Rover walking: a neuroendocrine controller for switching between rolling and walking locomotion

Final Report for CS39440 Major Project

Author: James C Finnis (jcf1@aber.ac.uk)
Supervisor: Dr. Mark Neal (mjn@aber.ac.uk)

5th May 2014
Version: 1.2 (Release)
Wordcount: 19962

This report was submitted as partial fulfilment of a BSc degree in Computer Science (G401)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Acknowledgements

I am grateful to Mark Neal for his supervision and suggestions through this project, and to Elio Tuci for helpful guidance in the mid-project demonstration.

I'd also like to thank Colin Sauzé and Hannah Dee for helpful comments throughout the process; Mark, Dave Price and Dave Barnes for their kind assistance in setting up the Vicon, Lawrence Tyler for help with using the Vicon, and finally my wife Catrin for useful criticism, advice and proofreading.

Abstract

Power management is an important issue in planetary rovers. In many cases the largest power expenditure is locomotion. In addition, planetary surfaces are extremely challenging to drive across, particularly on powdery slopes. Often rovers can become stuck, expending large amounts of power to escape (or never escaping at all). Therefore, novel locomotion modalities must occasionally be employed to both save power and get the rover out of trouble.

In this project, novel "walking" gaits are described for an ExoMars Concept-E locomotion prototype. These are compared with two different plain rolling speeds. The metrics used are selected based on prior work in the literature, theoretical considerations of their relationship to power usage, and on statistical analyses of their ability to consistently distinguish between and rank the different gaits.

The results of tests of all gaits at different inclinations, and under different surface conditions (i.e. surface packing) are presented, and show that rolling at a relatively high speed is generally most efficient in all cases, although on some steep, loosely packed runs the rover became stuck because of shearing forces acting on the soil.

The work demonstrates that rolling at a lower speed is inefficient because the motors were running near their stall speed. However, an analysis of the failure modes of the walking gaits shows that it is possible than an alternative low-level control regime may improve their efficiency markedly.

Furthermore, although a fast roll is generally more efficient, an artificial neuroendocrine system is presented which switches between the slow rolling gait and one of the two "walking" gaits, the input for which was the current. This was found to work well — the rover switched to the "lurching" mode when on the slope, and back again once on the flat.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Background & Objectives

In this section, the need for improved power management techniques on planetary rovers is outlined, and some existing work using artificial endocrine systems is described (with a brief description of such systems).

The possibility of increasing power efficiency over difficult terrain using the different locomotion modalities available to the ExoMars rover is then discussed, and the idea of using an AES to switch between the modalities — the "gaits" — is introduced.

Following this, the problems of measuring gait efficiency and finding a switching algorithm are discussed.

1.1 Background

Planetary rovers, such as the Mars Exploration Rovers *Spirit* and *Opportunity* and the Mars Science Laboratory rover *Curiosity*, operate in an extremely constrained power environment. The MSL has circumvented this problem by using a costly (and controversial) radioisotope generator [32], which provides a continuous 125W of electrical power from 2000W of thermal power (which is also used to heat the rover's components, protecting them from the harsh temperatures on Mars).

In contrast, the MER rovers and the future ExoMars rover use solar power to recharge batteries [8]. The solar panels for the MER provide about 140W for roughly four hours per day, some of which must be used for heating, and over time both batteries and solar panels degrade significantly. In addition, the amount of solar energy varies greatly during the Martian year, with possible dust storm activity decreasing it still further.

The latitude of the rover also determines the solar energy availability, with high latitudes being effectively "out of bounds" for a solar-powered mission: *Spirit* and *Opportunity* were limited to the equatorial regions of Mars between 10°N and 15°S.

Therefore, power management on a solar-powered rover is of paramount importance. If the available power can be utilised more efficiently, it may be possible to extend the range of the rover, increase the lifetime of the mission, do more science in the available time and even increase the range of available landing sites.

1.1.1 ExoMars and the ENDOVER project

The author has previously worked closely with Mark Neal and colleagues at EADS on the ENDOVER project (Mars Rover Autonomy using Endocrine based Algorithms). Using a mission simulator based on both solar energy availability information gleaned from the MER mission, and a detailed power usage model of the ExoMars rover, this project studied the possible gains from using an artificial endocrine system.

1.1.1.1 Artificial endocrine systems

Artificial Endocrine Systems (AES) are a relatively new technique in biologically inspired computing, in which Aberystwyth has taken a pioneering role [27, 34, 36].

In the biological endocrine system, hormones are released from endocrine glands in response to neural stimuli, and remain in the body after release with their concentrations decaying over time. These hormones modulate the behaviours of cells around the body which have the appropriate receptors. In many cases, these cells are neurons, and in these cases a hormone causes the neuron to respond differently. This provides two new features: a *broadcast* model of communication, and a *memory* — a stimulus may continue to have an effect on the system after it has ceased, for as long as the concentration of the relevant hormone remains significant.

In the prototypical AES as described in [36], the hormones exist as global scalar values which modulate the connection weights of an artificial neural network (ANN). This is shown in Figure 1.1, which assumes only a single hormone exists in the system.

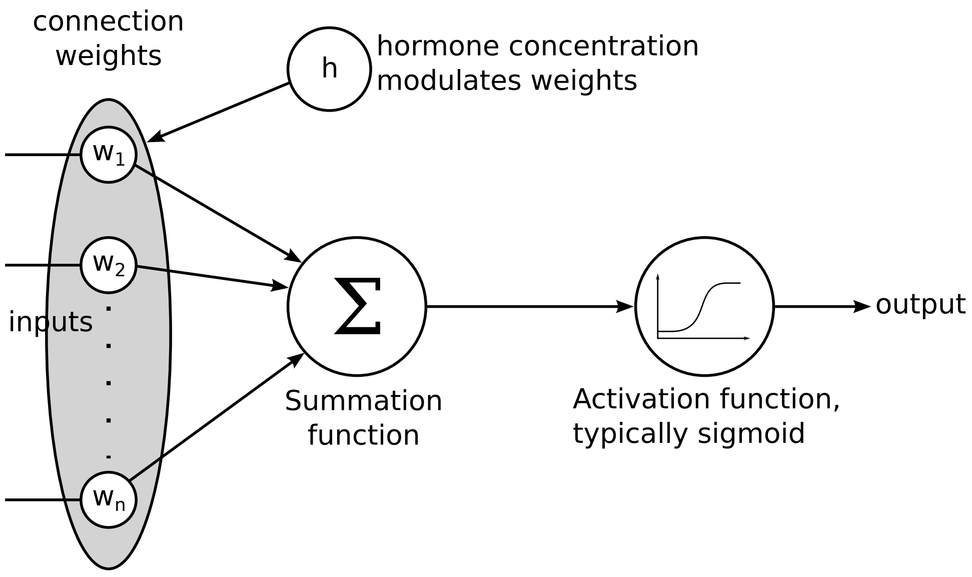

connection weights
h
hormone concentration modulates weights
w_1
w_2
inputs
w_n
Σ
Summation function
Activation function, typically sigmoid
output

Figure 1.1: A simple single-hormone artificial (neuro)endocrine system, shown modulating the connections of a single neuron.

This simple system has some powerful new properties compared with ANNs, the most notable of which is the memory effect mentioned above. In a standard ANN, the output of the network is a function of the input alone; whereas in an ANN+AES, the output is a function of the input and all the previous hormone stimuli. Applications of AESs include:

- short-term global behaviour modulation, such as modifying the behaviour of a robot in a hazardous situation or environment;
- long-term modulation, such as dealing with low power conditions [34];
- homeostasis — maintaining the state of a system by regulating internal variables [2];
- planning [36];
- cyclic behaviour, where a periodic function controls hormone release.

In the case of ENDOVER, remarkable efficiency gains were seen in both the duration of the mission and also in the management of the battery charge levels [28], showing that using artificial endocrine systems to change the behaviour of a rover during a mission can be a useful approach to power management.

1.1.2 Further work into power balancing

The author then went on to perform independent research into artificial endocrine systems[1], constructing a system to balance power usage across a wheeled rover [15]. This approach demonstrated that AES techniques can be useful in rover control over short timescales, as well as in mission planning.

1.1.3 Rover design and the possibility of "walking"

Most existing planetary rovers (including the MER and MSL rovers) are built around a"rocker/bogie" system as shown in Figure 1.2.

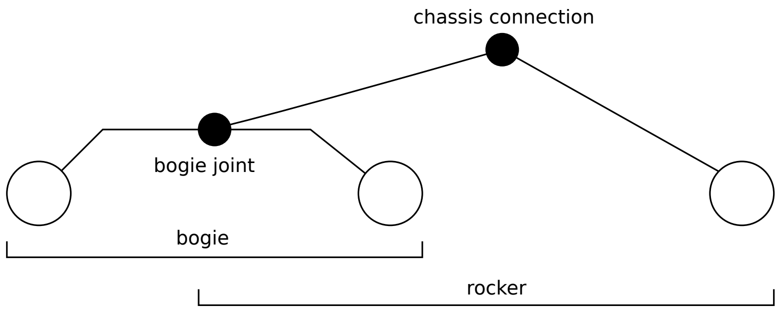


Figure 1.2: The rocker-bogie configuration currently favoured by NASA.

