. However, if one of the preconditions relating to the coordination between the four legs was omitted, or if more than one precondition was omitted in any one trial, then the algorithm was unable to re-learn them, and no stable walking pattern emerged.

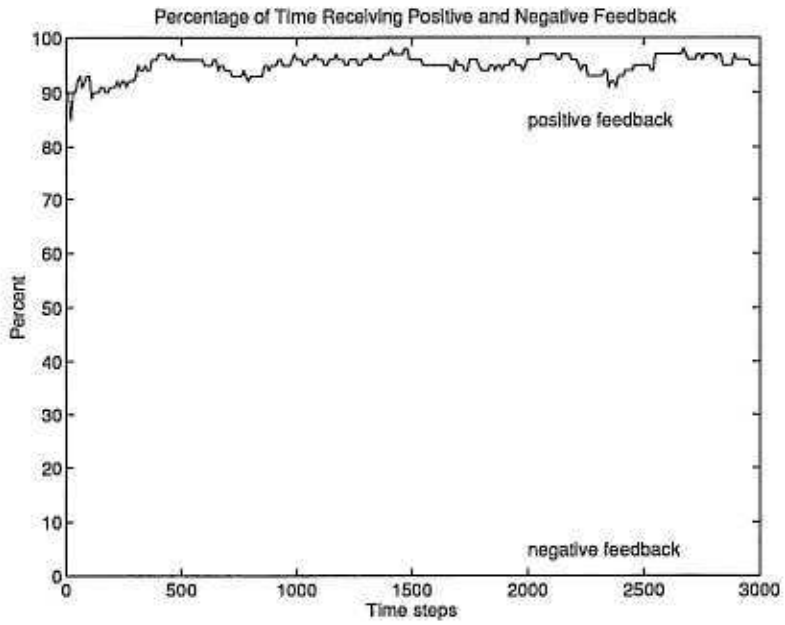


Figure 6–8: Graph of Feedback *v.* Time for Trial PA6

Thus, the algorithm was unable to learn a perfect solution for the problem, even when placed a very short distance away from it in the space of possible precondition combinations.

Version B — Global Move-Leg-Back Reflex

The first trial with version B of the algorithm was preprogrammed with the same preconditions as initially used for version A, listed in Table 6–4. With the staggered initial leg position (trial PB1S), Smudge did not monitor any new conditions for the duration of the 3000 time step run. However, it did fall over on a number of occasions, as illustrated by the non-zero negative feedback plot in Figure 6–9.

The algorithm fared less well when running with the unstaggered initial leg positions — all of the preprogrammed preconditions had been dropped after 2500 time steps, and no stable walking pattern was observed.

As with version A, some trials were conducted, using the staggered initial leg positions, where one or more of the preprogrammed preconditions were omitted. The results were similar to those for version A, i.e. there were some preconditions which could be omitted without affecting the performance of the robot (so

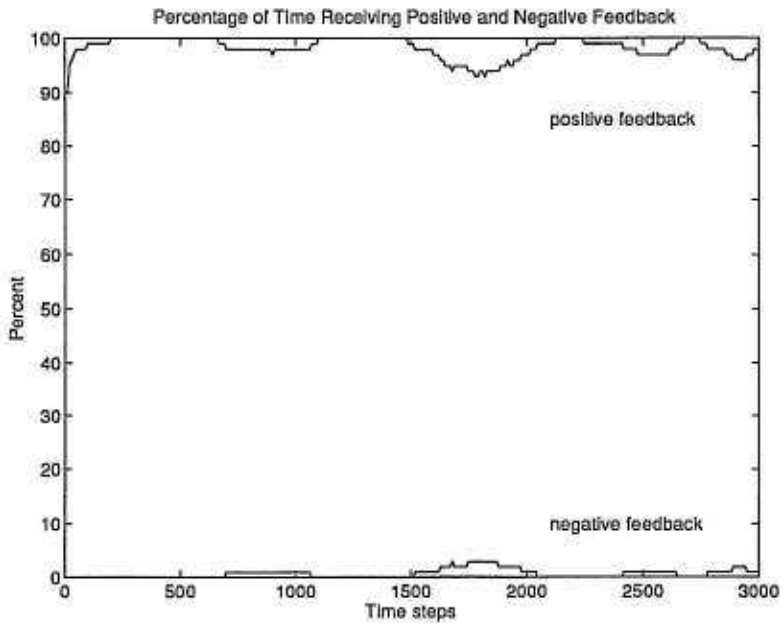


Figure 6–9: Graph of Feedback *v.* Time for Trial PB1S

they were not necessary in the first place), but if any essential preconditions were omitted, then the algorithm was unable to relearn them.

Version C — Learnable Move-Leg-Back Behaviours

Version C was equipped with the preprogrammed preconditions shown in Table 6–5. Somewhat surprisingly considering that this version had shown the best results in the basic trials, no stable walking pattern emerged with either the staggered or the unstaggered initial leg positions. In both cases, the behaviours frequently monitored conditions, adopting new preconditions and dropping some of the preprogrammed ones.

Several other trials were conducted, using different parameter values, different preprogrammed preconditions, and/or the initial leg position C described above, but none performed any better.

In most of the trials, Smudge was getting stuck in a position where all four legs were on the ground, but none of the behaviours' precondition lists were fulfilled, so that the whole robot was inactive (see Figure 6–7(c)). This situation was more likely to happen in Version C than in Versions A or B, because the latter pair had

Behaviour	*Preconditions*
Swing leg 0 forward	L1U F, L2U F, L3U F, L3F T
Move leg 0 back	L0U F
Swing leg 1 forward	L0U F, L2U F, L3U F, L2F T
Move leg 1 back	L1U F
Swing leg 2 forward	L0U F, L1U F, L3U F, L1B T
Move leg 2 back	L2U F
Swing leg 3 forward	L0U F, L1U F, L2U F, L1F T
Move leg 3 back	L3U F

Table 6–5: Preconditions Preprogrammed into Version C, trial PC1

Parameter / Threshold	*Value*
monitor duration	65
reliability target	0.75
statistics decay rate	0.99
maximum alpha angles for legs	45
threshold for triggering forward and backward conditions	15

Table 6–6: Default Parameter Values Used with Extensions to Algorithm

hard-wired reflexes to move the legs, so that the system as a whole was less likely to get stuck in a log jam.

6.2.3 Other Extensions

As most of the trials reported so far have not performed particularly well, it was decided to experiment with a few extensions to the basic algorithm described in Chapter 2.

In all of the following, unless otherwise stated, the parameter values and thresholds used are as listed in Table 6–6. These values were chosen by consideration of the parameter values in the best trials reported in previous sections. Those parameters not listed in this table took their default values as listed in Table 5–1.

Not Returning Legs to Initial Position after a Fall

It was apparent in the preceding trials that the algorithm was not getting a chance to explore a wide range of situations where the legs were in positions that may have prompted the behaviours to learn useful preconditions. This was because Smudge was falling over frequently, and the legs would be returned to their initial positions after each fall. Thus, the number of times that a particular leg may have been observed in a position near its extreme of movement in either the forward or backward direction was often relatively low.

In order to help the algorithm explore a wider range of the problem space, it was decided to try running it in a situation where the legs were *not* returned to their initial positions after a fall. Instead, any legs that were raised when Smudge fell were lowered, but the alpha angles of each of the legs were unchanged.

The following trials were all conducted with the unstaggered initial leg position. This was, of course, of little relevance, as the legs were not returned to this position after each fall.

Version A — Global Horizontal Balance Reflex In this trial, Smudge often got stuck in positions where all four legs were on the ground with horizontal angles summing to zero, and no precondition lists fulfilled. However, when the robot was moving, the percentage of time in which positive feedback was received outweighed that in which negative feedback was received by 40% to 20% on some occasions. This is a marked improvement on the performance of the algorithm when the legs were returned to their initial positions.

At some periods during the learning, the algorithm seemed to be making good progress, and Smudge appeared to be learning the staggered start position for itself through adopting a suitable set of preconditions. However, it would eventually get stuck in a position where all four legs were immobile, and the algorithm could only overcome this deadlock by dropping some of these useful preconditions.

Version B — Global Move-Leg-Back Reflex With this version, Smudge soon got into a position where all of its legs were swept forward. From here, they

would move backwards a little under the move-leg-back reflex until the swing-leg-forward behaviour for that leg was activated, returning the leg to the forward position. If the robot fell over at any stage, then the legs were just placed on the ground, and the behaviour pattern could continue more or less uninterrupted. The resulting behaviour pattern was that each leg remained near its extreme forward position for most of the time, just moving backwards by a few degrees (and hence contributing to the positive feedback) before swinging forward again. The problem was that the negative feedback received when Smudge fell over was not contributing to the statistics enough to force the reliability of the behaviours below threshold. Therefore, without having a reliability target of 1.00, the system did not develop a proper gait.

Version C — Learnable Move-Leg-Back Behaviours The trial with Version C produced very similar results to Version B — as the legs were not returned to their initial position after a fall, the robot could 'cheat' and receive sufficient positive feedback without developing a viable walking gait.

A Hard-Wired 'Only Lift One Leg at a Time' Rule

Another method to increase the overall stability of the system, and thereby promote the exploration of more of the problem space by the algorithm, is to build a 'hard-wired' rule into the algorithm which prevents more than one leg ever being raised simultaneously.

This rule was incorporated into the algorithm as follows; those swing-leg-forward behaviours which were eligible to become activated[5] were noted. If no swing-leg-forward behaviour was already active (i.e. if no leg was raised), then one of these was picked at random to be activated. Otherwise, none was activated.

---

[5]That is, they were not already active, and all of the preconditions on their list had been fulfilled.

Behaviour	*Precondition*
Swing leg 0 forward	L3F T
Swing leg 1 forward	L2F T, L1F F
Swing leg 2 forward	L1B T
Swing leg 3 forward	L1B F

Table 6–7: Preconditions learned by Version A, trial RA2

Version A — Global Horizontal Balance Reflex Several trials were conducted with Version A, using the staggered and unstaggered initial leg positions, and changing the value of the reliability target parameter. However, as found with many previous trials with Version A, the robot was liable to get stuck in a situation where all four legs were on the ground, and none could move because the horizontal angle sum of the legs was zero and no behaviours had a fulfilled precondition list.

With an unstaggered start and a reliability target of 0.95 (trial RA2)[6], the robot did reach a good level of performance after getting stuck many times during its early stages of learning. The plot of feedback is shown in Figure 6–10[7]. After 10000 time steps, positive feedback was being received 95% of the time, and negative feedback 0%. The preconditions learned at this stage are shown in Table 6–7. Note that the behaviours were still monitoring other conditions at this stage, that the negative feedback percentage was not always zero even after 10000 time steps, and that the robot still gets stuck with all legs on the ground from time to time (at which point the percentage figures for receiving positive and negative feedback both drop to zero).

---

[6]'R' denotes trials where the 'only lift one leg at a time' rule was used.

[7]The regions of the graph where positive and negative feedback figures are both zero indicate times when Smudge was stuck as illustrated in Figure 6–7(a). This situation is only resolved when the behaviours' statistics decay to below threshold so that new conditions are monitored and some of the existing preconditions may be dropped.

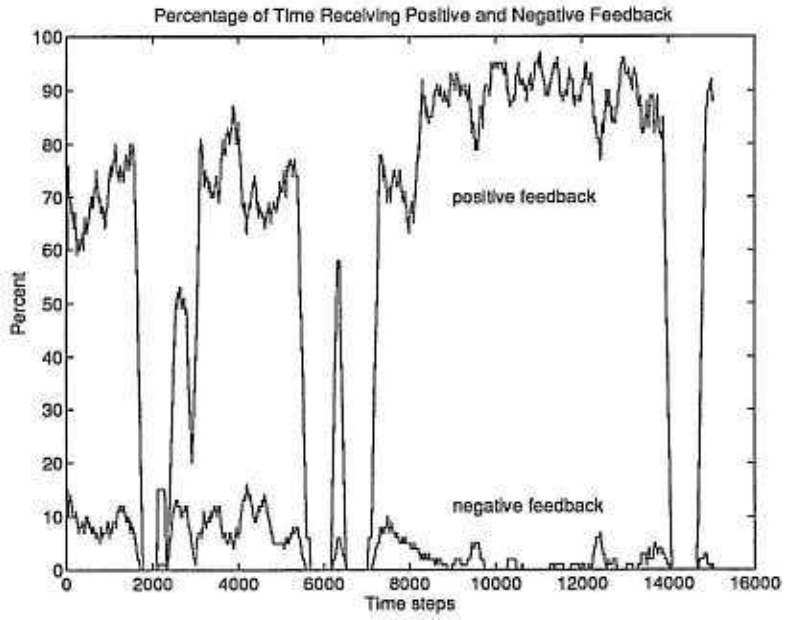


Figure 6–10: Graph of Feedback *v.* Time for Trial RA2

Version B — Global Move-Leg-Back Reflex Several trials were conducted with Version B, again with some using the staggered starting position, and others using the unstaggered position. The reliability target was also varied between trials.

With most trials, Smudge often got stuck in a situation where three legs were immobile at their extreme rearward position, and the fourth was swinging backwards and forwards through a small angle to drag the robot along and provide a positive feedback signal (see Figure 6–7(b)). In this situation, no negative feedback was received, so that none of the behaviours monitored new conditions to modify this behaviour.