In this system, two wheels on each side are on a bogie, which is connected to the third by a rocker connected to the chassis. The two sides are connected by a differential, such that when one side's rocker rotates, the other rotates in the opposite direction. This system helps equalise the loading across the rover both on transit and during landing, where significant forces impact the suspension [19], but requires complex mechanics which are prone to failure. The ExoMars "Concept-E" system provides even loading through constant contact of all wheels with the surface, without a cross-linked differential [24, 29].

[1]Strictly speaking, the system constructed in this work was an artificial paracrine system, modelling how closely-connected tissues communicate within an organism.

Concept-E is a "three-module concept": there are three independent suspension modules with two wheels each, each module having a central freely rotating pivot point [24] as shown in Figure 1.3. Each of the six wheels has three degrees of freedom: drive, steer and lift as shown in

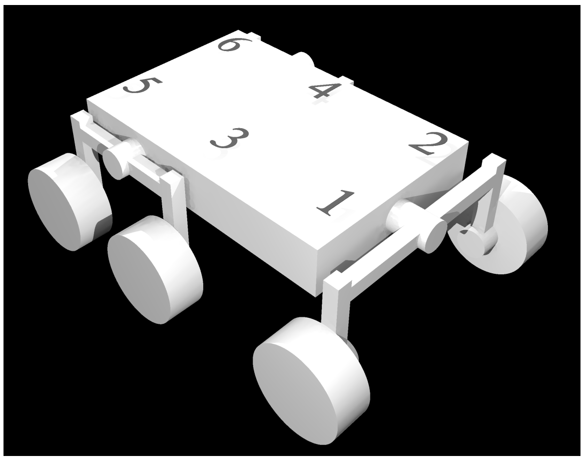


Figure 1.3: Configuration of the suspension modules and wheels in the ExoMars Concept-E chassis, with wheel numbering.

Figure 1.4.

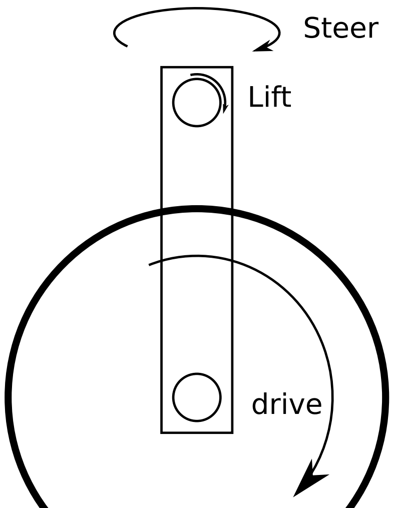


Figure 1.4: The rotation axes for each degree of freedom on each wheel in the ExoMars Concept-E chassis

- The **drive** motors rotate each wheel around its axis, rolling the rover across the terrain
- The **steer** motors rotate each wheel around the vertical axis, allowing the rover to turn or (if all wheels are rotated to the same angle) roll in a "crab" fashion
- The **lift** or **walking** motors rotate each wheel's leg at the point where it meets the chassis, allowing the leg to be swung forwards and backwards. This functionality was originally

intended to allow the legs to be stowed during the rover's flight to Mars, minimising the space taken in the spacecraft, but is still available after landing.

More details of the rover are available in Appendix A.

1.1.4 "Walking" and power efficiency

It is probable that simply rolling on all six wheels limits the rover to relatively flat and well-packed soil, because of the power expense of trying to roll through loose or slipping soil. In addition, there is a considerable risk to the rover of becoming stuck in such soil, as eventually happened to the *Spirit* rover. The ExoMars rover, however, has the ability to rotate its legs parallel to the rover body, permitting the development of additional locomotion modalities.

One example of such a modality demonstrated on a four-wheeled rover is "inching": a two-phase "gait" in which the forward wheels roll forwards while the back wheels remain static, after which the back wheels roll forward while the front wheels remain static. This gait shows an very large increase in tractive force (approximately double) caused by a "unified deep soil mass" underneath each static wheel [26].

Little work has been done on such "hybrid" gaits on six-wheeled rovers such as ExoMars — most research on walking with wheeled-legged robots has concentrated on rovers which are able to lift their legs above the surface to swing them forwards [21], thus emulating the gaits of legged robots.

1.1.5 Implementation of gaits

Biologically, gaits are thought to be the result of neural structures termed "central pattern generators": collections of neurons which produce a cyclic patterned output even when disconnected from any inputs. Evidence for CPGs in the spinal cord of vertebrates is discussed in [11], which describes how a CPG generates "alternating activity in groups of flexors and extensors." It is likely that a successful gait system will, in some way, be a model of biological CPGs.

Biological evidence for CPGs shows that there appears to be a number of such systems associated with each limb. This implies achieving a gait may require an interlocking system of "machines", each producing a different behaviour and modulating each others' behaviour (a "behavioural" system in Wettergreen and Thorpe's classification [37]).

While an attempt was made to implement gaits using a simple sequence of motor commands, the solution eventually arrived at in this study was a such a behavioural system, built around Brooks' subsumption architecture [3]. This is a bio-inspired system, given how closely it resembles the arrangement of CPGs postulated in vertebrates.

1.1.6 Different efficiencies at different inclines

It is likely that different gaits (including plain rolling) may have different efficiencies in different circumstances: the most simple example being the incline along which the rover is driving.

Other parameters which could have an effect include the condition of the surface, the number and type of solid obstacles (e.g. small boulders) and the lateral inclination. However, the scope of this project is necessarily limited by time and the available facilities.

Therefore it was decided that an investigation be made into the relative efficiencies of such gaits — first looking into possible metrics — at different inclines, and then an attempt be made at constructing a controller to switch between them as required. However, an attempt was made to perform the experiments on “packed” and “loose” soil, although no suitable equipment for measuring this quantitatively was available.

1.1.7 Summary

In this section, we have discussed power management in planetary rovers and compared the wheel configurations of such rovers. We suggest that the “walking” motors of the ExoMars rover may be suitable for executing “gaits” which have improved efficiency over difficult terrain. We will now analyse the problems involved in generating these gaits, evaluating them, and switching between them automatically.

1.2 Analysis

This section outlines the end goal of the research, and the subsidiary goals which needed to be achieved. This is followed by a brief discussion of some of the issues involved, which informed and limited the research questions. The research questions themselves follow.

1.2.1 Initial analysis

The ultimate goal was to construct a rover which can switch between gaits, selecting the most efficient gait for the circumstances.

Given the limited time and facilities available, “circumstances” were limited to a two terrain parameters: inclination and soil packing. Inclination is easy to vary: the Planetary Analogue Terrain Laboratory (see Section 3.2) has a sloped section for the rover to traverse, which has a roughly sigmoid profile permitting a large range of inclinations (see Figures 4.2 and 4.1). Packing is more difficult, because no facilities were available to measure this, so a simple binary value, “packed” or “loose” was selected.

In order to switch between gaits, gaits needed to be constructed and analysed for efficiency; therefore suitable efficiency metrics needed to be found.

1.2.2 Finding an efficiency metric

Metrics used by most existing studies are based on wheel slip — how much the rover wheels rotate, compared with how much the rover is moving [10,40]. However, this may not be the best measure to use on a planetary rover because it requires a highly accurate ground truth for the velocity. Most rovers rely on visual odometry, which can achieve an reasonable level of accuracy in a feature-rich environment but has problems with featureless, sandy areas: the precise areas where wheel slip is likely and gait switching may have benefits [25].

Therefore another efficiency metric should be found, relying entirely on interoceptive sensors: sensors which measure the internal rover state rather than external properties [7]. This process was the subject of the first series of experiments. Nevertheless, metrics involving slip were also investigated.

1.2.3 Construction of gaits

Software to generate different gaits was created (including ordinary rolling). There are many techniques for constructing such gaits, from simple pre-programmed sequences to bio-inspired solutions, but the solution selected should be a good fit for control by an AES in the final system. For this reason, Rodney Brooks' subsumption architecture was used, as described in section 2.10.5.

1.2.4 Research questions

The analysis described above leads to the following research questions:

Question 1: *Is there a metric based on interoceptive sensor data which is a sensible measure of efficiency, is sufficiently accurate, and is practical on our ExoMars Concept-E chassis-based prototype rover?*

Question 2: *Is there a locomotion technique (or "gait") for the our rover, involving both the drive and walking motors, which is more efficient (under some metric) than plain rolling when used on an incline, on both packed and loose soil?*

Question 3: *If so, is it possible to design an artificial endocrine or neuroendocrine system, controlling a subsumption architecture, which will switch between locomotion techniques automatically as the given metric demands?*

1.3 Research method

Here, the overall method is described, with statements of principle. In summary, the process is:

- investigate how to construct wheel-walking gaits;
- construct gaits to investigate;
- investigate metrics for efficiency;
- evaluate the gaits, comparing them against each other and plain rolling;
- select a wheel-walking gait which is more efficient that rolling up an incline, based on the metric;
- construct a neuroendocrine[2] controller which selects between rolling and wheel-walking, in response to the same or a similar metric to that found by the previous stage;
- compare this combined gait with both rolling and the original gaits.

[2]The terms "AES" and "neuroendocrine system" are used in a largely interchangeable way in this document. The systems discussed are artificial unless stated otherwise, and AESs usually have a neurally-inspired component — although that presented in Section 2.11 uses activation functions rather than perceptrons.

1.3.1 Principles

It was decided that all experiments should take place on the rover. A very brief attempt was made to construct a rover simulator using ODE, but this was abandoned after it was realised that simulator would be extremely large and highly inaccurate given the complex nature of the soil/wheel interactions. Any data collected from such a system would probably be completely irrelevant to a real rover.

A very simple simulator was later incorporated into the C++ robot control library, associated with a graphical simulation of the rover, but this was only used to help debug the gait and AES systems — no actual data was collected.

Each experiment (with the exception of the final runs of the AES because of time limitations) was performed at least five times, in identical conditions.

1.3.2 Gait construction

An obvious issue is that there are many ways to implement a wheel-walking gait. A more thorough investigation needs to analyse the space of possible gaits in a methodical way, and evaluate each gait. With the time given, only two walking gaits were constructed and analysed, based on a study of the literature.

1.3.3 Gait experiments

Five runs of each gait (including the two rolling speeds) were performed on packed and loose surfaces, both along a flat section of approximately 3m and up a roughly sigmoid slope with a maximum incline of about 20°.

1.3.4 Evaluating metrics

The results of the experiments on the flat were correlated across runs using Spearman's rank coefficient, generating correlation matrices showing the consistency of each metric. Rank correlation was used because of the nonlinear and non-normally distributed nature of the metrics when measured against inclination. Unfortunately, the rover's sensorium provided only four useful metrics: the exteroceptive slip ratio, Yoshida slip (see Section 4.3.1) and current/velocity ratio; and the interoceptive current measurement.

The metrics were also analysed from a theoretical standpoint, and from the point of view of their deployability on an actual planetary rover (as alluded to in Section 1.2.2). Interoceptive metrics were correlated with the extroceptive metrics: interoceptive metrics which correlate strongly with a theoretically extroceptive metric are more useful than the (possibly undeployable) extroceptive metric.

1.3.5 Evaluating metrics and gaits on the slope

The aim here was to answer the first and second research questions — is there a metric which shows differences between the various gaits at different inclinations, and is there a point at which one gait becomes more efficient than another?

The performance of each gait on the slope was analysed with the best of the metrics selected in the previous section, in order to find which performed best under which circumstances. Because of the complex behaviour of the gaits across the different runs, much of this analysis was qualitative or non-parametric in nature — Spearman's rank coefficient was used to test the metrics for consistency, for example. Attempts were made to analyse the behaviour in some depth.

1.3.6 Constructing a switching controller

A switching controller was then added to the system, which allowed the rover to switch between gaits using an AES fed by a suitable metric. This was tuned manually, and demonstrated to work. Unfortunately there was insufficient time to analyse its performance fully.

Chapter 2

Technology and Engineering

This chapter will describe the software and experimental setup developed for the study, and will also briefly cover the existing technology used. In many cases technology previously developed by the author was reused, and in these cases this will be clearly stated, and the technology will be fully documented in an appendix.

2.1 Overview

This section will describe the main elements of the experimental setup:

- The rover onboard systems;
- a laptop connected to the rover, which received monitoring data and sent commands using a monitoring system previously developed by the author;
- the rover scripting language, which ran on the rover and which was run by the experimenter via an SSH session from the laptop;
- the motion capture system, which captured data separately from all other systems;
- the collate scripts used to pull the rover and motion capture data together.

The setup is outlined in Figure 2.1.

2.2 Rover systems

The “Blodwen” rover (Figure 2.2) used in the experiments is a half-size ExoMars Concept-E prototype (as introduced in Section 1.1.3). While the rover chassis, wheels and gearing are of outside manufacture, the rover’s control system was developed as part of the author’s previous work where it was used to study paracrine control (see Section 1.1.2) and prototype an AES-based mission scheduling system (Section 1.1.1). It is described more fully in Appendix A. A thorough description of the control system can be found at [14].

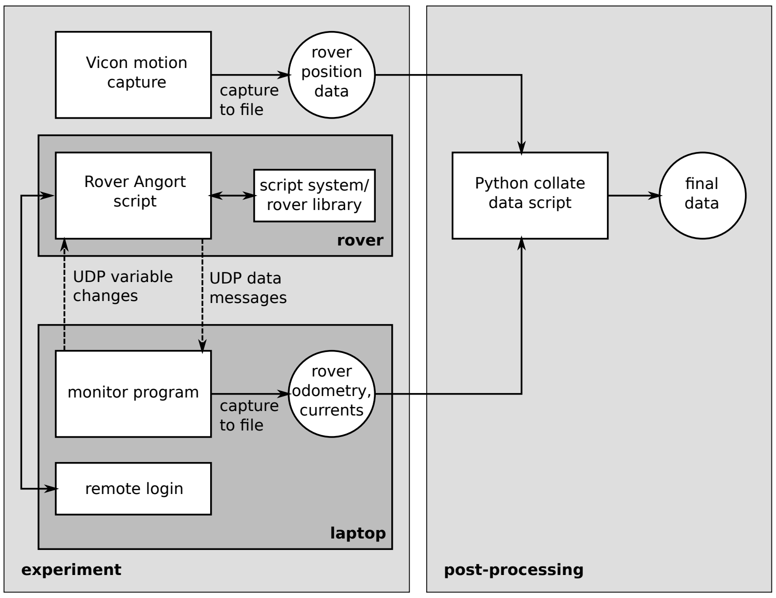


Figure 2.1: A diagram showing the main IT elements of the experimental setup, illustrating the data flows between them.

The rover carries a wireless router and two PCs, one for locomotion and one for possible future science (currently unused). A small laptop computer was connected to the rover's network, from which the experimenter logged into the rover and started the rover scripting language (see Section 2.6) with the appropriate script. Commands were then issued to reset and calibrate the rover, and start the experiment code.

The scripting language system sent data back to the laptop over UDP, where it could both be viewed and captured by the monitor system (see Section 2.3). The monitor could also send signals to the rover, and this was used to start the experiments. During the run, additional data was sent from the rover via the scripting language, giving details of additional variables pertinent to that experiment (e.g. hormone levels).

2.3 The rover monitor program

The monitor program was used to monitor the progress of the rover and also capture logging data. It is a medium sized application which developed during by the author in as part of the Tethys 2 project for EADS Foundation Wales, originally for the purpose of monitoring robotic boats. Because the requirements were somewhat fluid, it was made to be extremely configurable. It found extensive use during last year's research, both for its original purpose, and also for monitoring the rover during the experiments for a conference paper [15].

Use of the monitor program in the experiments made it easy to check the rover's internal states (driver unit temperatures, currents, wheel angles etc.), and gain an intuitive grasp of the behaviour of the gait algorithms as they were taking place (by showing current draw, etc.) Since configuration files already written for the rover, and outputting variables from the rover is so straightforward, it was obvious that the monitor should be used.

The program reads input on a UDP port, and allows it to be displayed by a number of widgets,

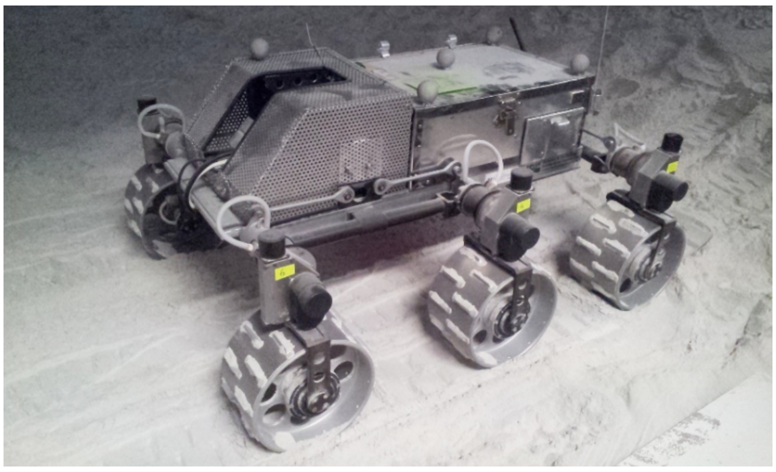


Figure 2.2: The rover "Blodwen" used in the experiments, built around a half-sized ExoMars Concept-E chassis.

including graphs, gauges, maps and status indicators. These are set up from a simple configuration file. A brief description of the data protocol used is given in Section B.1.

In addition, data can be sent in the opposite direction via a UDP port on the remote, so widgets can be specified which modify the state of the remote. These can be sliders, switches or momentary buttons.

The monitor can also capture the data it is monitoring to a log file. This facility was used to capture the experimental data coming from the rover.

To monitor a process, the user writes a configuration file specifying the variables which are expected in the input, how much previous data is to be stored for that variable (for graphing) and the widgets to be built. A simple annotated example of a configuration file is shown in Appendix B.2, as is the configuration file used for the current study (Appendix B.3).

A screenshot of the rover's monitor is shown in Figure 2.3.

2.4 Data fusion

During a run, rover position and orientation data was also captured by a motion capture system as described in Section 2.5. This system runs over a private network to which neither the rover nor laptop have access, therefore data was captured separately and fused later.

Therefore, the experiment was started (by setting a variable in the rover over UDP) and motion capture initiated at the same time. This was done simply by synchronous button presses — the timescales of changes in the data are sufficiently long that any slight ($< \sim$0.1s) difference in start times is insignificant. The `go` variable in the logged data, set by a packet sent from the monitoring system, indicated whether the experiment is running. Data where go $\neq 1$ was stripped out by the "collate" script which fuses the data (as described in 3.5).