However, on trial RB2, which had an unstaggered initial leg position and a reliability target of 0.95, much better performance was observed. During the first 6000 or so time steps, Smudge was behaving in much the same manner as had been observed in the other trials of this batch. However, by 6500 time steps, it had learned the preconditions shown in Table 6–8, and was receiving positive feedback 100% of the time. The graph of feedback percentages is shown in Figure 6–11. By 7800 time steps, some of the preconditions were dropped, but they were soon relearned. However, at 8900 time steps, Smudge got stuck in a position where all

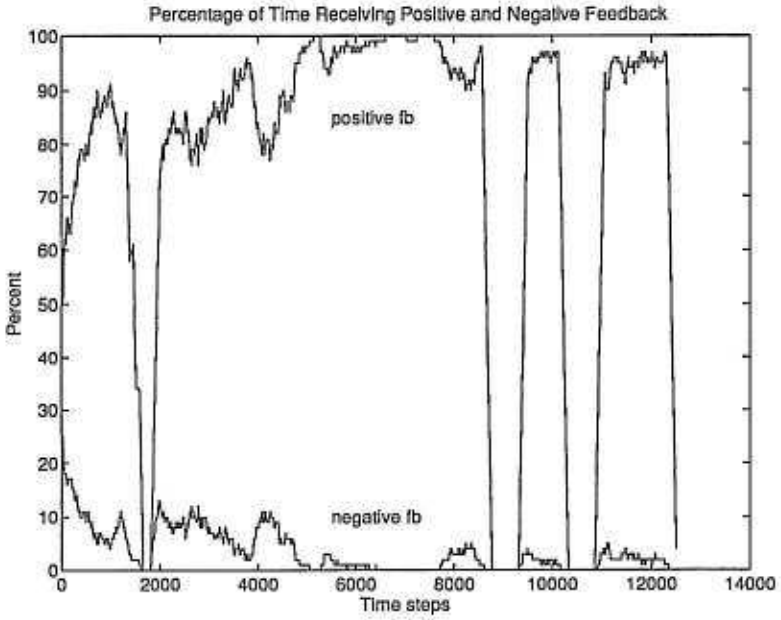


Figure 6–11: Graph of Feedback *v.* Time for Trial RB2

Behaviour	*Precondition*
Swing leg 0 forward	L3F T
Swing leg 1 forward	L2F T, L3F F
Swing leg 2 forward	L0F T, L1F F
Swing leg 3 forward	L1F F, L1B F

Table 6–8: Preconditions learned by Version B, trial RB2

four legs were on the ground at their rearward extreme of movement, and none of the behaviours' precondition lists were fulfilled (see Figure 6–7(b)). At this stage, the statistics for the behaviours decayed away until their reliability values were below threshold. Once this occurred, some of the learned preconditions were dropped, and the walking performance deteriorated.

Nonetheless, this was a significant trial, as it was the first time that the algorithm, starting with no preprogrammed preconditions, had managed to reach a stage where the balance between positive and negative feedback was 100% to 0% for a sustained length of time.

Version C — Learnable Move-Leg-Back Behaviours The first trial with Version C was with a staggered initial leg position and a reliability target of 0.75. After about 1100 time steps, Smudge had not learned any preconditions, but was walking such that the proportion of time receiving positive feedback was around 95%, and that receiving negative feedback was around 3%. The behaviours occasionally monitored conditions, but none was adopted. In other words, an acceptable solution to the problem had, in effect, already been programmed into the algorithm!

When started with the unstaggered leg position, the algorithm adopted some preconditions, but failed to find a stable state where the percentage of time receiving negative feedback remained below 5%. The performance was virtually unchanged when the reliability target was raised to 0.95.

A Combination

The trials which produced the best performance on each of the three versions run with the 'only lift one leg at a time' rule were rerun with this rule being combined with not returning the legs to their initial positions after a fall.

Version A — Global Horizontal Balance Reflex This trial used the unstaggered initial leg position and a reliability target of 0.95. During the run, Smudge got stuck with all four legs on the ground fairly frequently (Figure 6–7(a)), and in between these phases reached a balance between positive and negative feedback of up to 70% to 10%. As seen with the previous trials where the legs were not returned to their initial positions after a fall, the robot often got into situations where its legs were just moving back through a small angle before swinging to their extreme forward positions. This behaviour was such that the proportion of time receiving negative feedback was not great enough to bring the reliability values of the behaviours below threshold.

Thus, the combination approach did not perform as well as just having the 'only lift one leg at a time' rule for Version A.

Behaviour	*Precondition*	*Behaviour*	*Precondition*
Swing leg 0 forward	L3F T	Move leg 0 back	—
Swing leg 1 forward	L2F T	Move leg 1 back	—
Swing leg 2 forward	—	Move leg 2 back	—
Swing leg 3 forward	—	Move leg 3 back	L0B F

Table 6–9: Preconditions learned by Version C, trial CC1

Version B — Global Move-Leg-Back Reflex Version B was run from the same initial position and parameter values as was Version A. The results were very similar to those of Version A — the combination approach was not as good as just having the 'only lift one leg at a time' rule for Version B either.

Version C — Learnable Move-Leg-Back Behaviours As with the trials using the other versions of the algorithm, Version C was run with the unstaggered initial leg position, and a reliability target of 0.95 (trial CC1)[8]. During the first 7000 time steps, the run progressed in much the same manner as did those for the other versions. However, Smudge was less inclined to get stuck in the situations where these had done, because, with no hard-wired reflexes, such cases were less likely to persist. At 7000 time steps, the performance was not spectacular (roughly 75% positive feedback and 5% negative feedback), but after a further 200 time steps it had improved dramatically - in fact, to a point where positive feedback was being received 100% of the time. The preconditions learned at this stage are shown in Figure 6–9. This state was stable in that the behaviours only monitored other conditions very occasionally, and none was adopted as a new precondition. Smudge fell over on a few occasions between the time when these preconditions were adopted and the end of the run (11000 time steps), but this did not result in a change of preconditions for any of the behaviours. The plot of feedback percentages for this trial is shown in Figure 6–12.

[8]The first 'C' in the trial name denotes trials where a combination of the 'only lift one leg at a time' rule and the strategy of not returning legs to their initial positions after a fall was used.

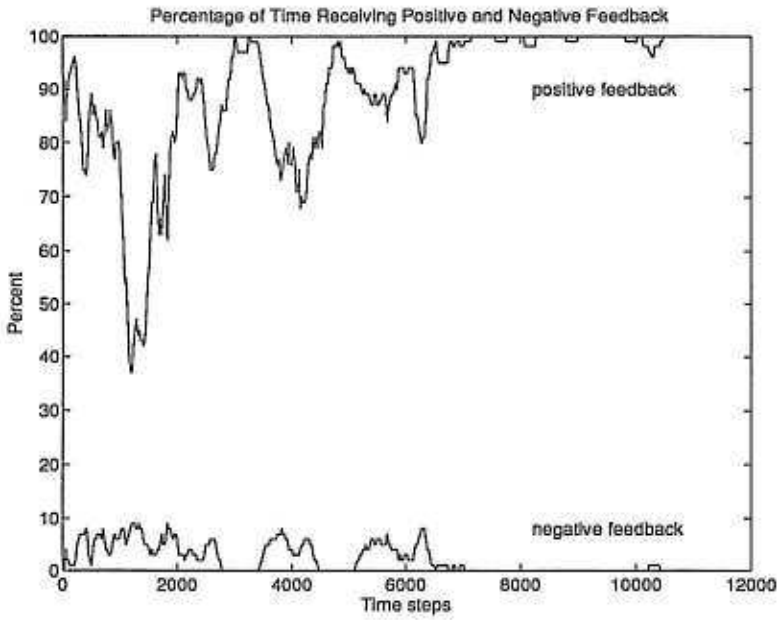


Figure 6–12: Graph of Feedback *v.* Time for Trial CC1

6.2.4 The Effect of Noise

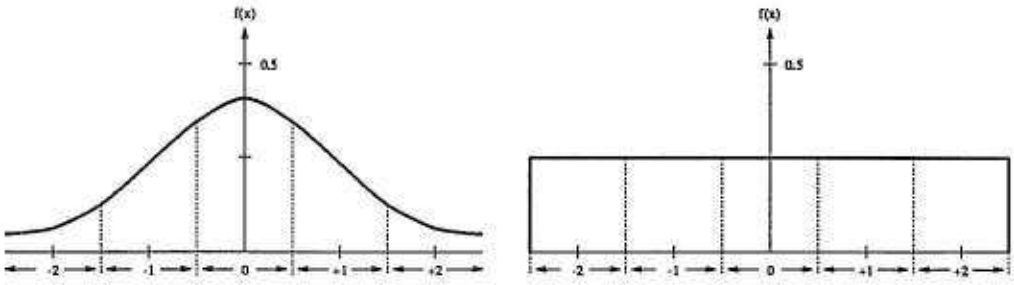
In order to test the robustness of the algorithm in a situation which more closely resembled 'real life', the trials of each version which had produced the best performance up to this point were tested under noisy conditions.

Specifically, each time that a leg was moved in the simulation by a change in its alpha angle, a degree of noise was injected into the position to where the leg was actually moved. Two batches of tests were run, using two different types of noise.

The first batch used the Normal distribution to add noise from the integer set $[-2, -1, 0, +1, +2]$ to the desired destination angle of each leg. The distribution is shown in Figure 6–13(a). The shape of this graph was derived from the Gaussian equation:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

where μ is the mean noise component, set to 0, and
σ is the standard deviation, set to 1.



(a) Normal Distribution (b) Even Distribution

Figure 6–13: Noise Distributions Used to Test the Robustness of Learning

The probability, $p(x)$, of an observation lying outside the range $-x$ to $+x$ may be derived by integrating this formula, and values for $p(x)$ are listed in statistical tables.

In the simulation, the probabilities of the noise being -2, -1, 0, $+1$ and $+2$, were, respectively, $\frac{1}{2}p(1.5)$, $\frac{1}{2}\{p(0.5) - p(1.5)\}$, $\{1 - p(0.5)\}$, $\frac{1}{2}\{p(0.5) - p(1.5)\}$, and $\frac{1}{2}p(1.5)$.

In the second batch of trials, the noise was evenly distributed among the same set of values; $[-2, -1, 0, +1, +2]$ (see Figure 6–13(b)).

Although the accuracy of the servos was not tested on the real machine, it is probable that the normally distributed noise reflects the real situation more closely than does the evenly distributed noise.

Version A — Global Horizontal Balance Reflex

The 'only lift one leg at a time' rule was used with the unstaggered initial leg position and a reliability target of 0.95 (as in trial RA2).

With normally distributed noise, the performance of the algorithm was quite drastically impaired compared to the noise-free trial; Smudge got stuck in situations where it could not move any of its legs much more frequently when noise was added.

With evenly distributed noise, the impairment was not so marked; the balance between time receiving positive feedback and negative feedback reached 90% to

1% at one stage, but this did not last for long and the robot was still more likely to get stuck with all of its legs immobile (because no behaviours were eligible to become active) than in the noise-free case.

These trials were repeated several times for both types of noise, but all resulted in similar performance.

The observation that evenly distributed noise led to a lesser impairment than normally distributed noise was unexpected, as the former type produces a higher proportion of non-zero noise values. The data produced from the runs was examined, and it was found that, due to a peculiarity with the pseudo-random number generator used in the algorithm code to produce the noise[9], exactly the *same* noise was generated each time the algorithm was run under a particular condition, so that the observed performance was not just similar between runs, but *identical*. The bad performance of the algorithm with normally distributed noise compared to evenly distributed noise might therefore have been due to the particular set of random numbers which had been generated. Indeed, on investigation of the noise generated during the trial, it was found that negative-valued noise was produced almost twice as often as positive-valued noise in the first 200 time steps of the run.

The problem with the pseudo-random number generator was fixed by setting the generator to a different seed on each run. The trials were repeated, but the performance with normally distributed noise was still observed to be worse than with evenly distributed noise. This therefore suggested that the result was due to a more fundamental difference between the two cases. This observation is discussed in more detail in Section 7.2.

Version B — Global Move-Leg-Back Reflex

Again, the 'only lift one leg at a time' rule was used, together with the unstaggered initial leg position and a reliability target of 0.95 (as in trial RB2).

---

[9]It was found that the generator was reset to the same seed every time the program was run, so that it always generated the same sequence of pseudo-random numbers.

Behaviour	*Precondition*	*Behaviour*	*Precondition*
Swing leg 0 forward	—	Move leg 0 back	L2U T
Swing leg 1 forward	L1B T	Move leg 1 back	—
Swing leg 2 forward	L0F T	Move leg 2 back	—
Swing leg 3 forward	—	Move leg 3 back	—

Table 6–10: Preconditions learned by Version C, trial NCC1

Several runs were tried using normally distributed noise, and all resulted in Smudge getting stuck more regularly than it did without noise.

With evenly distributed noise, the behaviours were able to learn good preconditions from time to time, but the noise always forced the behaviours to monitor new conditions and drop some of the existing preconditions.

The overall performance of the algorithm was fairly similar for both types of noise on most occasions, but trials with evenly distributed noise sometimes performed slightly worse.

Version C — Learnable Move-Leg-Back Behaviours

A combination of the 'only lift one leg at a time' rule and not returning the legs to their initial positions after a fall, with reliability target 0.95 was used (as in trial CC1).

Several runs were tried using normally distributed noise (trials NCC1)[10]. In some of these, Smudge was actually receiving positive feedback 100% of the time by 2500 time steps (this is *quicker* than was achieved without noise). The preconditions learned at this stage are shown in Table 6–10. However, after another 300 time steps Smudge got stuck with all four legs immobile (Figure 6–7(c)). As usual, this situation was only resolved when the behaviours started monitoring conditions and dropped some existing preconditions. After this point, good (but not perfect) performance re-emerged on several occasions up to the end of the trial (at 10000 time steps), but these periods were mixed with periods when the legs

[10]'N' denotes trials where noise was added to the leg movements

became immobile for a time. Hence, although perfect performance was achieved quicker on these trials than on the noise-free trial, it only lasted for a brief period, after which the performance was generally worse than it was without noise.

On trials with evenly distributed noise, the algorithm fared worse, generally, than either the noise-free or the normally distributed noise trials; Smudge got stuck on many occasions, and perfect performance was not achieved at any point in the 10000 time step runs.

Chapter 7

Summary and Discussion

The previous chapter described in some detail the results obtained from trials of the algorithm conducted in simulation. These results are now summarised, some comments are made about them, and it is then attempted to put the whole project into perspective regarding possible future work with the algorithm and other research with quadruped machines.

7.1 The Original Goals of the Learning Algorithm

Put simply, the original goal was to enable Smudge to learn how to walk. This entails learning how to coordinate the movement of its legs so that the robot moves forward without falling over. In terms of the mechanics of the algorithm, it means that a suitable set of preconditions must be found for each behaviour to enable the robot to exhibit this competence. As long as the environmental conditions remain constant, so should the preconditions — no existing preconditions should be dropped, and no new ones should be adopted. In this state, the robot should never fall over, i.e. the percentage of time in which negative feedback is received should always be 0%.