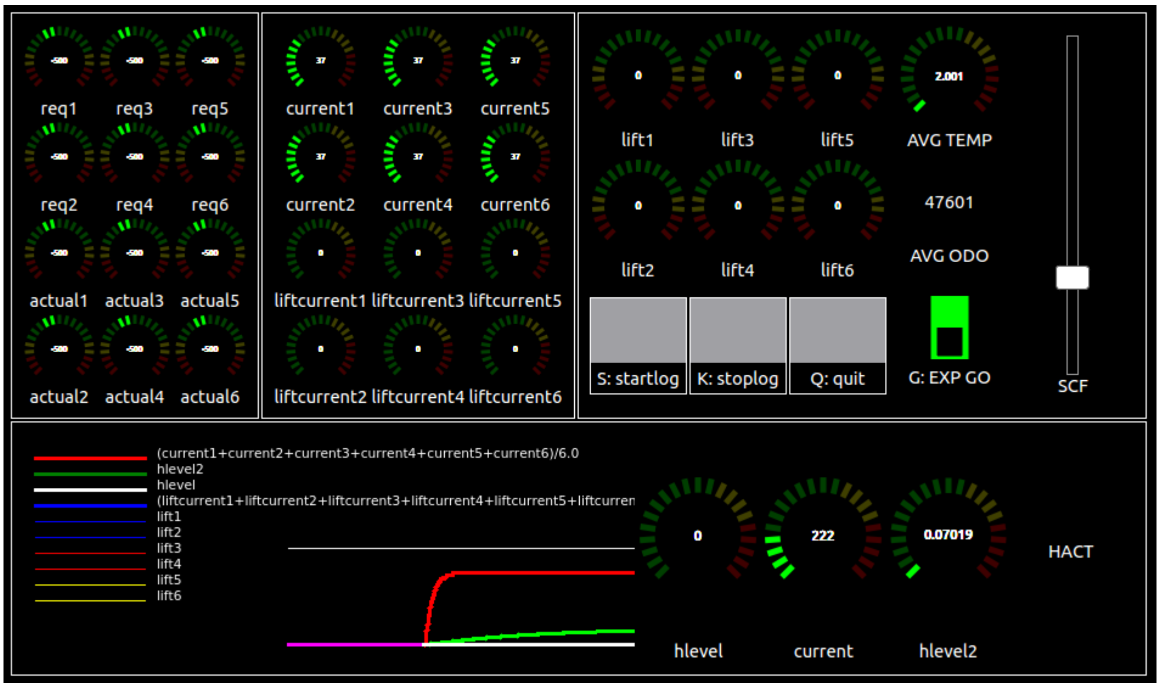


Figure 2.3: A screenshot of the monitor program, monitoring the rover during a (simulated) endocrine experiment.

2.5 Motion capture

The Planetary Terrain Analogue Laboratory (PATLab)[1] in which the experiments took place has a Vicon motion capture system. This consists of about a dozen cameras covering most of the "Mars yard" — the area covered in soil simulant. Initially, it was thought that this system would be unavailable or unsuitable for the required experiments, so some time was spent researching an alternative. This is discussed in Appendix J.

2.5.1 Introduction

Vicon are among the world leaders in motion capture systems, and a full Vicon MX system with 12 T-series cameras is installed in the PATLab. These can provide millimetre accuracy over the entire experimental area. This system (in common most motion capture systems) uses a set of active near infra-red cameras.

These cameras illuminate and capture images of the volume to be studied. Objects under study are marked with small, highly reflective spherical markers. Corresponding markers in the images are detected and the position determined using 3D vision techniques. Rigid objects made up of a set of markers can be labelled, and then the location and orientation of the centroids of these objects can be traced.

The Vicon system is on its own private network, with only one (Windows-based) computer connected. This is because of the large quantity of data which can produced, at up to 120 frames per second.

[1]See Section 3.2

2.5.2 Tracking an object

The first task was determining how to track the rover as a single object. Nominating the set of markers which made up the rover in the on-screen capture image was relatively easy, but as the rover moved, the object appeared to rotate wildly and the "lock" was often lost altogether. This was due to the initial positioning of the markers: to track the object successfully, it must have a unique appearance from all angles — otherwise, the vision algorithms are likely to get confused between different poses.

Therefore the markers were repositioned in a new configuration, forming a quadrilateral on the roof of the rover with one corner "pulled in", a marker in the middle, and an extra marker on the top of the rover's "flagpole." This arrangement is shown in Figure 2.4, with the image modified to show the markers' positions in red, with crosses in black. The true colour of the markers is silver-grey, as in Figure 2.2.

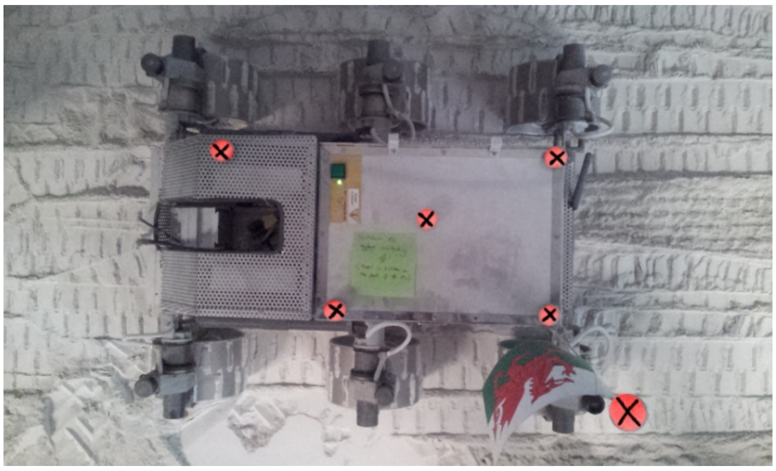


Figure 2.4: The positions of the fiducial markers on the rover, shown in red with crosses.

2.5.3 Recording the track and synchronising

The next task was recording the rover's track across the surface. Vicon's iQ software produces files in a proprietary format, which was of little use. There is, however, a comprehensive post-processing system which allows a set of filters to clean up the data and save in a variety of formats.

For these experiments, the existing example pipeline for cleaning the data was supplemented with a final stage to export to CSV. This final stage exports a 7 column CSV file describing the position and orientation (as Euler angles) of the rover, against frame number. The coordinate system of the Vicon is shown in Figure 2.5, with a crude sketch of the Mars yard and rover track.

Synchronising this data with the data captured from the rover itself was achieved simply by starting the capture at the same time as starting the rover running, and the data from each file were merged using a Python script (see section 3.5 below).

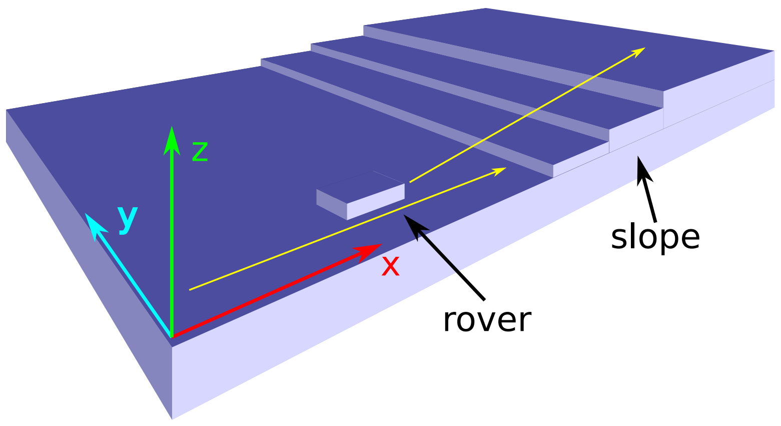


Figure 2.5: Coordinate system of the PATLab Mars Yard in the motion capture system. The flat and slope experiment tracks are shown in yellow.

2.6 Rover scripting system

The code for all the experiments was written in Angort, the language briefly described in Section 2.6.1 and more fully documented in Appendix I. Its purpose of each experiment's script is to generate a gait (or simple roll) and send a continuous stream of information to the monitor (see Section 2.3). In addition, each script manages starting the experiment in such a way that the capture can be synchronised with the start of data capture on the Vicon motion capture system.

2.6.1 Angort

Over the last year, a C++ library was built for high-level control (as detailed in Appendix A.7). However, C++ is not suited for controlling the rover in real-time. For this purpose, the author's own command and scripting language Angort was used [13].

Originally developed as part of research into language design and implementation, Angort has the advantage of being able to control and monitor the rover efficiently in real time — for example, to set the required speed of all the drive motors a simple function (often referred to, following the Forth usage, as a "word") could be defined:

```
:d |speed:| wheels each { ?speed i!drive };
```

which could then be used thus:

```
2500 d
```

to set the speed to 2500. To give a brief flavour of the language, here is an annotated version of the `d` function:

```
:d |speed:|      # begin defining a word with 1 parameter "speed", and no locals
    wheels       # push the predefined range "wheels": numbers from 1 to 6
    each {       # start an iterator over the range
        ?speed   # push "speed" onto the stack
        i        # push the wheel number onto the stack
        !drive   # set the speed of wheel i to the given speed
    }            # end loop
    ;            # end definition
```

Angort is a *concatenative* programming language, in which all expressions denote functions which manipulate data on a stack — similar languages include Forth, PostScript and Joy. For example:

```
4 5 + .
```

will stack the value 4, then 5, then remove the two values on the stack and replace them with their sum, before printing them[2].

2.6.1.1 Rationale for use

The first few experiments were done using Angort because a scripting language environment based upon it already existed. This was written to help control the rover in earlier research. However, it was initially envisaged that the more complex parts of the code, such as the gait systems, would be written in C++.

However, when the first attempts were made to develop a subsumption architecture in C++, the code rapidly became large and complex, primarily because of the need to pass differently-typed values (and nil values) through the system in a sensible way. An attempt was therefore made to write the system in Angort, which was surprisingly easy. Therefore Angort was used for all experimental development, with a few extra functions written in C++ (for communication with the monitor program and rover control).

While it would have been possible to use another scripting language such as Lua or Python, it would have taken prohibitively long to learn how to interface C++ to these languages. In addition, Angort's terse syntax made it suitable for real time remote control, which would have been more difficult in either candidate. A very brief introduction to the language is presented in Section 2.7 with reference to code used in the experiments. A more full reference is in Appendix I, and code snippets throughout this document are heavily commented.

2.6.2 Rover scripting environment

Angort is easily incorporated into a program, consisting of a very easy-to-use C++ library. Therefore building a simple application to control the rover was extremely simple, and was part of work undertaken during the industrial year. Much of this code was not developed for this study, but a list of the rover control words added is given in Appendix H.

2.6.3 Updating in the scripting application

An important aspect of the scripting application is how the rover data is updated, and how data is received from and sent to the monitor. When the interpreter is idle (i.e. at a command prompt waiting for input) a thread repeatedly calls `update()` on the rover object (see Appendix A.7). This sends a request to the rover hardware to fetch data on all sensors, ensuring the data objects are up to date. The `handleUDP` function is also called, which writes the rover data to the UDP monitor (the code for `handleUDP` is given in full in Appendix B.1).

[2]This is also a line of valid Forth, an early robot control language which was a major inspiration.

This ensures both that Angort command queries always have up-to-date data, and that the monitor program's data is kept up-to-date, when the system is idle. This is very important when remotely controlling the rover.

However, this automatic system is disabled during script execution, so the running script must periodically perform the sequence

```
update handleudp
```

This is typically done in every cycle, but takes a finite amount of time ($\sim$ 0.3s) because of the amount of data which needs to be transferred over the relatively slow serial line between the PC to the rover hardware, and between the master Arduino and the motor drives slaves over I^2C.

2.7 Experiment code

The rover scripting system has a file which is loaded automatically before any other script, called `script.ang`. This contains both remote control elements and elements common to all experiments. The source code for this file is given in full in Appendix D.1.

Some important words (Angort functions) for managing experiments are defined in this file: `expStart` and `expUpdate`. These are worth describing in some detail, from the point of view of both their functionality and familiarisation with the language, and this is done in Appendix D.2. In summary, however:

- `go` is a UDP property — syntactically identical to a global variable, it can also be set by a UDP packet from the monitor system. The EXP GO switch on the monitor sets this value, starting the experiment.
- `expStart` zeroes all motor positions and speeds, resets the odometry, and waits for the `go` property sent from the monitor to become true.
- `expUpdate` updates the rover data (which can take some time), polls for incoming UDP data, and sends UDP data to the monitor. It then returns true if the experiment has been halted, by checking the value of `go`.

2.8 The rolling experiment code

These two words are used to build the simple rolling experiment code, used to evaluate plain rolling locomotion across the surface. This is very brief:

```
:rollexp |speed:|    # one parameter: speed, no locals.

    reset calib      # reset the rover and send calibration data
    expStart         # wait for the experiment to start

    # once started, set the desired speed

    ?speed setdriveall

    # and loop, updating and handling UDP, until go becomes
    # false.
    {
        expUpdate ifleave
    }
    "Stopping.".

    0 setdriveall   # stop all drive motors
    0 setliftall    # centre all lift motors (just in case)
    0 setsteerall   # centre all steer motors (just in case)
;
```

The rover is first calibrated with `reset calib` (to set up the internal parameters) and the `setdriveall` word (defined in `script.ang`) sets the motors to the commanded speed, read from the parameter.

There follows the main experiment loop. Because driving requires no more commands to be sent to the rover, this simply calls `expUpdate` and leaves the loop if it returns true. Finally, a message is printed and all the motors are set to known positions.

2.9 Simulator

During gait development, it was found that bad gait code would often cause the rover to behave erratically and possibly dangerously. In addition, determining how the legs and wheels were behaving was very difficult. Therefore, the simulation built into the PC library was improved, and a new visual simulator using the same UDP data as the monitoring program was written.

2.9.1 Internal rover library simulator

Similar problems occurred during development of the rover, so a very crude simulation was built into the PC library last year. This is enabled by instantiating the library with an optional argument, which is done by calling the rover scripting application with the `-s` command line option. In simulation mode, the library will not attempt to communicate with the hardware.

The simulation is very basic — all motors move to their commanded speeds and positions with lag, using the simple exponential moving average: if the desired position is d and the actual position is a, then

$$a_t \leftarrow ka_{t-1} + (1-k)d_{t-1}$$

where k is the smoothing constant. Originally, k was the same for all motors, but this proved deceptive — desynchronisation of the alternate gait was not seen in simulation because of this. Therefore a modification was made, and k is now very slightly different for each motor. A crude simulation of current was also used in endocrine switching tests, where current is estimated as a linear function of the motor speed (for drive only).

2.9.2 Graphical simulator

It was still extremely difficult to debug gaits, because the there was no visual output from the simulator. Therefore, for this project, a few hours were spent writing a 3D simulation of the rover in Python and OpenGL. This receives the same UDP packets as the monitor system (see Section 2.3) and draws a small rover on the screen. In addition, it relays the same information to another port so that the monitor can still run. This system proved invaluable during development, both for debugging and safety reasons. Source code is in the electronic submission, in the `project/draw` directory.

A screenshot is shown in Figure 2.6.

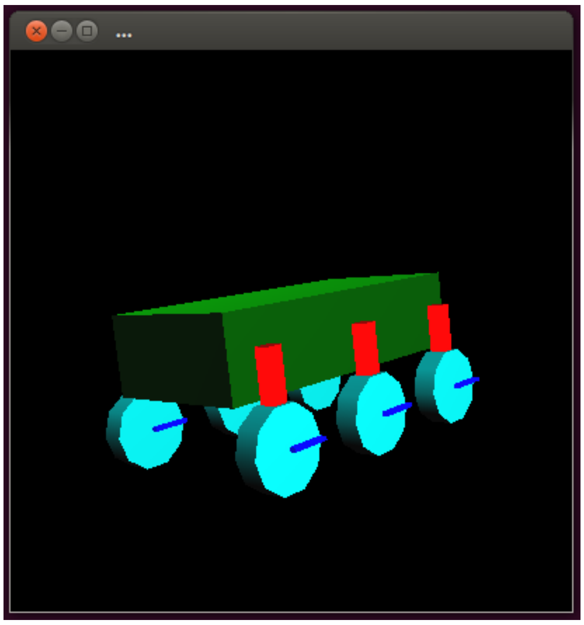


Figure 2.6: A screenshot of the graphical rover simulator.

2.10 Walking gait construction

This section describes how the walking gaits were constructed, the simple rolling "gait" already having been discussed in Section 2.8.

As has been mentioned in Section 1.1.4, little work has been done on walking gaits in rovers like ExoMars, which is required by its suspension to keep all wheels on the ground. Walking usually involves lifting some legs, moving them forward, then putting them down again, which ExoMars cannot do.

One possible approach is "inching" — repeatedly lengthening and shortening the wheelbase by rolling the back and front wheels forwards, as referred to in the aforementioned section. This achieved a remarkable improvement in tractive force, indicated by a large reduction in slip for the same amount of backward force applied by a drawbar [26]. The first gait developed for this project is "lurching" — a version of inching for a six-wheeled rover.

The other gait developed is the "alternating" gait, which is inspired by the "alternating tripod" used by some insects (and many legged robots).

2.10.1 Basic motions — “roll” and “push”

Both the lurching and alternating gaits are made up of two basic motions: **rolling** and **pushing**, shown in Figure 2.7.

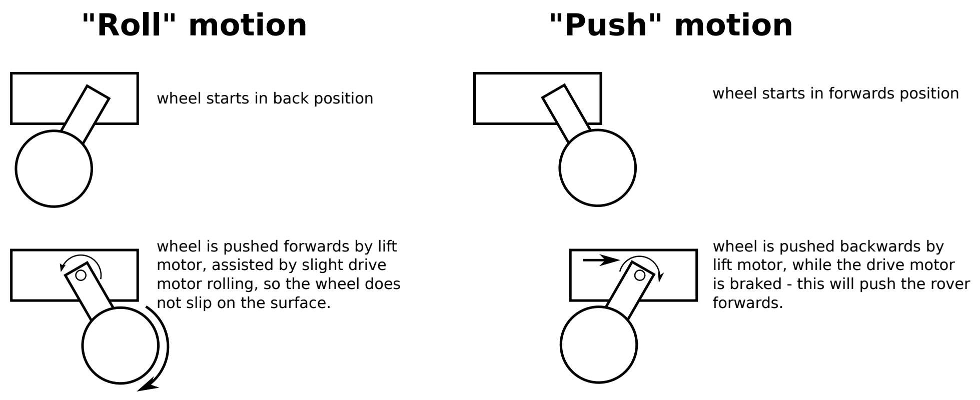

"Roll" motion
"Push" motion
wheel starts in back position
wheel starts in forwards position
wheel is pushed forwards by lift motor, assisted by slight drive motor rolling, so the wheel does not slip on the surface.
wheel is pushed backwards by lift motor, while the drive motor is braked - this will push the rover forwards.