7.2 The Results Obtained

Using the algorithm described in [Maes & Brooks 90], with the slight modifications listed in Section 2.5, the observed performance was very poor. If the global horizontal balance reflex was replaced by a move-leg-back reflex, then the performance of the algorithm was still bad. With the reflexes replaced by learnable move-leg-back behaviours, then the performance improved dramatically (see, for example, Figure 6–6). However, although the robot could learn a near-optimal solution in this case, on none of the trials was negative feedback actually eliminated altogether.

Attempts were made to preprogram suitable sets of preconditions into the behaviours for each version of the algorithm in order to demonstrate that the goal was achievable under such a learning mechanism. A set was found with which the algorithm equipped with the global horizontal balance reflex could display perfect performance, but this goal was not realised with either the global move-leg-back reflex or the learnable move-leg-back behaviours. Particular difficulty was encountered when trying to find a set of preconditions for the version with learnable move-leg-back behaviours. In this case, with eight behaviours rather than the four of the other two cases, the problem of trying to specify sufficient preconditions to produce a reliable walking pattern, but not so many that Smudge got caught in situations where none of the behaviours' precondition lists was fulfilled, was much more difficult.

Two extensions were added to the basic algorithm, in order to promote the search of more of the problem space by the algorithm:

1. *Not Returning Legs to their Initial Position after a Fall.* The trials using this extension were generally not very successful, for reasons explained in Section 6.2.3.

2. *Including an 'Only Lift One Leg at a Time' Rule.* This led to an improvement in performance during some periods of most trials, but also to a situation where Smudge was more likely to get stuck in positions where all four legs

were immobile on the ground and none of the behaviours' preconditions lists was fulfilled.

A combination of these two extensions was also tested. This did not result in good performance from trials using either of the reflex leg movements, but with learnable move-leg-back behaviours, very good performance, sustained over a long period, was observed (see Figure 6–12). Even in this trial, however, having learned a suitable set of preconditions, Smudge still fell over occasionally.

Some trials were run in which noise was added to the leg movements in the simulation, in order to model reality a little more closely. It was found that noise generally led to an overall decrease in the performance of the robot, and/or to a decrease in the duration for which good sets of preconditions survived.

For trials where noise was added to the algorithm using the global horizontal balance reflex, the surprising result that evenly distributed noise produced a *smaller* deterioration in performance than did normally distributed noise was observed.

The trials were carefully analysed to see why this was the case. As mentioned in the previous chapter, when the algorithm was using the global horizontal balance reflex it was inclined to get stuck in positions such as the one illustrated in Figure 6–7(a). In a noise-free environment, such situations could only be resolved by one of the behaviours dropping a precondition so that a leg could swing forward again. Now, it sometimes happened that the actual total of horizontal angles in such a deadlock did not add to exactly zero, because the legs were only moved by integer angles. In such a situation, the horizontal balance reflex may be sending a correction signal of less than 1° to the legs, but they would not move because of the integer constraint. However, if noise is added to this correction signal, then there is a chance that the leg will move even if the correction is smaller than 1°. As soon as any leg is moved, the horizontal angle sum is no longer zero and the deadlock situation is resolved. With evenly distributed noise there is a higher probability of having more extreme noise values than with normally distributed noise, so the chance of the deadlock situation being resolved is greater. Hence, trials with evenly distributed noise were less likely to get stuck for long periods, so their overall performance was better than those with normally distributed noise.

The types of situation in which the algorithm equipped with either the move-leg-back reflex or the learnable move-leg-back behaviours were likely to get stuck (see Figure 6–7(b) & (c)) could not be resolved by small, random movements of the legs, so that trials with evenly distributed noise did not perform better than those with normally distributed noise in these cases.

7.3 Comparison of Preconditions Learned by Best Trials

Tables 6–1, 6–3, 6–7, 6–8, 6–9 and 6–10 show the preconditions learned in phases of very good, stable performance in different trials. A comparison of these reveals that there was no universal agreement as to what constituted a good set of preconditions; the precondition L2F T for the swing-leg-1-forward behaviour was present in all the examples, and L3F T was a precondition of the swing-leg-0-forward behaviour in four of the six, but there was little agreement elsewhere. This was not particularly surprising, as a particular behaviour could chose to coordinate its activity relative to any of the three legs it did not control.

In trials incorporating the learnable move-leg-back behaviours, the algorithm only learned preconditions relating to either the swing-leg-forward behaviour, or the move-leg-back behaviour for any one leg, but not both. This, again, was not surprising, because one behaviour or the other should be active throughout the gait cycle, and the algorithm prevented both from being active at any one time. Thus, as long as one behaviour was coordinated with the movement of the other legs, then it was sufficient that the other should become active whenever the explicitly-coordinated behaviour was inactive.

Slightly more interesting was the fact that in trials C12, CC1 and NCC1 (Tables 6–3, 6–9 and 6–10 respectively), good performance was achieved despite the fact that in each case one leg had learned *no* preconditions relating to either its swing-leg-forward or move-leg-back behaviours. In the latter pair of trials, this was probably because the 'only lift one leg at a time' rule was incorporated, so that, as long as three legs were explicitly coordinated, the fourth would only be eligible to be lifted at a suitable time. The rule was not incorporated in trial C12,

however, which explains why the proportion of time receiving negative feedback never dropped to 0% in this case.

7.4 Some Reasons for Failure to Achieve Perfect Performance

From the results of the trials that were conducted, it is apparent that there are some reasons for the failure to achieve perfect performance which are specific to each of the three versions of the algorithm, and some which are more generally related to the fact that a four-legged robot was used in place of one with six legs.

With the global horizontal balance reflex (as used by Maes and Brooks with Ghengis), for example, it was seen that Smudge was liable to get stuck in deadlock positions, where the horizontal angles of the legs summed to zero and no behaviour was eligible to become active (Figure 6–7(a)). This situation was more likely to occur on Smudge than on Ghengis, because Smudge had four legs and 12 perceptual conditions (an 'up', 'forward' and 'backward' condition for each leg) compared to Ghengis' six legs and six perceptual conditions (an 'up' condition for each leg) — with fewer legs and the possibility of each behaviour learning more preconditions, it was more likely that none of the behaviours would have a fulfilled precondition list at a given point in the run.

With learnable move-leg-back behaviours, legs which were on the ground were not always being pushed backwards, because the move-leg-back behaviours were not always active. In this version of the algorithm, Smudge was therefore much more likely to drag legs along the ground than in the other versions. As long as some of the legs are moving, positive feedback is received and the dragging is not penalised, so Smudge does not try to find a better gait (Figure 6–7(c)).

There are several points which are relevant to all versions of the algorithm. One is that, on several trials, Smudge got into situations where all four legs were on the ground being moved backwards until a point where the robot became unstable. The legs were then returned to their initial position, and the walking pattern would repeat. The problem was that negative feedback was only received at one point

during the sequence, so that, if the reliability target was less than 1.0 then such a motion would persist. The problem with setting the reliability target to 1.0 is that it would then be extremely unlikely that *any* set of preconditions will persist — even if Smudge never fell over it still may not receive *positive* feedback at every time step, so the reliability of some behaviours would be less than 1.0.

It was also observed that in trials where a staggered initial leg position was used, the performance of the algorithm was often worse than with an unstaggered initial position. This may seem surprising at first glance, because the former position was actually a position taken from the goal gait, whereas the latter was not. However, the effect of the staggered position was to bias the learning process, making some leg positions more likely to be explored than others. For example, leg 3 started near its extreme forward position in the staggered case, so that it was relatively rare for it to be observed in a rearward position because the robot had often fallen over before leg 3 had a chance to move there, especially during the initial stages of learning.

The question of stability also had a much more general influence on the performance of the algorithm in all of the trials. As a four-legged robot can only lift one leg at a time while remaining stable, whereas a six-legged robot can lift up to three at a time, the former is much less likely to be able to adequately explore the full range of perceptual conditions during a learning run, especially the 'leg forward' and 'leg backward' conditions when the relevant thresholds for triggering these conditions are high. An attempt to improve the overall stability of the robot by including an 'only lift one leg at a time' rule in the algorithm resulted in improved performance in most situations.

7.5 Further Experiments with the Algorithm

The results reported in the previous chapter suggest a number of additional experiments which may be conducted. These include

- *Testing whether Additional Perceptual Conditions were Necessary.* The original assumption that it would be necessary to include 'leg forward' and 'leg backward' conditions could be tested by removing such conditions from a

algorithm; using a staggered initial leg position required prior knowledge of the solution, using the 'only lift one leg at a time' rule introduced an extra non-local component to the algorithm, and with the number of parameters and thresholds which could be varied, it often felt as if the experimenter was doing more work than the learning algorithm to find a solution.

An additional point is that all of these experiments were run in the simplest environment possible — a (simulated) smooth, level floor, but the real world is full of obstacles and inclines. Even if the algorithm could be made to perform perfectly in the simple case, it is hard to imagine it being able to cope with more complicated environments.

7.7 Possible Extensions

There a many ways in which the algorithm could be extended in an attempt to improve its effectiveness when used with a four-legged robot. For example,

- *Penalising Dragging of Legs.* This applies particularly to the versions incorporating the move-leg-back reflex and the learnable move-leg-back behaviours. A penalty might be imposed, in the form of negative feedback, if any leg is stationary for a certain length of time, or if a leg is on the ground but not moving when positive feedback is being received.

- *Giving a Larger Weight to the Negative Feedback Signal.* This would help in situations where the robot was able to settle into an unsatisfactory walking pattern because negative feedback was only being received for a brief period of the gait. Note that this would probably not be necessary on the real robot, as it would take a longer time to recover after each fall, so more negative feedback would be received anyway. This suggests another solution to the problem on the simulation — make the robot stay 'on the floor' (receiving negative feedback) for a longer duration each time it falls over.

- *Adding more Sensors to the Robot.* The physical version of Smudge is equipped with force-sensing circuits to enable monitoring of the load on each

of the servos. There is also a free pin on one of the microcontrollers, which may be used for another sensor, such as a tilt sensor. The former could be used to tell Smudge when one of the legs was pressing against an obstacle, and the latter to provide advanced warning of when a fall was likely to occur. However, in order to incorporate such features into the system, some fairly extensive modifications to the existing algorithm would be required.

7.8 Relating Results to Other Research

Much work has been devoted to the control of quadruped walking machines by Shigeo Hirose and colleagues at the Tokyo Institute of Technology. In [Hirose 84], a 'classical' control system is described which provides a quadruped with a robust, statically stable walking pattern. The robot used has joints at the knees as well as at the hips, and is able to negotiate obstacles and stairs very effectively. In other words, this system was far superior to anything achieved in the present study.

[Brooks 91a] describes work on a successor to Ghengis, called Attila. This robot has six legs, each with three degrees of freedom, an active whisker, a gyro stabilised pan-tilt head carrying a range finder and a CCD camera, and over 150 sensors. Obviously, such a machine will require a control architecture of vastly greater complexity than that of Ghengis — the paper describes some of the problems and challenges facing the designer of such a system.

As we come to expect greater performance from our robots, it is likely that statically stable gaits will not be sufficient to provide fast enough locomotion in many cases, particularly with quadrupeds. Indeed, it is not clear that any natural quadruped employs a statically stable gait (see [Raibert 86] pp 89–92). In the same text (pp 95–106), Raibert describes preliminary experiments with a quadruped hopping machine. Although the reported system had many limitations, it suggests that the implementation of a robust and practical dynamically stable quadruped is feasible.

However, Raibert and colleagues' work involves classical systems of control, whereas one of the aims of the present project was to investigate the applicability of behaviour-based architectures to the control of quadruped walking. Within

the past few years it has become popular to model robot control architectures on the neural circuitry of animals (see, for example, [Beer & Chiel 91], [Ekeberg 93] and [Cliff 93]). An investigation into the different types of coordination systems employed by animals for walking is described in [Cruse 91]. This research is useful not only for building artificial systems which replicate such control mechanisms, but also for comparing the solutions found in nature to those found in artificial systems that have been developed independently.

A proposal to build a neural network based control system for the control of legged locomotion, inspired by natural legged systems, is described in [Wadden *et al* 93]. It will be of great interest to see the results of such an endeavour.

Meanwhile, research is continuing apace in the development of classical control architectures, and also of hybrid systems which incorporate some elements of both approaches. It is likely that classical systems will continue to out-perform behaviour-based ones for some years to come. However, there will come a stage where the required competences of a system will simply be too complicated to be explicitly programmed. When such an impasse is reached, behaviour-based systems will hopefully have progressed far enough to be able to learn increasingly more complicated competences by themselves. Through the study of control architectures which can develop and modify themselves, the possibility of gaining insight into natural control systems also arises, and vice versa.

Bibliography

[Beer & Chiel 91] R.D. Beer and H.J. Chiel. The neural basis of behavioural choice in an artificial insect. In Meyer and Wilson, editors, *From Animals to Animats. Proceedings of the First International Conference on the Simulation of Adaptive Behavior*, pages 247–253. MIT Press, Cambridge, MA, 1991.

[Brooks 89] R.A. Brooks. A robot that walks: Emergent behaviors from a carefully evolved network. AI memo 1091, MIT Artificial Intelligence Laboratory, 1989.

[Brooks 91a] R.A. Brooks. Challenges for complete creature architecture. In Meyer and Wilson, editors, *From Animals to Animats. Proceedings of the First International Conference on the Simulation of Adaptive Behavior*, pages 434–443. MIT Press, Cambridge, MA, 1991.

[Brooks 91b] R.A. Brooks. Intelligence without reason. AI memo 1293, MIT Artificial Intelligence Laboratory, 1991.

[Cliff 93] D. Cliff. Animate vision in an animat fly: A study of computational neuroethology. In *International Workshop on Mechatronical Computer Systems for Perception and Action*, pages 11–18. Högskolan Halmstad, 1993.

[Cruse 91] H. Cruse. Coordination of leg movements in walking animals. In Meyer and Wilson, editors, *From Animals to Animats. Proceedings of the First International Conference on*

the Simulation of Adaptive Behavior, pages 105–119. MIT Press, Cambridge, MA, 1991.