Figure 2.7: The two basic motions of which the lurch and alternating gaits are comprised.

- **Rolling** consists of moving the leg forwards while rolling the wheel along the ground, so that the wheel rolls along the surface without slipping.

  This is achieved by setting the control cap of the lift motor to higher than usual[3], so that it will respond to an error with more torque, while setting the gains of the drive motor to very low. The result is that the drive motor provides just enough torque to keep the wheel rolling smoothly along the ground, while the lift motor actually moves the wheel forwards.

- **Pushing** simply consists of swinging a leg from a forward position, where the wheel is ahead of its normal position, to a backward position. It is the pushing action which exerts a force against the surface, moving the rover forwards.

  It is achieved by simply commanding the lift motor to move to the given position with the control system’s parameters set normally.

2.10.2 The “lurching” gait

This is the simplest gait to understand, and is shown in Figure 2.8. A video of this gait is also available[4], see Table E.1. This gait begins by pushing all the wheels into a backward position, so that the rover itself is slung forwards. Then each pair of wheels, starting at the front, rolls forwards. To complete the gait cycle, all the wheels are then pushed backwards. It is this last motion that gives the gait the characteristic “lurch.”

Because all the wheels push together the action is a simple rotation of the rover around the axis joining the wheel pairs, and none of the wheels actually change their positions. Instead, the rover body moves forwards.

[3]All motors are PID controlled, but the P and D gains of the lift and steer motors are currently zero, so motors are controlled by I-gain only. This integral term has two extra parameters: decay and cap, which determine the geometric decay and maximum magnitude of the error.

[4]`https://www.youtube.com/watch?v=ufw0CnLncjY`

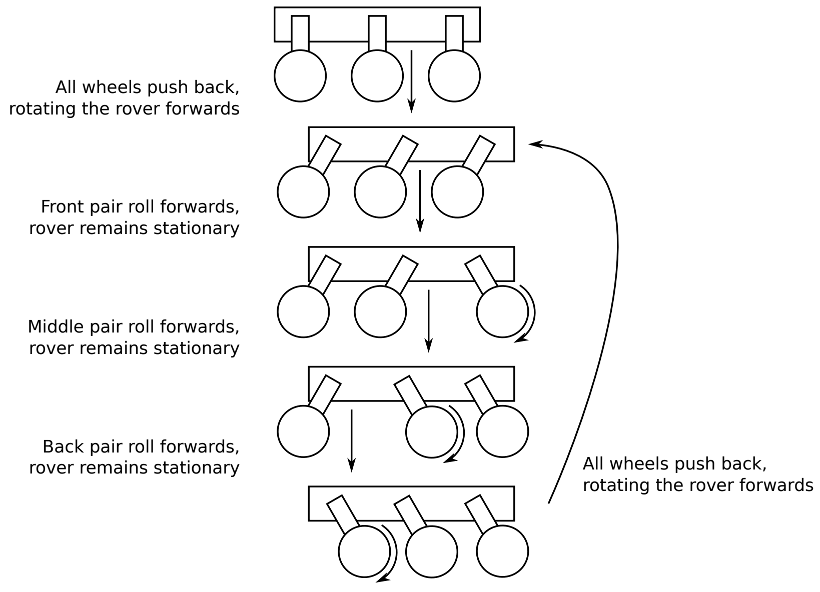


Figure 2.8: The lurching gait

2.10.3 Alternating gait

This is a rather more complex gait which is based on the commonly-used “alternating tripod” [1, 35], originally inspired by the gait of cockroaches (although cockroaches often use a more stable gait in which four legs are on the ground at one time [22]).

In the alternating tripod, the legs form two triangles — left-front and back with right-middle, and right-front and back with left-middle. One triangle is always on the ground providing support, while the other is being lifted and moved forwards. Thus, the animal or robot is always on a stable base. The action is shown in Figure 2.9.

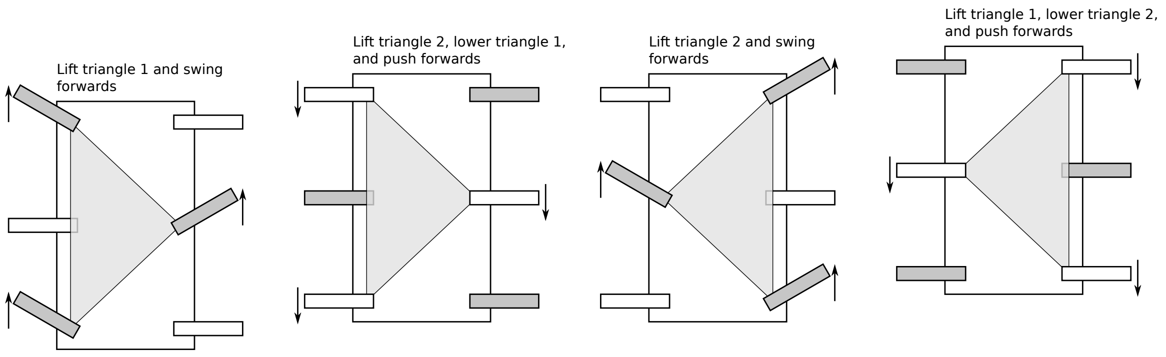

Lift triangle 1 and swing
forwards
Lift triangle 2, lower triangle 1,
and push forwards
Lift triangle 2 and swing
forwards
Lift triangle 1, lower triangle 2,
and push forwards

Figure 2.9: The alternating tripod gait used by some insects and many hexapod robots. Dark grey indicates that a leg is lifted and not in contact with the surface.

2.10.3.1 The rolling/pushing implementation

It is not possible to implement this gait directly on the Concept-D chassis because the suspension guarantees that all wheels are in contact with the ground, and the wheels cannot be lifted without moving along the surface. However, it is possible to approximate it using the two motions introduced above:

- First action: triangle 1 rolls forwards, while triangle 2 pushes backwards.
- Second action: triangle 2 rolls forwards, while triangle 1 pushes backwards.

This is shown in Figure 2.10. One complexity introduced is that the legs rolling forward must

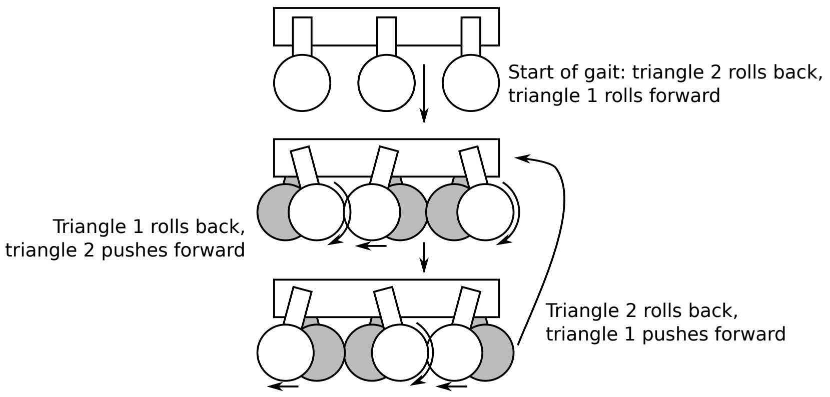

Start of gait: triangle 2 rolls back,
triangle 1 rolls forward
Triangle 1 rolls back,
triangle 2 pushes forward
Triangle 2 rolls back,
triangle 1 pushes forward

Figure 2.10: The alternating gait

move faster than the legs pushing backwards — those latter legs are pushing the body of the rover forwards, and the rolling legs must therefore move fast enough to move ahead of the rover body, and not merely keep up with it. A video of this gait is available[5] — see Table E.1.

2.10.4 Initial development

The simplest method for generating a gait was initially thought to be a table of desired positions at given times within a gait cycle. However, a movement may take different times to complete under different conditions. Therefore each step needs to wait for the previous step to continue — effectively turning the system into a finite state automaton.

An attempt was made to write this in C++, but rapidly abandoned as it was realised that both implementation and testing would be much more straightforward in Angort, by implementing each state as a hash containing entry, exit and update anonymous functions.

Although implementing the lurch gait was straightforward, attempts made to generate an alternating gait (see below) using a single state machine became extremely complex. It was rapidly realised that each wheel needed its own, independent state machine — and this was recognised as being a pre-existing paradigm in robotics: the Brooks subsumption architecture [4].

2.10.5 The subsumption architecture

This is a relatively early technique in reactive robotics, but one well-suited to the problem. This architecture describes a robotic system as a set of behaviours, each instantiated by a small network of finite state machines (augmented with timers and some local storage).

[5]`https://www.youtube.com/watch?v=BjRkYZ7-_gE`

The developer constructs each behaviour and tests it thoroughly before adding more layers of machines whose outputs can override, or "subsume," the outputs in more "primitive" layers.

This methodology has proven useful in gait generation [3], and has the important advantage of producing extensible systems: once the gait system has been constructed, modifying its behaviour with an AES should be a simple matter of adding another layer. In addition, it is also a biologically inspired system which provides a way of reconciling the continuously-valued AES with a system using discrete states, in an entirely biologically inspired way.