[Ekeberg 93] O. Ekeberg. Neural control of vertebrate locomotion. a computer simulation study. In *International Workshop on Mechatronical Computer Systems for Perception and Action*, pages 375–378. Högskolan Halmstad, 1993.

[Hirose 84] S. Hirose. A study of design and control of a quadruped walking vehicle. *International Journal of Robotics Research*, 3:113–133, 1984.

[Maes & Brooks 90] P. Maes and R.A. Brooks. Learning to coordinate behaviors. *Proceedings of the AAAI*, 2:796–802, 1990.

[Raibert 86] M.H. Raibert. *Legged Robots That Balance.* MIT Press, Cambridge, MA, 1986.

[Song & Waldron 89] S-M. Song and K. J. Waldron. *Machines That Walk: The Adaptive Suspension Vehicle.* MIT Press, Cambridge, MA, 1989.

[Todd 85] D.J. Todd. *Walking Machines. An Introduction to Legged Robots.* Kogan Page, London, 1985.

[Wadden *et al* 93] T. Wadden, O. Ekeberg, and A. Lansner. Towards ann based control of simulated legged locomotion. In *International Workshop on Mechatronical Computer Systems for Perception and Action*, pages 379–382. Högskolan Halmstad, 1993.

Appendix A

Final Specification of Smudge

Body length:	248mm
Body width:	69mm
Leg length:	150mm
Weight:	550g
Servo type 1:	Futaba FP–S143
Servo type 2:	Futaba S5101
Microcontrollers:	Microchip Technology Inc. PIC 16C71
Power supply:	5.0V D.C. external supply