It is key to the architecture that an output may sometimes "subsume" another. For example, consider the very simple system in Figure 2.11. Here, if machine 2 is producing a nil output,

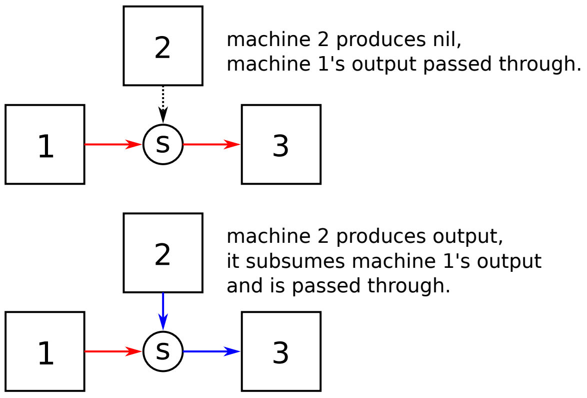

2
machine 2 produces nil,
machine 1's output passed through.
1
S
3
2
machine 2 produces output,
it subsumes machine 1's output
and is passed through.
1
S
3

Figure 2.11: Example of subsumption

machine 1's output is passed to the input of machine 3. However, if machine 2 does start to produce an output, then it will subsume that of machine 1, replacing it.

This is the mechanism which is used to add layers of functionality to a subsumption architecture, by adding new subsystems which subsume behaviour which has already been built and tested.

2.10.5.1 Possible alternative approaches

There are many alternatives — possibly an even better technique would have been to construct a neural network walker implementing a set of CPGs, modulated by an AES. However, the time taken to build such a system would be prohibitive for a project of this scale.

2.10.6 Implementing subsumption in Angort

Angort has first-order functions, so writing data structures containing elements of code is straightforward. This proved invaluable for the implementation of subsumption. The core of the subsumption architecture is in the file `experiments/sub.ang`, and is shown in full in Appendix D.2.3. Each augmented finite state machine is implemented as an Angort hash table, keyed on symbols (single-word strings, actually stored as integers internally for speed). Note that the file `experiments/test.ang` contains a set of unit tests for the subsumption system.

The items which must be present in a machine hash are:

- `` `name ``[6] — the name of the machine;
- `` `inputs `` — a hash containing the initial values of the machine's inputs, keyed by their names (also symbols);
- `` `outputs `` — a hash containing the initial values of the machine's outputs;
- `` `states `` — a hash containing the state functions keyed by the symbols `` `entry ``, `` `exit `` and `` `update ``.

Code for an example machine is shown in Appendix D.2.4. Typically, machines are written as functions to generate a machine of the required type (by stacking and returning the appropriately structured hash). These functions often take parameters such as wheel numbers.

Once a set of functions to create the required machine has been written, they must be linked together. This "plumbing" is done by a function to run all machines, then copy the appropriate outputs to the appropriate inputs of other machines. For a very basic 2-machine system this could be as simple as:

```
:run
    updateallmachines       # update all machines registered with initmachine
    `op1 ?machine1 readout  # read machine1's output 1
    `in  ?machine2 writein  # and write that value to machine2's input
;
```

but in a real system will be much more complex. Subsumption is done using the `subsume` function: the example in Figure 2.11 could be written as:

```
    `out ?machine1 readout  # stack machine 1's output
    `out ?machine2 readout  # stack machine 2's output
    subsume                 # machine 2 will subsume machine 1 if non-nil
    `in  ?machine3 writein  # and write to machine 3
```

This provides the ability to write complex gait systems, and then build on them with an AES or some other higher-level behaviour at a later date.

2.10.7 The lurching gait implementation

The architecture for the lurching gait is shown in Figure 2.12. This represents only one side of the rover; each side is independent and both are identical. Each side has the following machines:

- A sensor machine (labelled IN in the figure) which supplies actual drive speed and lift positions.
- An output machine (labelled OUT) which is supplied with required drive and lift values, and a configuration symbol. This can be either `` `roll `` or `` `std ``. If the former is specified, the motor control parameters are set for the rolling action, and in the case latter they are set to the standard values for pushing (see Section 2.10.1). Nil values will cause no commands to be sent to the motors, leaving the state unchanged.
- Two machines called `isback` and `isfwd` which returns boolean values, indicating whether the associated leg is fully back or fully forwards.

[6] A symbol is written in Angort preceded by a back-tick.

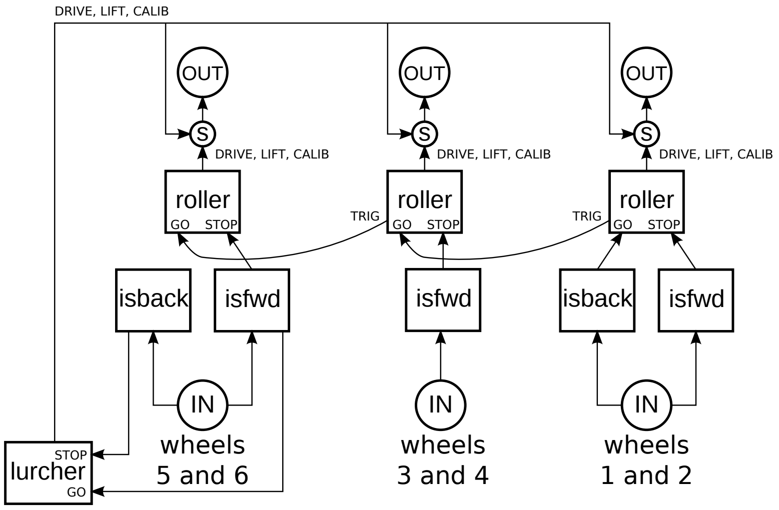


Figure 2.12: The subsumption architecture for the lurching gait (one side shown).

- A `roller` machine, which has two inputs: `'go` and `'stop`, and three outputs for drive speed, leg position and calibration setting (see the output machine above). An addition, there is a boolean output called `'trigger`.

  On receiving a non-zero `'go` signal, this will output control values for the "roll forward" motion. This will continue until the `'stop` input is true, whereupon it will set its `'trigger` output to true, wait for a short period, and return to its initial state (resetting the trigger).

Finally, there is a `lurcher` machine on each side which has two inputs and the three outputs for connection to an output machine, as with `roller`. When the `'go` input becomes true, this will output the "push backwards" motion until `'stop` becomes true.

Given the connections in Figure 2.12, this will cause the following sequence of events on each side (assuming the rover starts with all legs in the back position — see below):

- `isback` triggers the roller machines on wheels 1 and 2 (front wheels) to roll forwards;
- when this is complete, the front rollers stop and trigger wheels 3 and 4 (middle) to roll forwards;
- when this is complete, the middle rollers stop and trigger wheels 5 and 6 (rear) to roll forwards;
- when this is complete, the rear rollers stop and trigger the lurchers on each side;
- the lurchers subsume the roller signals causing all the wheels to push back, rotating the rover forwards.

It is interesting to note that control is decentralised, and each side's gait cycle is independent. This can be useful if one side becomes "bogged down" in soil: the other side can continue to move, eventually lifting the stuck side out.

Note that the sequence must be started by all wheels being moved to the back position — this is done by the experiment code:

```
:tst
    build    # build all the machines
    expStart # start experiment

    BACKANGLE setliftall # set the leg positions to back

    # main loop
    {
        run  # run the "plumbing" and update the machines
        expUpdate ifleave
    }
;
```

The full code for the lurch gait experiment is in Appendix D.2.5.

2.10.8 The alternating gait implementation

2.10.8.1 Attempts at decentralised control

Initial attempts to build this gait considered each wheel (front, middle and back) as a separate unit. All wheels would be initialised to one of the two alternating positions. Then, each wheel would push back if it found itself forwards, and roll forwards if it was in the back position. This is shown in figure 2.13. This led to severe synchronisation problems — a push action would start while the

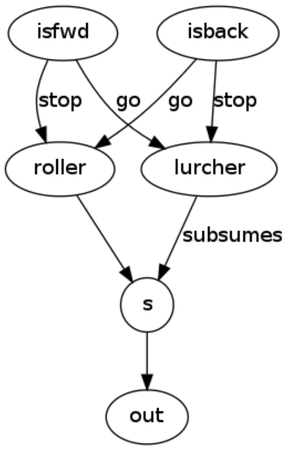


Figure 2.13: A single wheel's architecture in an early, non-working alternating gait.

roll on the far side was still static, because it takes time for the rolling action to build up speed. Therefore, an attempt was made to wait for the rolling to have fully started on the opposite wheel before the lurching began. This was done using a simple machine to detect a relatively high actual speed in the drive wheel, and is shown in Figure 2.14.

This appeared to work well in the graphical simulator (see Section 2.9), but failed dismally on the real rover: the pairs of wheels desynchronised rapidly, because the wheels move at slightly different speeds over the surface. Worse, if resistance caused a wheel to turn while in the pushing (lurching) action, it would trigger the lurch behaviour in its opposite number. This is clearly shown in Figure 2.15, which plots the erratic lift position of wheel 1 during this gait, along with the state numbers for the rolling and lurching machines for that wheel. In this run, lurching is being re-triggered in wheel 1 during its roll phase by slippage in the opposite wheel.