Appendix B

Program Code for the Learning Algorithm

```
/*-------------------------------------------------------------------------------------------------------------
   ALGORITHM    The learning algorithm.
-------------------------------------------------------------------------------------------------------------*/

#include "top.h"

void    initialise_leg_positions(void);
void    initialise_learning(void);
void    update_condition_flags(void);
char    select_a_behaviour_for_group(char, char, char []);
void    continue_active_behaviours(void);
void    global_horizontal_balance(int []);
void    get_advance_distance(int [], float []);
void    update_stats(char, bool, bool);
void    monitor_next_condition(char);
void    continue_monitoring(char);
void    start_behaviour(char);
void    drop_condition(char);
void    adopt_condition(char, bool);
void    next_time_step(float [], float [], bool *);
void    update_output_file(void);
void    clear_monitoring_stats(char);
bool    condition_set(char, char);
bool    reliable(char);
bool    selectable(char);
float   relevance(char);
float   reliability(char);
float   interestingness(char);
float   corr(char, float [][][]);
float   get_int_answer(bool, float, float);
float   relative_pos(int, int, float);
char    get_behaviour_number(char, char);

int     plus_noise(int);

/* definitions of global parameters used by the algorithm */

#define number_of_behaviours    (char)  4
#define number_of_groups        (char)  4
#define forward_threshold       (int)   15
#define backward_threshold      (int)   -15
#define leg_forward_extent      (int)   45
#define swing_angle             (int)   10
#define horiz_balance_amount    (float) 1.00

#define correlation_threshold   (float) 0.65
#define monitor_duration        (float) 65
#define reliability_target      (float) 0.95
#define stats_initial_value     (float) 10
#define stats_decay_rate        (float) 0.99

/* The following definition is that given in 'A Book on C' and agrees with calling rand() many times to get max */

#define RAND_MAX                2147483646

/* definitions of data structures used by the algorithm */

bool    leg_n_up[4], leg_n_forward[4], leg_n_backward[4];
                                                        /* The perceptual conditions:-      */
                                                        /* throughout the program, these    */
                                                        /* conditions are referred to by    */
                                                        /* a 2D array, where                */
                                                        /* element 0: condition (0=leg up,  */
                                                        /*      1=leg forward, 2=leg back)  */
                                                        /* element 1: leg number (0-3)      */

char    order_of_conditions[13][2];                     /* specifies the order in which     */
                                                        /* conds are evaluated by behav's   */
                                                        /* [a][b]:                          */
                                                        /* a = order of conditions          */
                                                        /* b = (a)th condition (or [-,99]   */
                                                        /*      to mark end of list)        */

char    precondition[4][13][3];                         /* [a][b][c]:                       */
                                                        /* a = behaviour number             */
                                                        /* b = precondition number          */
                                                        /* c = condition, with element 3    */
                                                        /*      indicating whether cond     */
                                                        /*      should be true or false,    */
                                                        /*      (or [-,99,-] to mark end of */
                                                        /*       precond list)              */

bool    behaviour_active[4];
bool    behaviour_monitoring[4];

float   pos_stats[4][2][2], neg_stats[4][2][2];         /* [a][b][c]:                       */
```

```
                                                                    /* a = behaviour number          */
                                                                    /* b = rows of stat table        */
                                                                    /* c = columns of stat table     */

char    behaviour_n_monitoring_precond[4];                          /* indexed thru order_of_conds[][] */
float   behav_non_prec_pos_stats[4][2][2];
float   behav_non_prec_neg_stats[4][2][2];
int     monitoring_clock[4];

int     time_step_counter;

char    number_of_behaviours_in_group[number_of_groups];

/*-------------------------------------------------------------------------------------------------------------------
   initialise_leg_positions     Called only once, at the beginning of the simulation
   ------------------------
-------------------------------------------------------------------------------------------------------------------*/
void initialise_leg_positions(void)
{
 char i;

 for (i = 0; i < 4; i++) {
   alpha[i] = 0;
   beta[i] = LEG_DOWN;
 }

}

/*-------------------------------------------------------------------------------------------------------------------
   initialise_learning  Initialise variables used by the learning algorithm
   -------------------
-------------------------------------------------------------------------------------------------------------------*/
void initialise_learning(void)
{
  char i, j, k;

  for (i = 0; i < 12; i++) {                                        /* o_o_c[n][0]=cond, [n][1]=leg no */
    order_of_conditions[i][0] = i % 3;                              /* so order is:                    */
    order_of_conditions[i][1] = i / 3;                              /*    10u, 10f, 10b, 11u, ...      */
  }                                                                 /*    12u, ..., 13u, 13f, 13b      */
  order_of_conditions[12][1] = 99;                                  /* <- to mark end of the list      */

  for (i = 0; i < number_of_behaviours; i++) {                      /* initial precondition list for   */
    precondition[i][0][0] = 0;                                      /* each behaviour just contains    */
    precondition[i][0][1] = i;                                      /* the precondition that the       */
    precondition[i][0][2] = false;                                  /* corresponding leg is not up     */
    precondition[i][1][1] = 99;
    behaviour_active[i] = false;
    behaviour_monitoring[i] = false;
    behaviour_n_monitoring_precond[i] = (11 + i * 3) % 12;
    for (j = 0; j < 2; j++)

      for (k = 0; k < 2; k++) {
        pos_stats[i][j][k] = stats_initial_value;
        neg_stats[i][j][k] = stats_initial_value;
      }
    clear_monitoring_stats(i);
  }

  for (i = 0; i < number_of_groups; i++)
    number_of_behaviours_in_group[i] = 1;

}

/*-------------------------------------------------------------------------------------------------------------------
   next_time_step       Function returns values indicating how far Smudge has advanced along the X and Y axes, plus
   --------------       information about Smudge's stability in the new position (i.e. a) how far the CoM is from the
                        boundary of support defined by the legs currently on the ground, and b) which two legs on the
                        ground define the section of this boundary which is closest to the CoM). The global alpha and
                        beta values are also updated directly.
-------------------------------------------------------------------------------------------------------------------*/
void next_time_step(float advance_values[2], float stable_values[3], bool *previous_neg_fb)
{
  char  i, j, k;
  bool  positive_fb, negative_fb;
  int   advance_angles[4];
  char  count, to_start[4];

  time_step_counter++;
  update_condition_flags();

  if (*previous_neg_fb) {                                           /* If Smudge fell over last time,  */
    initialise_leg_positions();                                     /* return legs to starting position, */
    for (i = 0; i < number_of_behaviours; i++)                      /* and stop all currently active   */
      behaviour_active[i] = false;                                  /* behaviours.                     */

    get_positions();
    advance_values[0] = advance_values[1] = 0.0;
  }
  else {
    count = 0;
    for (i = 0; i < number_of_groups; i++)
      count = select_a_behaviour_for_group(i, count, to_start);
    if (count > 0)
      start_behaviour(to_start[(int) rand() % count]);
    continue_active_behaviours();
    global_horizontal_balance(advance_angles);

    get_positions();
    get_advance_distance(advance_angles, advance_values);

  get_stability_info(stable_values);
```

```
  negative_fb = (stable_values[0] < -10.0);
  positive_fb = (advance_values[1] > 0.0 && !negative_fb);

  for (i = 0; i < number_of_behaviours; i++)
    update_state(i, positive_fb, negative_fb);                    /* including monitoring stats & clock */

  for (i = 0; i < number_of_behaviours; i++) {
    if (!behaviour_monitoring[i] && !reliable(i))
      monitor_next_condition(i);
    if (behaviour_monitoring[i])
      continue_monitoring(i);
  }

  for (i = 0; i < 4; i++) {                                       /* transfer data to global variables */
    monitor_clock_data[i] = (behaviour_monitoring[i]) ? monitoring_clock[i] : 0; /* to be displayed by simulation  */
    for (j = 0; j < 2; j++) {
      monitor_data[i][j] = (behaviour_monitoring[i]) ? order_of_conditions[behaviour_n_monitoring_precond[i][j]] : 99;
    }
    j = 0;
    do {
      for (k = 0; k < 3; k++) {
        precond_data[i][j][k] = precondition[i][j][k];
      }
    } while (precondition[i][j++][1] != 99);
  }
  pos_fb_array[time_step_counter % FB_STAT_SLICE] = (positive_fb) ? 1 : 0;
  neg_fb_array[time_step_counter % FB_STAT_SLICE] = (negative_fb) ? 1 : 0;

  *previous_neg_fb = (bool) negative_fb;                          /* so Smudge knows whether he should  */
                                                                  /* reinitialise stance at next timestep */
  if (time_step_counter % 10 == 0)
    update_output_file();                                         /* output statistics to file */

}

/*-------------------------------------------------------------------------------------------------------------------
   update_condition_flags      Called at the beginning of the control loop
   ----------------------
-------------------------------------------------------------------------------------------------------------------*/
void update_condition_flags(void)
{
  char  i;
  int   sign;

  for (i = 0; i < 4; i++) {
    sign = (i > 1) ? -1 : 1;
    leg_n_up[i] = (beta[i] == LEG_UP);
    leg_n_forward[i] = (sign * alpha[i] > forward_threshold);
    leg_n_backward[i] = (sign * alpha[i] < backward_threshold);
  }

}


/*-------------------------------------------------------------------------------------------------------------------
   condition_set          Arguments:
   -------------                 n = condition number
                                 m = leg number
                          Returns the current boolean value of perceptual condition n for leg m.
-------------------------------------------------------------------------------------------------------------------*/
bool condition_set(char cond, char leg)
{
  switch (cond) {
  case 0:
    return leg_n_up[leg];
  case 1:
    return leg_n_forward[leg];
  case 2:
    return leg_n_backward[leg];
  default:
    printf("condition_set: Invalid condition number");
    return false;
  }
}

/*-------------------------------------------------------------------------------------------------------------------
   relevance     Function returns the relevance of the behaviour indicated by its argument. The relevance is a
   ---------     floating point number between -2 and +2.
-------------------------------------------------------------------------------------------------------------------*/
float relevance(char n)
{
  return corr(n, pos_stats) - corr(n, neg_stats);
}

/*-------------------------------------------------------------------------------------------------------------------
   reliability  Function returns the reliability of the behaviour indicated by its argument. The reliability is a
   -----------  floating point number between 0 and 1.
                Argument:
                      n = behaviour number
-------------------------------------------------------------------------------------------------------------------*/
float reliability(char n)
{
  float pos_denom, neg_denom, max1, max2;

  pos_denom = pos_stats[n][0][0] + pos_stats[n][1][0];
  neg_denom = neg_stats[n][0][0] + neg_stats[n][1][0];

  max1 = Max((pos_stats[n][0][0] / pos_denom), (pos_stats[n][1][0] / pos_denom));
  max2 = Max((neg_stats[n][0][0] / neg_denom), (neg_stats[n][1][0] / neg_denom));

  return Min(max1, max2);
}

/*-------------------------------------------------------------------------------------------------------------------
   interestingness      Function returns the interestingness of the behaviour indicated by its argument. The
```

```
   ---------------      interestingness is a floating point number between 0 and 1.
                Argument:
                        n = behaviour number
-------------------------------------------------------------------------------------------------------------------------*/
float interestingness(char n)
{
  float pos_answer = 0.0, neg_answer = 0.0;
  char  condition_flag;

  if (behaviour_monitoring[n]) {
    condition_flag = condition_set(order_of_conditions[behaviour_n_monitoring_precond[n][0],
                                   order_of_conditions[behaviour_n_monitoring_precond[n][1]);

    pos_answer = get_int_answer(condition_flag,
                                behav_mon_prec_pos_stats[n][0][0] - behav_mon_prec_pos_stats[n][1][0],
                                behav_mon_prec_pos_stats[n][0][1] - behav_mon_prec_pos_stats[n][1][1]);
    neg_answer = get_int_answer(condition_flag,
                                behav_mon_prec_neg_stats[n][0][0] - behav_mon_prec_neg_stats[n][1][0],
                                behav_mon_prec_neg_stats[n][0][1] - behav_mon_prec_neg_stats[n][1][1]);
  }

  return Max(pos_answer, neg_answer);

}

/*-------------------------------------------------------------------------------------------------------------------------
   get_int_answer       Called by function 'interestingness' to calculate partial solution.
   --------------
                Arguments:
                        condition_on = flags whether the condition under consideration is currently on
                        on_stats     = sum of the statistics relating to the condition being on
                        off_stats    = sum of the statistics relating to the condition being off
-------------------------------------------------------------------------------------------------------------------------*/
float get_int_answer(bool condition_on, float on_stats, float off_stats)
{
  if (condition_on && on_stats < off_stats)
    return  off_stats / (on_stats + off_stats);
  else if (!condition_on && off_stats < on_stats)
    return on_stats / (on_stats + off_stats);
  else
    return 0.0;
}

/*-------------------------------------------------------------------------------------------------------------------------
   corr         Returns the Pearson product-moment correlation coefficient of the statistics passed in as a 2x2 array
   ----         Arguments:
                        n = index to the second argument
                        s[n][2][2] = an array containing the relevant statistics
-------------------------------------------------------------------------------------------------------------------------*/
float corr(char n, float s[][2][2])
{
  float denominator;


  denominator = (float) sqrt((s[n][1][1] + s[n][1][0]) * (s[n][1][1] + s[n][0][1]) * \
                             (s[n][0][0] + s[n][0][1]) * (s[n][0][0] + s[n][1][0]));

  return (denominator == 0.0) ? 0 : (s[n][0][0] * s[n][1][1] - s[n][1][0] * s[n][0][1]) / denominator;
}

/*-------------------------------------------------------------------------------------------------------------------------
   reliable     Function returns TRUE if the behaviour indicated by its argument is reliable, FALSE otherwise.
   --------     Argument:
                        n = behaviour number
-------------------------------------------------------------------------------------------------------------------------*/
bool reliable(char n)
{
  return (bool) (reliability(n) >= reliability_target);
}

/*-------------------------------------------------------------------------------------------------------------------------
   select_a_behaviour_for_group According to the relative relevance, reliability and interestingness of selectable
   ---------------------------- behaviours, and activate the choice.
                Argument:
                        n = group number
-------------------------------------------------------------------------------------------------------------------------*/
char select_a_behaviour_for_group(char n, char number_to_start, char to_start[])
{
  char  i, behaviour;
  char  behaviour_to_start = 99;
  char  number_of_selectable_behaviours = 0;
  char  selectable_behaviours[number_of_behaviours_in_group[n]];
  float prob[number_of_behaviours_in_group[n]];
  float shot;
  float total = 0.0;

  for (i = 0; i < number_of_behaviours_in_group[n]; i++)
    if (selectable(get_behaviour_number(n, i)))
      selectable_behaviours[number_of_selectable_behaviours++] = i;

  for (i = 0; i < number_of_selectable_behaviours; i++) {
    behaviour = get_behaviour_number(n, selectable_behaviours[i]);
    prob[i] = 2 * (2 - relevance(behaviour) \
                   + reliability(behaviour) \
                   + 0.5 * interestingness(behaviour));
  }

  shot = ((float) rand() / (float) RAND_MAX) * \
    11.0 * (float) number_of_selectable_behaviours;          /* maximum value of prob[n] is 11 */

  for (i = 0; total < shot && i < number_of_selectable_behaviours; i++) {
    total += prob[i];
    if (total >= shot) {
      behaviour_to_start = get_behaviour_number(n, selectable_behaviours[i]);
    }
```

```
  }

  if (behaviour_to_start < 99)
    to_start[number_to_start++] = behaviour_to_start;

  return number_to_start;
}

/*-------------------------------------------------------------------------------------------------------------------
   selectable     Returns TRUE if the behaviour indicated by the argument both is not currently active and has all
   ----------     preconditions fulfilled. Otherwise returns FALSE.
                  Argument:
                        n = behaviour number
-------------------------------------------------------------------------------------------------------------------*/
bool selectable(char n)
{
  bool  answer = true;
  int   i;

  for (i = 0; i < 4 && answer == true; i++) {
    if (behaviour_active[i])
      answer = false;
  }
  if (answer == true)
    for (i = 0; answer == true && precondition[n][i][1] != 99; i++)
      answer = !(condition_set(precondition[n][i][0], precondition[n][i][1]) == precondition[n][i][2]);

  return answer;
}

/*-------------------------------------------------------------------------------------------------------------------
   start_behaviour      Initiate the behaviour indicated by the argument.
   ---------------
                  Argument:
                        n = behaviour number
-------------------------------------------------------------------------------------------------------------------*/
void start_behaviour(char n)
{
  behaviour_active[n] = true;

  switch (n) {
  case 0:
    beta[0] = LEG_UP;
    break;
  case 1:
    beta[1] = LEG_UP;
    break;
  case 2:
    beta[2] = LEG_UP;
    break;
  case 3:
    beta[3] = LEG_UP;

    break;
  default:
    printf("START_BEHAVIOUR: invalid argument!\n");
  }

  return;
}

/*-------------------------------------------------------------------------------------------------------------------
   continue_active_behaviours   For the time being, this function just advances any legs which are currently off the
   --------------------------   ground. If such a leg reaches the forward limit of its motion, then it is lowered to
                                the ground, and the corresponding behaviour_active[] flag is set to false.
-------------------------------------------------------------------------------------------------------------------*/
void continue_active_behaviours(void)
{
  int   i, sign;

  for (i = 0; i < 4; i++) {
    sign = (i > 1) ? -1 : 1;
    if (beta[i] == LEG_UP)
      alpha[i] = plus_noise(alpha[i] - sign * swing_angle);
    if ((sign * alpha[i]) >= leg_forward_extent) {                  /* NB this is a bit of a fudge, as  */
      beta[i] = LEG_DOWN;                                           /* it equates leg numbers with      */
      alpha[i] = sign * leg_forward_extent;                         /* behaviour numbers. This is OK    */
      behaviour_active[i] = false;                                  /* with 4 behaviours, but will have */
    }                                                               /* to be changed for more behaviours*/
  }

}

/*-------------------------------------------------------------------------------------------------------------------
   global_horizontal_balance    Moves all legs which are currently on the ground in a direction such that the sum of the
   -------------------------    horizontal angles of all four legs becomes closer to zero. The percentage of the required
                                correction by which each leg is actually moved is determined by the value of the constant
                                'horiz_bal_amount' defined at the beginning of the file.
-------------------------------------------------------------------------------------------------------------------*/
void global_horizontal_balance(int angles[4])
{
  char  i;
  int   current_sum, onground, correction, sign;

  current_sum = alpha[0] + alpha[1] + alpha[2] + alpha[3];
  onground = number_on_ground();
  correction = (int) (horiz_balance_amount * ((float) current_sum / (float) onground));

  for (i = 0; i < 4; i++) {
    sign = (i > 1) ? -1 : 1;
    if (beta[i] == LEG_DOWN) {  /* should really question whether beahviours are active or inactive */
      angles[i] = plus_noise(sign * correction);
      alpha[i] += angles[i];
    }
    else
```

```
      angles[i] = 0;
    if (sign * alpha[i] > leg_forward_extent) {
      angles[i] -= sign * ((sign * alpha[i]) - leg_forward_extent);
      beta[i] = LEG_DOWN;
      alpha[i] = sign * leg_forward_extent;
      behaviour_active[i] = false;
    }
    if (sign * alpha[i] < -leg_forward_extent) {
      angles[i] += sign * ((sign * alpha[i]) + leg_forward_extent);
      beta[i] = LEG_DOWN;
      alpha[i] = sign * -leg_forward_extent;
      behaviour_active[i] = false;
    }
  }
}

/*-------------------------------------------------------------------------------------------------------------------
  plus_noise    Add a degree of noise to the parameter
  ----------
-------------------------------------------------------------------------------------------------------------------*/
int plus_noise(int mu)
{
  return (mu - 2 + (rand() % 5));                                   /* first line is for evenly  */
/*  int shot, answer, sign;                                         /* distributed noise         */

  shot = rand() % 1000;
  answer = (shot < 382) ? 0 : (shot < 866) ? 1 : 2;                 /* uncomment these lines for normally */
                                                                    /* distributed noise          */
  sign = 1 - 2 * (rand() % 2);

  return (mu + sign * answer);*/
}

/*-------------------------------------------------------------------------------------------------------------------
  get_advance_distance
  --------------------
-------------------------------------------------------------------------------------------------------------------*/
void get_advance_distance(int angles[4], float result[2])
{
  int   i, onground = 0;
  float delta_x = 0.0, delta_y = 0.0;

  for (i = 0; i < 4; i++) {
    if (pos[i][2] < offset[i][2]) {                                 /* i.e. if this foot is on ground  */
      onground++;
      delta_x += relative_pos(i, 0, (float) (alpha[i] + angles[i])) - \
        relative_pos(i, 0, (float) alpha[i]);
      delta_y += relative_pos(i, 1, (float) (alpha[i] + angles[i])) - \
        relative_pos(i, 1, (float) alpha[i]);

    }

  }

  if (onground != 0) {
    delta_x /= (float) onground;                                    /* get average of delta x          */
    delta_y /= (float) onground;                                    /* and of delta y                  */
  }

  result[0] = delta_x;
  result[1] = delta_y;

}

/*-------------------------------------------------------------------------------------------------------------------
  relative_pos Function used in the calculation of the change in x and y coordinates of the feet as the robot
  ------------ advances.
-------------------------------------------------------------------------------------------------------------------*/
float relative_pos(int i, int j, float angle)
{
  return pos1(i, j, angle, (float) beta[i]) + pos2(i, j, angle, (float) beta[i]);
}

/*-------------------------------------------------------------------------------------------------------------------
  update_stats
  ------------
-------------------------------------------------------------------------------------------------------------------*/
void update_stats(char n, bool pos_fb, bool neg_fb)
{
  char  index, i, j;

  for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++) {
      pos_stats[n][i][j] *= stats_decay_rate;
      neg_stats[n][i][j] *= stats_decay_rate;
    }

  pos_stats[n][pos_fb ? 0 : 1][behaviour_active[n] ? 0 : 1] += stats_initial_value * (1.0 - stats_decay_rate);
  neg_stats[n][neg_fb ? 0 : 1][behaviour_active[n] ? 0 : 1] += stats_initial_value * (1.0 - stats_decay_rate);

  if (behaviour_monitoring[n]) {
    monitoring_clock[n]--;
    if (behaviour_active[n]) {
      index = behaviour_n_monitoring_precond[n];
      behav_mon_prec_pos_stats[n][pos_fb ? 0 : 1]\
        [condition_set[order_of_conditions[index][0], order_of_conditions[index][1]) ? 0 : 1]--;
      behav_mon_prec_neg_stats[n][neg_fb ? 0 : 1]\
        [condition_set[order_of_conditions[index][0], order_of_conditions[index][1]) ? 0 : 1]--;
    }
  }

}

/*-------------------------------------------------------------------------------------------------------------------
```

```
/*-------------------------------------------------------------------------------------------------------------------
   monitor_next_condition
   ----------------------
                Argument:
                        n = behaviour number
-------------------------------------------------------------------------------------------------------------------*/
void monitor_next_condition(char n)
{
  char  i, j;

  behaviour_n_monitoring_precond[n]++;
  if (order_of_conditions[behaviour_n_monitoring_precond[n]][1] == 99)
    behaviour_n_monitoring_precond[n] = 0;

  behaviour_monitoring[n] = true;
}

/*-------------------------------------------------------------------------------------------------------------------
   continue_monitoring  Continue monitoring a condition for the behaviour specified by the argument. Updates the
   -------------------  monitoring clock, and will stop monitoring when monitor_duration is surpassed. Otherwise,
                        if the correlation between the condition and feedback is strong enough, then it is adopted
                        as a new precondition. If it is not strong enough after monitoring, then the condition is
                        dropped (if it is already in the precondition list, it is removed).
-------------------------------------------------------------------------------------------------------------------*/
void continue_monitoring(char n)
{
  float correl;

  if (monitoring_clock[n] > monitor_duration)
    drop_condition(n);
  else {
    if ((correl = corr(n, behav_mon_prec_neg_stats)) >= correlation_threshold)
      adopt_condition(n, false);
    else if (correl <= -correlation_threshold)
      adopt_condition(n, true);
    else if ((correl = corr(n, behav_mon_prec_pos_stats)) >= correlation_threshold)
      adopt_condition(n, true);
    else if (correl <= -correlation_threshold)
      adopt_condition(n, false);
  }

}

/*-------------------------------------------------------------------------------------------------------------------
   drop_condition
   --------------
                Argument:
                        n = number of the behaviour which is monitoring the condition
-------------------------------------------------------------------------------------------------------------------*/
void drop_condition(char n)
{
  int   i, j;


  behaviour_monitoring[n] = false;

  for (i = 0; precondition[n][i][1] != 99; i++)
    if (precondition[n][i][0] == order_of_conditions[behaviour_n_monitoring_precond[n]][0] && \
        precondition[n][i][1] == order_of_conditions[behaviour_n_monitoring_precond[n]][1]) {
      for (; precondition[n][i][1] != 99; i++)                          /* shift all preconditions above */
        for (j = 0; j < 3; j++)                                         /* the monitored one down by one */
          precondition[n][i][j] = precondition[n][i + 1][j];            /* position in the list          */
      break;
    }

  clear_monitoring_state(n);
}

/*-------------------------------------------------------------------------------------------------------------------
   adopt_condition
   ---------------
                Argument:
                        n = number of the behaviour which is monitoring the condition
-------------------------------------------------------------------------------------------------------------------*/
void adopt_condition(char n, bool value)
{
  bool  there_already = false;
  int   i;

  behaviour_monitoring[n] = false;

  for (i = 0; precondition[n][i][1] != 99; i++)
    if (precondition[n][i][0] == order_of_conditions[behaviour_n_monitoring_precond[n]][0] && \
        precondition[n][i][1] == order_of_conditions[behaviour_n_monitoring_precond[n]][1]) {
      there_already = true;
      break;
    }

  precondition[n][i][2] = value;

  if (!there_already) {
    precondition[n][i][0] = order_of_conditions[behaviour_n_monitoring_precond[n]][0];
    precondition[n][i][1] = order_of_conditions[behaviour_n_monitoring_precond[n]][1];
    precondition[n][i + 1][1] = 99;
  }

  clear_monitoring_stats(n);
}

/*-------------------------------------------------------------------------------------------------------------------
   clear_monitoring_stats       Reset the statistics for the given behaviour to zero, and reset the monitoring clock
   ----------------------
                Argument:
                        n = behaviour number
-------------------------------------------------------------------------------------------------------------------*/
void clear_monitoring_stats(char n)
```

```
{
  char  i, j;

  for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++) {
      behav_mon_prec_pos_stats[n][i][j] = 0.0;
      behav_mon_prec_neg_stats[n][i][j] = 0.0;
    }

  monitoring_clock[n] = 0;
}

/*-------------------------------------------------------------------------------------------------------------------
   get_behaviour_number Function returns the real number of a behaviour given its group number and number within that
   -------------------- group.
                Arguments:
                        group = group number
                        group_behaviour = number of the behaviour within that group
-------------------------------------------------------------------------------------------------------------------*/
char get_behaviour_number(char group, char group_behaviour)
{
  int   answer = 0;

  for (; group-- > 0;)
    answer += number_of_behaviours_in_group[group];

  return (char) (answer + group_behaviour);
}

/*-------------------------------------------------------------------------------------------------------------------
   update_output_file   Write data for this time-step to the output file referred to by the 'ofp' pointer.
   ------------------   The template for this file is as follows:-

        time_step_counter               x 1
        condition vectors:              leg_n_up x 4, leg_n_forward x 4, leg_n_backward x 4
        precondition                    x 4 x 13 x 3
        behaviour_active                x 4
        behaviour_monitoring            x 4
        pos_stats                       x 4 x 2 x 2
        neg_stats                       x 4 x 2 x 2
        behaviour_n_monitoring_precond  x 4
        monitoring_clock                x 4
        behav_mon_prec_pos_stats        x 4 x 2 x 2
        behav_mon_prec_neg_stats        x 4 x 2 x 2
        percentage positive feedback    x 1
        percentage negative feedback    x 1
        relevance                       x 4
        reliability                     x 4
-------------------------------------------------------------------------------------------------------------------*/
void update_output_file(void)
{

  int   i, j, k;

  fprintf(ofp, "%-5i ", time_step_counter);
  for (i = 0; i < 3; i++)
    for (j = 0; j < 4; j++)
      fprintf(ofp, "%-1i ", (i == 0) ? leg_n_up[j] : (i == 1) ? leg_n_forward[j] : leg_n_backward[j]);
  for (i = 0; i < 4; i++)
    for (j = 0; j < 13; j++)
      for (k = 0; k < 3; k++)
        fprintf(ofp, "%-2i ", precondition[i][j][k]);
  for (i = 0; i < 4; i++)
    fprintf(ofp, "%-1i ", behaviour_active[i]);
  for (i = 0; i < 4; i++)
    fprintf(ofp, "%-1i ", behaviour_monitoring[i]);
  for (i = 0; i < 4; i++)
    for (j = 0; j < 2; j++)
      for (k = 0; k < 2; k++)
        fprintf(ofp, "%-7.3f ", pos_stats[i][j][k]);
  for (i = 0; i < 4; i++)
    for (j = 0; j < 2; j++)
      for (k = 0; k < 2; k++)
        fprintf(ofp, "%-7.3f ", neg_stats[i][j][k]);
  for (i = 0; i < 4; i++)
    fprintf(ofp, "%-2i ", behaviour_n_monitoring_precond[i]);
  for (i = 0; i < 4; i++)
    fprintf(ofp, "%-4i ", monitoring_clock[i]);
  for (i = 0; i < 4; i++)
    for (j = 0; j < 2; j++)
      for (k = 0; k < 2; k++)
        fprintf(ofp, "%-7.3f ", behav_mon_prec_pos_stats[i][j][k]);
  for (i = 0; i < 4; i++)
    for (j = 0; j < 2; j++)
      for (k = 0; k < 2; k++)
        fprintf(ofp, "%-7.3f ", behav_mon_prec_neg_stats[i][j][k]);
  fprintf(ofp, "%-3i ", get_fb_stat(pos_fb_array));
  fprintf(ofp, "%-3i ", get_fb_stat(neg_fb_array));
  for (i = 0; i < 4; i++)
    fprintf(ofp, "%-5.2f ", relevance[i]);
  for (i = 0; i < 4; i++)
    fprintf(ofp, "%-4.2f ", reliability[i]);
  fprintf(ofp, "\n");
}
```

```
/*-------------------------------------------------------------------------------
   ISSTABLE     Program to check whether Smudge will be stable for a given set of leg angles.
-------------------------------------------------------------------------------*/

#include "top.h"

float   get_normal(int, int, int);
int     number_on_ground(void);
int     get_sign(int, int [], float [][]);
void    get_stability_info(float []);
void    check_stable(int, float[]);

/*-------------------------------------------------------------------------------
   get_stability_info   Returns a number indicating how stable the robot is. If 3 or 4 legs are on the ground, then the
   ------------------   value returned is the shortest distance between the robot CoM and the boundary of the polygon of
                        support defined by the leg positions. A negative distance indicates that the robot is unstable.
                        If fewer than 3 legs are on the ground, then -999.9 is returned, indicating that the robot is
                        very instable.
-------------------------------------------------------------------------------*/
void get_stability_info(float answer[3])
{
  int onground;

  onground = number_on_ground();

  if (onground < 3)
    answer[0] = -999.9;
  else
    check_stable(onground, answer);

  return;
}

/*-------------------------------------------------------------------------------
   check_stable Called by stable if there are 3 or 4 legs on the ground. Calculates whether the robot CoM lies within
   ------------ the polygon of support, and returns the shortest distance from the CoM to the boundary of the polygon.
-------------------------------------------------------------------------------*/
void check_stable(int onground, float answer[3])
{
  int i, j;
  int n = 0;
  int index[4];

  float normal[4][2];                                   /* The normal vectors of support polygon.    */
  float gradient[4];                                    /* Gradients of normal vectors.              */
  float point[4][2];                                    /* Points on polygon boundary closest to CoM.*/
  float distance[4];                                    /* Distance of each point to CoM.            */
  float min_distance;

  for (i = 0; i <= 3; i++)                              /* index[] points to the feet which          */
    if (pos[i][2] < offset[i][2])                       /* are on the ground.                        */
      index[n++] = i;


  for (i = 0; i < onground; i++)                        /* Get normal vectors for each line          */
    for (j = 0; j <= 1; j++)                            /* Equation of each line expressed as        */
      normal[i][j] = get_normal(index[i], index[(i + 1) % onground], j);   /*       (r - p).n = 0       */
                                                        /*        -   -  -                           */
  for (i = 0; i < onground; i++) {
    if (fabs(normal[i][0]) < 0.001) {                   /* i.e. if gradient is infinite              */
      point[i][0] = 0.0;
      point[i][1] = pos[index[i]][1];
    }
    else {
      gradient[i] = normal[i][1] / normal[i][0];        /* Find gradient of normal vector.           */
      point[i][0] = ((pos[index[i]][0] * normal[i][0]) + (pos[index[i]][1] * normal[i][1])) \
        / (normal[i][0] + (gradient[i] * normal[i][1]));/* Now find the point on each line           */
      point[i][1] = gradient[i] * point[i][0];          /* which is perpendicular to the             */
    }                                                   /* CoM (which is (0,0) in the robot          */
    distance[i] = (float)                               /* coordinate frame), and the distance       */
      sqrt(pow((double) point[i][0], 2.0) + pow((double) point[i][1], 2.0));   /* from the point to the CoM.*/

    if (i == 0 || distance[i] < min_distance) {
      min_distance = distance[i];
      answer[1] = (float) index[i];
      answer[2] = (float) index[(i + 1) % onground];
    }

  }

  answer[0] = (float) get_sign(onground, index, point) * min_distance;

  return;
}

/*-------------------------------------------------------------------------------
   get_normal   Returns the nth component of the normal vector of the line connecting points i and j.
   ----------
                Arguments:
                        i, j, n
-------------------------------------------------------------------------------*/
float get_normal(int i, int j, int n)
{
  int p, sign;
                                                        /* Normal of 2D vector (x,y) is (y,-x) */
  p = 1 - n;
  sign = (n == 0) ? 1 : -1;

  return (float) sign * (pos[i][p] - pos[j][p]);
}

/*-------------------------------------------------------------------------------
   number_on_ground     Returns the number of feet on the ground.
   ----------------
-------------------------------------------------------------------------------*/
```

```
int number_on_ground(void)
{
  char i;
  char onground = 0;

  for (i = 0; i <= 3; i++)
/*    onground += (pos[i][2] < offset[i][2]); */
    onground += (beta[i] == LEG_DOWN) ? 1 : 0;

  return onground;
}

/*-------------------------------------------------------------------------------------------------------------------
   get_sign      Returns 1 if robot CoM lies within polygon of support defined by feet, -1 otherwise.
-------------------------------------------------------------------------------------------------------------------*/
int get_sign(int onground, int index[4], float point[4][2])
{
  int sign;                                                          /* Remember, robot CoM is at (0,0)     */

  switch(onground) {
  case 3:                                                            /* Case: 3 legs are on the ground      */
    if (index[0] == 1 || index[1] == 2)                              /* i.e. if legs 0 or 1 are off ground  */
      sign = (point[0][0] > 0 && point[1][1] < 0 && point[2][0] < 0) ? 1 : -1;
    else
      sign = (point[0][1] > 0 && point[1][0] > 0 && point[2][0] < 0) ? 1 : -1;
    break;
  case 4:                                                            /* Case: 4 legs are on the grpund      */
    sign = (point[0][1] > 0 && point[1][0] > 0 && point[2][1] < 0 && \
            point[3][0] < 0) ? 1 : -1;
    break;
  }

  return sign;
}
```

```
/*-------------------------------------------------------------------------------------------------------------------
   POS          Functions to return cartesian coordinates of the robot's legs (in the robot's coordinate frame) given
                the leg angles.
-------------------------------------------------------------------------------------------------------------------*/

#include "top.h"

float pos[4][3];

void get_positions(void);

float pos1(char, char, float, float);
float pos2(char, char, float, float);

double radians(float);

/*-------------------------------------------------------------------------------------------------------------------
   get_positions        Find (x,y,z) coordinates (in robot coordinate frame) for each foot.
   -------------
-------------------------------------------------------------------------------------------------------------------*/
void get_positions(void)
{
  char i, j;

  for (i = 0; i <= 3; i++)
    for (j = 0; j <= 2; j++)
      pos[i][j] = offset[i][j] + pos1(i, j, (float) alpha[i], (float) beta[i]) + pos2(i, j, (float) alpha[i], (float) beta[i]);
}

/*-------------------------------------------------------------------------------------------------------------------
   pos1         Return requested component of position of top of leg relative to servo axis of rotation.
   ----

                Arguments:
                        i = leg number (0 - 3)
                        j = ordinate (0=x, 1=y, 2=z)
                        angle1 = alpha angle for leg i
                        angle2 = beta angle for leg i
-------------------------------------------------------------------------------------------------------------------*/
float pos1(char i, char j, float angle1, float angle2)
{
  char sign;

  switch (j) {
  case 0:
    sign = (i == 1 || i == 2) ? -1 : 1;
    return (float)
      (sign * SERVO1_L1 * sin(radians(angle1)));
  case 1:
    sign = (i == 2 || i == 3) ? -1 : 1;
    return (float)
      (sign * SERVO1_L1 * cos(radians(angle1)));
  case 2:

    return 0.0;
  default:
    return 999.9;
  }
}

/*-------------------------------------------------------------------------------------------------------------------
   pos2         Return requested component of position of foot of leg relative to top of leg.
   ----

                Arguments:
                        i = leg number (0 - 3)
                        j = ordinate (0=x, 1=y, 2=z)
                        angle1 = alpha angle for leg i
                        angle2 = beta angle for leg i
-------------------------------------------------------------------------------------------------------------------*/
float pos2(char i, char j, float angle1, float angle2)
{
  int sign;

  switch (j) {
  case 0:
    sign = (i == 0 || i == 3) ? -1 : 1;
    return (float)
      (sign * LEG_LENGTH * cos(radians(angle2)) *
      cos(radians(angle1)));
  case 1:
    sign = (i == 2 || i == 3) ? -1 : 1;
    return (float)
      (sign * LEG_LENGTH * cos(radians(angle2)) *
       sin(radians(angle1)));
  case 2:
    return (float) (- LEG_LENGTH * sin(radians(angle2)));
  default:
    return 999.9;
  }
}

/*-------------------------------------------------------------------------------------------------------------------
  radians       Convert degrees into radians
  -------

                Arguments:
                        degrees (integer)
-------------------------------------------------------------------------------------------------------------------*/
double radians(float x)
{
  return ((double) x) / 180.0 * M_PI;
}
```

```
/*-------------------------------------------------------------------------------------------------
   LENGTHS.H    Specifies the dimensions of Smudge, plus the beta angles for raised and lowered legs.
-------------------------------------------------------------------------------------------------*/

#define BODY_LENGTH      250
#define BODY_BREADTH     65
#define SERVO1_L1        20
#define SERVO2_L1        7
#define SERVO_POSITION   30
#define LEG_VERT_OFFSET  20
#define LEG_LENGTH       150
#define LEG_DOWN         30
#define LEG_UP           -10
```

```
/*-------------------------------------------------------------------------------------------------
   TOP.H          Specifies Include files, standard definitions and external functions and variables.
-------------------------------------------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "useful.h"
#include "lengths.h"

#define offsetx          (float) ((((float) BODY_BREADTH) / 2.0) + SERVO2_L1)
#define offsety          (float) ((((float) BODY_LENGTH) / 2.0) - SERVO_POSITION)
#define offsetz          (float) LEG_VERT_OFFSET
#define FB_STAT_SLICE    (int)   200

extern float    pos1(char, char, float, float);                          /* functions defined in */
extern float    pos2(char, char, float, float);                          /* file pos.c           */
extern double   radians(float);

extern float    get_normal(int, int, int);                               /* functions defined in */
extern int      number_on_ground(void);                                  /* file isstable.c      */
extern int      get_sign(int, int [], float [][]);
extern void     get_stability_info(float []);
extern void     check_stable(int, float []);

extern void     initialise_leg_positions(void);                          /* functions defined in */
extern void     initialise_learning(void);                               /* file algorithm.c     */
extern void     next_time_step(float [], float [], bool *);

extern int      get_fb_stat(char []);                                    /* function defined in  */
                                                                         /* sim.c                */

extern int      alpha[4], beta[4];                                       /* global variables     */
extern float    pos[4][3];
extern const float offset[4][3];
extern int      advance_angle;
extern char     precond_data[4][13][3], monitor_data[4][2];
extern int      monitor_clock_data[4];
extern char     pos_fb_array[FB_STAT_SLICE], neg_fb_array[FB_STAT_SLICE];
extern int      time_step_counter;
extern FILE     *ofp;
```

Appendix C

Program Code for the Simulator

```
/*-------------------------------------------------------------------------------------------------------------------
   SIM          XView code and general control routines for the simulation
-------------------------------------------------------------------------------------------------------------------*/

#include "xview/xv_headers.hh"
#include "top.h"

#define SHRINK_FACTOR           (float) 2.0
#define S_BODY_LENGTH           (float) ((float) BODY_LENGTH / SHRINK_FACTOR)
#define S_BODY_BREADTH          (float) ((float) BODY_BREADTH / SHRINK_FACTOR)
#define S_BODY_TOPLEFT_X        (float) (S_BODY_LENGTH / 2.0)
#define S_BODY_TOPLEFT_Y        (float) (S_BODY_BREADTH / 2.0)
#define s_offsetx               (float) (offsetx / SHRINK_FACTOR)
#define s_offsety               (float) (offsety / SHRINK_FACTOR)
#define s_offsetz               (float) (offsetz / SHRINK_FACTOR)

const float offset[4][3] = {
                {-offsetx,  offsety, -offsetz},
                { offsetx,  offsety, -offsetz},
                { offsetx, -offsety, -offsetz},
                {-offsetx, -offsety, -offsetz}
        };

const float s_offset[4][3] = {
                {-s_offsetx,  s_offsety, -s_offsetz},
                { s_offsetx,  s_offsety, -s_offsetz},
                { s_offsetx, -s_offsety, -s_offsetz},
                {-s_offsetx, -s_offsety, -s_offsetz}
        };

typedef struct position {
          int           x_coord;

          int           y_coord;
          float         hip_pos[4][3];
          float         foot_pos[4][3];
} Position;

Frame           frame;
Display         *dpy;
GC              gc;
Window          canvas_win;
Panel           panel1, panel2;
Canvas          canvas;
struct itimerval timer;
Panel_item      slider, leg[4][2], info_msg[6], behaviour_msg[4][3], stability_msg;
Position        oldposition, newposition;
bool            last_fb = false;
char            go_ahead = FALSE;
char            info[6][10];
char            beta_message[40];
char            stability_string[20];
char            behaviour_info[4][3][43];
char            precond_data[4][13][3], monitor_data[4][2];
char            pos_fb_array[FB_STAT_SLICE], neg_fb_array[FB_STAT_SLICE];
int             monitor_clock_data[4];
int             alpha[4], beta[4];
float           total_x = 0.0, total_y = 0.0;                                   \
FILE            *ofp;                                                                   /* pointer to output file */

Notify_value    animate(void);
Position        make_position(int, int);
Position        get_new_position(Position, float []);
char            *itoa(int);
char            *get_precondition_string(char);
char            *get_condition_string(char, char);
int             get_fb_stat(char array[]);                      \
void            delete(Position);
void            draw(Position, int, int, float);
void            DrawDottedLine(int, int, int, int, int);
void            swap(int *, int *, int *, int *);
void            update_display_data(int);
void            update_statistics(int);
void            initialise_statistics(void);
void            change_alpha(Panel_item), change_beta(Panel_item);          \

main(int argc, char *argv[])
{
  void          quit(void), start(void), stop(void);
  void          adjust_speed(Panel_item, int);
  void          canvas_repaint(Canvas, Xv_Window, Display, Window, Xv_xrectlist);

  char          rows, cols, i, index;
  char          *text[] = {"Front Left", "Front Right", "Rear Right", "Rear Left"};
  char          *lines = "_____________________________";          }
```

```
char            *label[] = {"Elapsed time-steps :", "% positive feedback :", \
                            "% negative feedback :", "Distance travelled (cm)", \
                            "                on x-axis :", "                on y-axis :"};
char            behaviour_text[4][13];
int             button_x = 4, button_y = 100;                   /* origin of panel items for controlling leg ang
int             info_x = button_x, info_y = button_y + 210;     /* origin of panel items for status information
int             stability_x = info_x + 65, stability_y = info_y + 140;  /* origin of stability message

time_t          now;                                            /* used for time-stamping the output file */

strcpy(beta_message, "beta angles:   down = ");                 /* initialise message string to display beta ang.
strcat(beta_message, itoa(LEG_DOWN));
strcat(beta_message, ",    up = ");
strcat(beta_message, itoa(LEG_UP));

for (i = 0; i < 4; i++) {                                       /* initialise array of message strings to displa
  strcpy(behaviour_text[i], "BEHAVIOUR ");                      /* behaviour numbers
  strcat(behaviour_text[i], itoa(i));
  strcat(behaviour_text[i], ":");
}

initialise_leg_positions();

xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

frame = (Frame) xv_create(NULL, FRAME,
        FRAME_LABEL,            "Smudge Simulator",
        XV_WIDTH,               985,
        XV_HEIGHT,              545,
        NULL);

panel1 = (Panel) xv_create(frame, PANEL,
        PANEL_LAYOUT,           PANEL_VERTICAL,
        OPENWIN_SHOW_BORDERS,   TRUE,
        XV_WIDTH,               280,
        XV_HEIGHT,              540,
        XV_X,                   0,
        XV_Y,                   0,
        NULL);
xv_create(panel1, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Start",
        PANEL_NOTIFY_PROC,      start,
        NULL);
xv_create(panel1, PANEL_BUTTON,
        XV_X,                   60,
        XV_Y,                   4,
        PANEL_LABEL_STRING,     "Stop",
        PANEL_NOTIFY_PROC,      stop,
        NULL);
xv_create(panel1, PANEL_BUTTON,
        XV_X,                   220,
        XV_Y,                   4,

        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      quit,
        NULL);
slider = (Panel_item) xv_create(panel1, PANEL_SLIDER,
        PANEL_LABEL_STRING,     "Speed",
        PANEL_SHOW_VALUE,       FALSE,
        PANEL_VALUE,            50,
        PANEL_NOTIFY_PROC,      adjust_speed,
        NULL);
xv_create(panel1, PANEL_MESSAGE,
        PANEL_LABEL_STRING,     lines,
        NULL);

for (rows = 0; rows < 2; rows++)
  for (cols = 0; cols < 2; cols++) {
    index = (cols * rows) + ((1 - cols) * (3 - rows)); /* index is set to the correct leg no. as used by alpha[] etc */
    xv_create(panel1, PANEL_MESSAGE,
      PANEL_LABEL_STRING,       text[index],
      XV_X,                     button_x + (136 * cols),
      XV_Y,                     button_y + (80 * rows),
      NULL);
    xv_create(panel1, PANEL_MESSAGE,
      PANEL_LABEL_STRING,       "alpha",
      PANEL_LABEL_BOLD,         TRUE,
      XV_X,                     button_x + (136 * cols),
      XV_Y,                     button_y + 20 + (80 * rows),
      NULL);
    leg[index][0] = (Panel_item) xv_create(panel1, PANEL_NUMERIC_TEXT,
      XV_X,                     button_x + 50 + (136 * cols),
      XV_Y,                     button_y + 20 + (80 * rows),
      PANEL_MAX_VALUE,          45,
      PANEL_MIN_VALUE,          -45,
      PANEL_VALUE,              alpha[index],
      PANEL_VALUE_DISPLAY_LENGTH,       3,
      PANEL_VALUE_STORED_LENGTH,        3,
      PANEL_NOTIFY_PROC,        change_alpha,
      PANEL_CLIENT_DATA,        index,
      NULL);
    leg[index][1] = (Panel_item) xv_create(panel1, PANEL_CHOICE,
      XV_X,                     button_x + (136 * cols),
      XV_Y,                     button_y + 40 + (80 * rows),
      PANEL_CHOICE_STRINGS,     "Down", "Up", NULL,
      PANEL_VALUE,              0,
      PANEL_NOTIFY_PROC,        change_beta,
      PANEL_CLIENT_DATA,        index,
      NULL);
  }

xv_create(panel1, PANEL_MESSAGE,
      XV_X,                     button_x,
      XV_Y,                     button_y + 160,
      PANEL_LABEL_STRING,       beta_message,
```

```
        NULL);
    xv_create(panel1, PANEL_MESSAGE,
        PANEL_LABEL_STRING,     lines,
        NULL);

    for (rows = 0; rows < 6; rows++) {
      xv_create(panel1, PANEL_MESSAGE,
        XV_X,                   info_x,
        XV_Y,                   info_y + (18 * rows),
        PANEL_LABEL_STRING,     label[rows],
        NULL);
      info_msg[rows] = (Panel_item) xv_create(panel1, PANEL_MESSAGE,
        XV_X,                   info_x + 150,
        XV_Y,                   info_y + (18 * rows),
        PANEL_LABEL_STRING,     info[rows],
        NULL);
    }
    xv_create(panel1, PANEL_MESSAGE,
        PANEL_LABEL_STRING,     lines,
        NULL);

    stability_msg = (Panel_item) xv_create(panel1, PANEL_MESSAGE,
        XV_X,                   stability_x,
        XV_Y,                   stability_y,
        PANEL_LABEL_STRING,     stability_string,
        NULL);

    panel2 = (Panel) xv_create(frame, PANEL,
        PANEL_LAYOUT,           PANEL_HORIZONTAL,
        OPENWIN_SHOW_BORDERS,   TRUE,
        XV_WIDTH,               700,
        XV_HEIGHT,              140,
        XV_X,                   280,
        XV_Y,                   0,
        NULL);

    for (rows = 0; rows < 2; rows++)
      for (cols = 0; cols < 2; cols++) {
        index = (rows * 2) + cols;
        xv_create(panel2, PANEL_MESSAGE,
          PANEL_LABEL_STRING,   behaviour_text[index],
          PANEL_LABEL_BOLD,     TRUE,
          XV_X,                 4 + (350 * cols),
          XV_Y,                 4 + (70 * rows),
          NULL);
        xv_create(panel2, PANEL_MESSAGE,
          PANEL_LABEL_STRING,   "Preconditions             Monitoring  Clock",
          XV_X,                 4 + (350 * cols),
          XV_Y,                 24 + (70 * rows),
          NULL);
        xv_create(panel2, PANEL_MESSAGE,
          PANEL_LABEL_STRING,   "------------              --------  -----",

          XV_X,                 4 + (350 * cols),
          XV_Y,                 34 + (70 * rows),
          NULL);
        for (i = 0; i < 3; i++)
          behaviour_msg[index][i] = (Panel_item) xv_create(panel2, PANEL_MESSAGE,
                PANEL_LABEL_STRING,     behaviour_info[index][i],
                XV_X,                   4 + ((i == 0) ? 0 : (i == 1) ? 155 : 215) + (350 * cols),
                XV_Y,                   44 + (70 * rows),
                NULL);
      }

    canvas = (Canvas) xv_create(frame, CANVAS,
        CANVAS_REPAINT_PROC,    canvas_repaint,
        CANVAS_X_PAINT_WINDOW,  TRUE,
        OPENWIN_SHOW_BORDERS,   TRUE,
        XV_WIDTH,               700,
        XV_HEIGHT,              400,
        XV_X,                   282,
        XV_Y,                   142,
        NULL);
    canvas_win = (Window) xv_get(canvas_paint_window(canvas), XV_XID);

    dpy = (Display *) xv_get(frame, XV_DISPLAY);
    gc = DefaultGC(dpy, DefaultScreen(dpy));

    oldposition = make_position(200, 0);
    initialise_statistics();
    initialise_learning();

    ofp = fopen("data/data.out", "a");
    now = time(NULL);
    fprintf(ofp, "SMUDGE Simulation: Session began %s\n", ctime(&now));
    srand(time(NULL));                                  /* seed the random number generator */
    xv_main_loop(frame);
    fclose(ofp);
    exit(0);
}

void quit(void)                                         /* called by Quit button */
{
    xv_destroy_safe(frame);
}

void start(void)                                        /* called by Start button */
{
    int value;

    go_ahead = TRUE;

    value = (int) xv_get(slider, PANEL_VALUE);

    if (value > 0) {
```

```
    timer.it_value.tv_usec = (130 - value) * 1000;
    timer.it_interval.tv_usec = (130 - value) * 1000;
    notify_set_itimer_func(frame, animate, ITIMER_REAL, &timer, NULL);
  }
  else
    /* turn it off */
    notify_set_itimer_func(frame, NOTIFY_FUNC_NULL, ITIMER_REAL, NULL, NULL);

  /* advance one frame */
  animate();
}

void stop(void)                                                    /* called by Stop button */
{
  go_ahead = FALSE;

  notify_set_itimer_func(frame, NOTIFY_FUNC_NULL, ITIMER_REAL, NULL, NULL);
}

void canvas_repaint(Canvas canvas, Xv_Window paint_window, Display *dpy, Window xwin, Xv_xrectlist *area)
{
  int   width, height;

  width = (int) xv_get(paint_window, XV_WIDTH);
  height = (int) xv_get(paint_window, XV_HEIGHT);

  XClearArea(dpy, xwin, 0, 0, width, height, FALSE);
  draw(oldposition, 0, 0, 0.0);
}

Notify_value animate(void)                                         /* draw Smudge at next */
{                                                                  /* time-step           */
  float advance_values[2], stable_values[3];

  next_time_step(advance_values, stable_values, &last_fb);
  newposition = get_new_position(oldposition, advance_values);
  delete(oldposition);
  draw(newposition, (int) stable_values[1], (int) stable_values[2], stable_values[0]);
  oldposition = newposition;

  return NOTIFY_DONE;
}

void adjust_speed(Panel_item item, int value)
{

  if (go_ahead == TRUE) {
    if (value > 0) {
      timer.it_value.tv_usec = (130 - value) * 1000;
      timer.it_interval.tv_usec = (130 - value) * 1000;
      notify_set_itimer_func(frame, animate, ITIMER_REAL, &timer, NULL);
    } else

      /* turn it off */
      notify_set_itimer_func(frame, NOTIFY_FUNC_NULL, ITIMER_REAL, NULL, NULL);
  }
}

void change_alpha(Panel_item item)                                 /* manual adjustment of */
{                                                                  /* alpha values from    */
  int i = (int) xv_get(item, PANEL_CLIENT_DATA);                   /* control panel        */
  alpha[i] = xv_get(item, PANEL_VALUE);
}

void change_beta(Panel_item item)                                  /* manual adjustment of */
{                                                                  /* beta values from     */
  int i = (int) xv_get(item, PANEL_CLIENT_DATA);                   /* control panel        */
  beta[i] = (xv_get(item, PANEL_VALUE) == 1) ? LEG_DOWN : LEG_UP;
}

Position make_position(int x, int y)                               /* create a new Position */
{                                                                  /* data structure       */
  Position      temp;
  char          i, j;
  int           half_height = ((int) xv_get(canvas, CANVAS_HEIGHT)) / 2;

  get_positions();

  temp.x_coord = x;
  temp.y_coord = half_height - y % half_height;
  for (i = 0; i < 4; i++)
    for (j = 0; j < 3; j++) {
      temp.hip_pos[i][j] = s_offset[i][j] - (pos[i, j, (float) alpha[i], (float) beta[i]) / SHRINK_FACTOR);
      temp.foot_pos[i][j] = pos[i][j] / SHRINK_FACTOR;
    }

  return temp;
}

Position get_new_position(Position old_pos, float advance_values[])  /* return a new Position data structure  */
{                                                                    /* based on the old structure and advance */
  Position      temp;                                                /* values passed into the function        */
  char          i, j;
  int           half_height = ((int) xv_get(canvas, CANVAS_HEIGHT)) / 2;

  temp.x_coord = ((old_pos.x_coord - (int) S_BODY_TOPLEFT_X - (int) (advance_values[1] / SHRINK_FACTOR)) \
                  % ((int) xv_get(canvas, CANVAS_WIDTH)) - (int) S_BODY_LENGTH) \
                 - (int) S_BODY_TOPLEFT_X;
  temp.y_coord = half_height - ((old_pos.y_coord - half_height) - (int) (advance_values[0] / SHRINK_FACTOR)) % half_height;

  for (i = 0; i < 4; i++)
    for (j = 0; j < 3; j++) {
      temp.hip_pos[i][j] = s_offset[i][j] - (pos[i, j, (float) alpha[i], (float) beta[i]) / SHRINK_FACTOR);
      temp.foot_pos[i][j] = pos[i][j] / SHRINK_FACTOR;
    }
```

```
        NULL);
  }

  for (i = 0; i < 6; i++)
    if (strcmp((char *) xv_get(info_msg[i], PANEL_LABEL_STRING), info[i]) != 0)
      xv_set(info_msg[i],
        PANEL_LABEL_STRING,     info[i],
        NULL);

  if (strncmp((char *) xv_get(stability_msg, PANEL_LABEL_STRING), stability_string, 12) != 0)
    xv_set(stability_msg,
        PANEL_LABEL_STRING,     stability_string,
        NULL);

  for (i = 0; i < 4; i++)
    for (j = 0; j < 3; j++)
      if (strcmp((char *) xv_get(behaviour_msg[i][j], PANEL_LABEL_STRING), behaviour_info[i][j]) != 0)
        xv_set(behaviour_msg[i][j],
               PANEL_LABEL_STRING,     behaviour_info[i][j],
               NULL);
}

void update_statistics(int stable_flag)
{
  char i;

  strcpy(info[0], itoa(time_step_counter));

  strcpy(info[1], itoa(get_fb_stat(pos_fb_array)));
  strcpy(info[2], itoa(get_fb_stat(neg_fb_array)));

  strcpy(info[4], itoa((int) (total_x / 10.0)));
  strcpy(info[5], itoa((int) (total_y / 10.0)));

  if (stable_flag)
    strcpy(stability_string, " Smudge is stable");
  else
    strcpy(stability_string, "SMUDGE IS UNSTABLE!");

  for (i = 0; i < 4; i++) {
    strcpy(behaviour_info[i][0], get_precondition_string(i));
    if (monitor_clock_data[i] == 0)
      strcpy(behaviour_info[i][1], "");
    else
      strcpy(behaviour_info[i][1], get_condition_string(monitor_data[i][0], monitor_data[i][1]));
    strcpy(behaviour_info[i][2], itoa(monitor_clock_data[i]));
  }
}

char *get_precondition_string(char n)                                   /* function returns a char array of all */
{                                                                       /* current preconditions for behaviour  */
  char i;                                                               /* n suitable for display on the panel  */
  char answer[43];

  strcpy(answer, "");
  for (i = 0; precond_data[n][i][1] != 99; i++) {                       /* 99 marks end of precondition list */
    if (i > 0)
      strcat(answer, ", ");
    strcat(answer, get_condition_string(precond_data[n][i][0], precond_data[n][i][1]));
    strcat(answer, " ");
    strcat(answer, (precond_data[n][i][2] == true) ? "T" : "F");
  }

  return answer;
}

char *get_condition_string(char n, char m)                              /* function returns a char array      */
{                                                                       /* describing a particular condition  */
  char answer[5];                                                       /* (e.g. L1F) suitable for display on */
                                                                        /* the panel                          */
  if (n == 99 || m == 99)
    strcpy(answer, "");                                                 /* 99 is a default value */
  else {
    strcpy(answer, "L");
    strcat(answer, itoa(n));
    strcat(answer, (n == 0) ? "U" : (n == 1) ? "F" : "B");
  }

  return answer;
}

int get_fb_stat(char array[])
{
  int   i, counter = 0;

  if (time_step_counter == 0)
    return 0;

  for (i = 0; i < FB_STAT_SLICE && i < time_step_counter; i++)
    counter += array[i];

  return (counter * 100 / i);
}

void initialise_statistics(void)                                        /* Called once, at the start of the */
{                                                                       /* simulation                       */
  char  i, j;

  strcpy(info[0], "0");
  strcpy(info[1], "0");
  strcpy(info[2], "0");
  strcpy(info[3], "");
  strcpy(info[4], "0");
```

```
  strcpy(info[5], "0");

  for (i = 0; i < 4; i++) {
    for (j = 0; j < 2; j++) {
      precond_data[i][0][j] = (j == i) ? 99 : 0;
      monitor_data[i][j] = 99;
    }
    monitor_clock_data[i] = 0;
    for (j = 0; j < 3; j++)
      strcpy(behaviour_info[i][j], "");
  }

}

void swap(int *x_1, int *y_1, int *x_2, int *y_2)
{
  int temp_x, temp_y;

  temp_x = *x_1;
  temp_y = *y_1;
  *x_1 = *x_2;
  *y_1 = *y_2;
  *x_2 = temp_x;
  *y_2 = temp_y;
}

char *itoa(int n)                                   /* Converts an integer to a char array      */
{
  int i;
  int negative = FALSE;
  char count = 2;                                   /* length of returned string is at least 2 */
  char string[10];                                  /* (including \0 at end)                    */

  for (i = n; ((i /= 10) != 0); count++);           /* count number of digits                   */

  if (n < 0) {                                      /* to make space for the minus sign         */
    count++;
    n = -n;
    negative = TRUE;
  }

  for (string[--count] = '\0'; count > 0;) {
    string[--count] = (count == 0 && negative) ? '-' : n % 10 + '0';
    n /= 10;
  }

  return string;
}
```

Appendix D

Program Code for the Microcontrollers

```
; STAGE 1 ************************************************************
; Output values onto Port B pin 1. Cycle through a series of three
; pulse widths to drive the servo through a cycle of three positions.
;
; At 4MHz, instruction cycles take 0.001 ms. The RTCC prescaler is
; off, and the RTCC starts at 256-(32-2)=226, giving an interrupt
; every 0.032ms.
;
; Pin Outs
;       Bit     Port B
;       0       unused
;       1       output to servo
;       2-7     unused

        include "c:\picsim\picreg.equ"

        OPTIONS EQU     81h
        start1  EQU     0Ch             ; duration of hi signal
        count1  EQU     0Dh             ; 1ms timer
        signal  EQU     0Eh             ; 0 = low, 1 = high
        ct20a   EQU     0Fh             ; 20ms timer, LSB
        ct20b   EQU     10h             ; 20ms timer, MSB
        sending EQU     11h             ; signal high flag
        count_s EQU     12h
        count_m EQU     13h
        count_l EQU     14h

        pos0    EQU     D'31'           ; pos0=(31x.032)=0.99ms
        pos1    EQU     D'47'           ; pos1=(47x.032)=1.50ms
        pos2    EQU     D'63'           ; pos2=(63x.032)=2.02ms
        strt20a EQU     D'113'          ; signal every
        strt20b EQU     D'3'            ; (2*256+113)x.032=20ms

        rtstrt  EQU     D'226'          ; for a .032ms interrupt
                                        ; (226 = 256 - (32 - 2))

        ORG     00h
                GOTO    init            ; initialisation

        ORG     04h                     ; interrupt routine
                BCF     INTCON, GIE     ; global interrupt disable
                MOVLW   rtstrt
                MOVWF   RTCC            ; reset RTCC counter
                BCF     INTCON, RTIF    ; clear RTCC flag
                DECFSZ  ct20a, 1        ; update 20ms timer
                GOTO    update1         ; if 20ms not expired
                DECFSZ  ct20b, 1
                GOTO    set20ct
                MOVLW   strt20a         ; reset counters
                MOVWF   ct20a                   MOVLW   strt20b
                MOVWF   ct20b
                MOVF    start1, w
                MOVWF   count1
                BSF     sending, 0      ; set sending flag
                CALL    sendhi
                BSF     INTCON, GIE     ; global interrupt enable
                RETFIE

        update1 BTFSC   sending, 0      ; is signal high?
                CALL    cont1
                BSF     INTCON, GIE     ; global interrupt enable
                RETFIE

        set20ct MOVLW   h'FF'
                MOVWF   ct20a
                GOTO    update1

        cont1   DECFSZ  count1, 1       ; update 1ms timer
                RETURN
                BCF     sending, 0      ; clear sending flag
                CALL    sendlo
                RETURN

        sendhi  MOVLW   1h
                MOVWF   signal
                CALL    send
                RETURN

        sendlo  CLRF    signal
                CALL    send
                RETURN

        send    MOVLW   TRISB
                MOVWF   FSR             ; indirect to TRISB register
                MOVLW   b'00000000'     ; set all Port B pins to o/p
```

```
        RETURN
send2   BCF     Port_B, 1
        RETURN

A2D     BSF     Port_B, 0       ; for debugging
        CLRF    ADCON0          ; disable A/D conversions
        BCF     INTCON, ADIE    ; disable A/D interrupts
        MOVLW   TRISB
        MOVWF   FSR             ; indirect to TRISB register
        MOVLW   b'00000000'     ; set all Port B pins to o/p
        MOVWF   0h
        MOVLW   b'00000011'
        ANDWF   Port_B, 1       ; clear bits 2-7 of Port B
        MOVF    ADRES, w        ; read A/D conversion result
        MOVWF   result
        ANDLW   b'11111100'     ; mask out bits 0 and 1
        IORWF   Port_B, 1       ; write to bits 2-7 of Port B
        CALL    config          ; reconfigure Port A in case
                                ;   of noise interference
        BCF     Port_B, 0       ; for debugging
        RETFIE

ORG     50h
init    CLRF    sending         ; clear sending flag
        MOVLW   OPTIONS
        MOVWF   FSR
        MOVLW   b'01001000'     ; set prescaler off (1:1)
        MOVWF   0h
        MOVLW   strt20a         ; initialise 20ms timer
        MOVWF   ct20a           ;       "
        MOVLW   strt20b         ;       "
        MOVWF   ct20b           ;       "

        CALL    config          ; Configure Port A correctly
        CLRF    ADCON0          ; disable A/D conversions
        MOVLW   b'10100000'     ; Enable RTCC interrupts
        MOVWF   INTCON

        MOVLW   rtstrt
        MOVWF   RTCC
        MOVLW   pos0
        MOVWF   count1          ; initialise 1ms count
        BSF     sending, 0
        CALL    sendhi          ; send initial hi signal

loop    MOVLW   pos0            ; loop changes value of
        MOVWF   start1          ; start1 every 1.5 seconds
        CALL    wait
        MOVLW   pos1
        MOVWF   start1
        CALL    wait
        MOVLW   pos2

        MOVWF   start1
        CALL    wait
        GOTO    loop

wait    MOVLW   D'10'
        MOVWF   count_l
        BSF     ADCON0, ADON    ; enable A/D convertor
        BSF     INTCON, ADIE    ; enable A/D interrupts
        BSF     ADCON0, GO      ; begin an A/D conversion
        CALL    time_l          ; wait for ~1 second
        RETURN

time_s  NOP                     ; each NOP is 1 cycle
        NOP
        NOP                     ; running at 4MHz,
        NOP                     ; each cycle takes .001ms
        NOP
        NOP
        NOP                     ; there are 7 NOPs
        DECFSZ  count_s, 1      ; this takes 1 or 2 cycles
        GOTO    time_s          ; this takes 2 cycles
        RETURN                  ; result = wait count_s * 10 cycles
                                ; (i.e. count_s * .01ms (+.002ms))

time_m  MOVLW   D'99'
        MOVWF   count_s
        CALL    time_s          ; wait for .992ms
        NOP
        DECFSZ  count_m, 1      ; the time_m loop takes 9 cycles
        GOTO    time_m          ; (i.e. .008ms at 4MHz)
        RETURN                  ; result = wait count_m * 1ms

time_l  MOVLW   D'100'
        MOVWF   count_m
        CALL    time_m          ; wait for 0.1s
        DECFSZ  count_l, 1
        GOTO    time_l
        RETURN                  ; result = wait count_l * .1s

config  MOVLW   TRISA
        MOVWF   FSR
        MOVLW   b'00011111'     ; All Port A pins are i/p
        MOVWF   0h
        MOVLW   ADCON1
        MOVWF   FSR
        MOVLW   b'00000000'     ; All Port A pins analogue
        MOVWF   0h
        RETURN

;
END
```