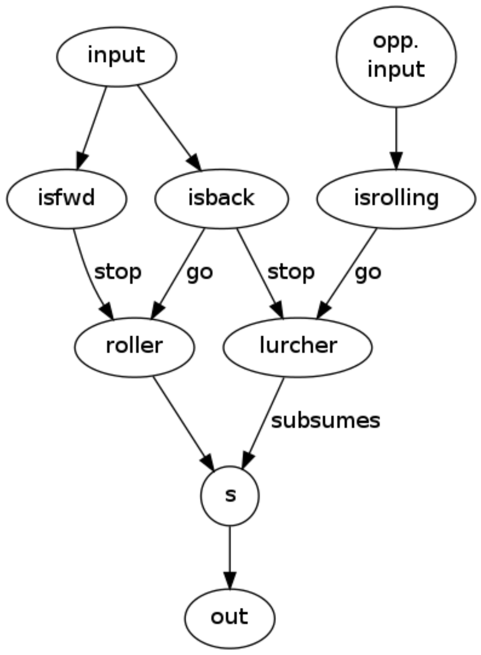


Figure 2.14: A single wheel's architecture in an early, non-working alternating gait with an attempt at synchronising the push to the opposing wheel's roll.

2.10.8.2 A more centralised solution

The solution required an architecture in which the two alternating tripods were synchronised with each other throughout the gait. This was done by only permitting a change in motion when both triangles are in the correct positions. The architecture for a single wheel within this system is shown in Figure 2.16. This uses similar machines to the lurching gait, configured in a different way: each wheel has a lurcher and a roller, but the roll forward is triggered when all wheels in the same triangle are fully back, and all wheels in the opposing triangle are fully forwards. Similarly, the lurch phase is triggered when all wheels in this triangle are fully forwards, and all wheels in the opposing triangle are fully back.

There are two differences in the roller machine:

- it is now configured to roll much faster, to be able to move the wheels in front of the rover while the rover is being pushed forwards;
- there is no longer a trigger output.

Again, the gait must be triggered by moving the wheels to an initial pose, so the experiment does this before the main loop starts. The full code is in Appendix D.2.6.

2.11 Artificial neuroendocrine system

In the next chapter, it will be seen that switching between a rolling speed of 500 and the lurch gait might provide benefits — roll-500 is more efficient than lurch on the flat, but lurch becomes more

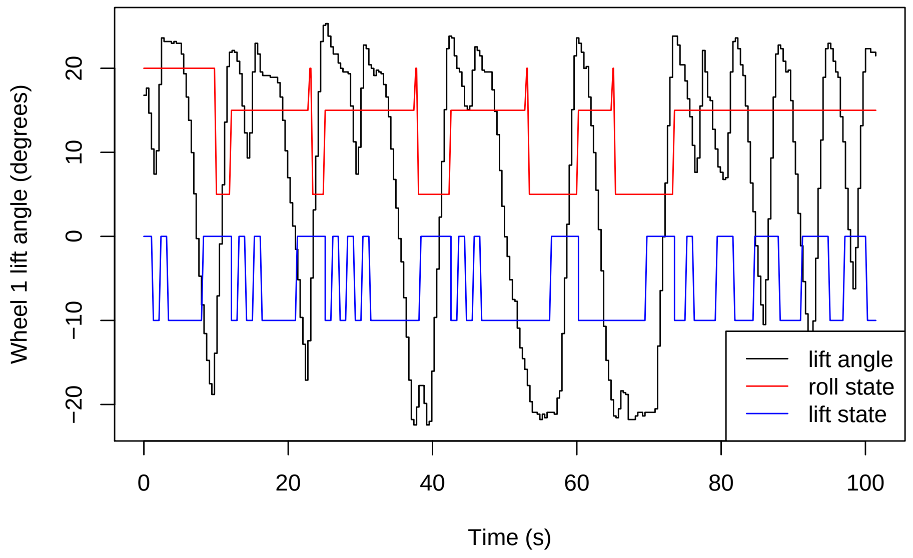


Figure 2.15: A plot of the lift position of wheel 1 during the second failed alternating gait, along with the state numbers for the rolling and lurching machines (high means that machine is active).

efficient on the slope. As mentioned above in Section 1.2.2, exteroceptive sensors which rely on the rover's location are not suitable in our setup, so we must rely on an interoceptive input to the AES. Current was chosen, based on the analysis in Section 4.6.

2.11.1 Design of the AES framework

The fundamental unit of the AES as implemented in this project is the `hormone` , which encapsulates the hormone level itself and the input and output perceptrons. In each hormone:

- A sensory perceptron detects stimuli. The output of this perceptron is multiplied by a "release rate" constant associated with the hormone.
- A limiting function is then imposed, which multiplies the release rate by $0.95 - h$ where h is the current hormone level. This ensures that the hormone's level saturates at 0.95.
- The value obtained is added to the hormone level. The hormone level is set to geometrically decay at a given rate.
- The hormone level is then sent through another perceptron to produce the output value.

In practice, both input and output perceptrons are bare activation functions: since there is only a single input no summation is required. This is shown in Figure 2.17. Each hormone has the following parameters:

- input sigmoid centre and width,

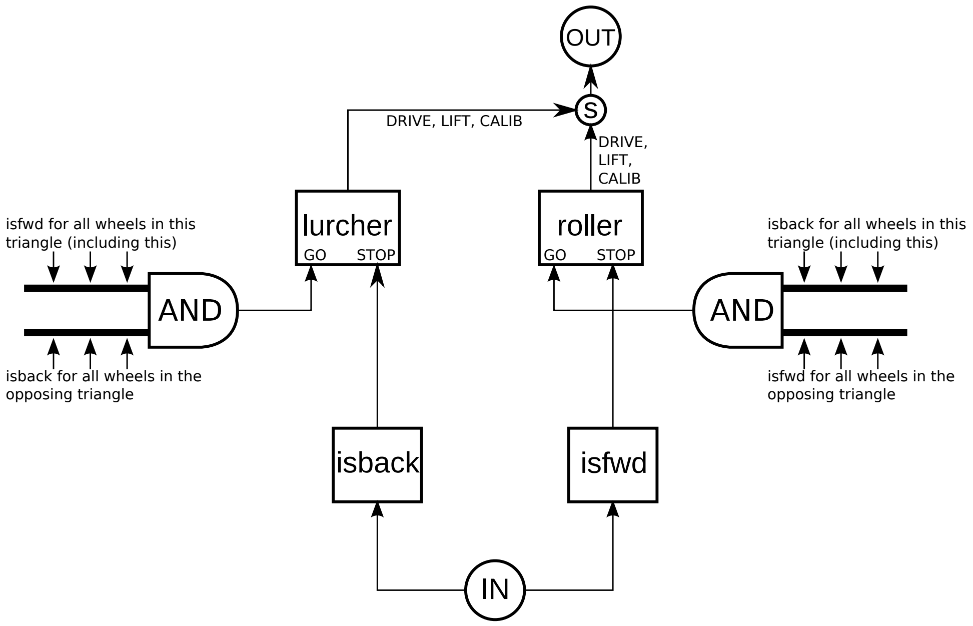

OUT
S
DRIVE, LIFT, CALIB
DRIVE, LIFT, CALIB
lurcher
GO STOP
roller
GO STOP
isfwd for all wheels in this triangle (including this)
AND
isback for all wheels in the opposing triangle
isback for all wheels in this triangle (including this)
AND
isfwd for all wheels in the opposing triangle
isback
isfwd
IN

Figure 2.16: Subsumption architecture for a single wheel inside the alternating gait.

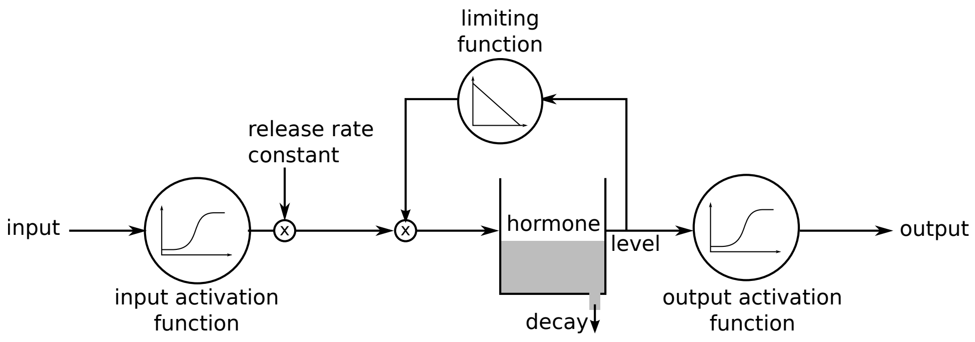


Figure 2.17: The architecture of a single hormone in the AES

- release rate,
- decay constant,
- output sigmoid centre and width.

Each sigmoid is defined by its centre c and width w, where c refers to the value for which the sigmoid output is 0.5 and w is the width of the region where the output is close to neither 1 nor 0. The empirical function for the sigmoid is then

$$\sigma(x, w, c) = \frac{1}{1 - (e^{0.1w(x-c)} + 1)}$$

This gives a transitional region where $\sigma(-w/2, w, 0) = 0.007$ and $\sigma(w/2, w, 0) = 0.993$.

A sigmoid for $w = 1, c = 0.5$ is shown in Figure 2.18. The source code for the AES framework in Angort is in Appendix D.2.7.

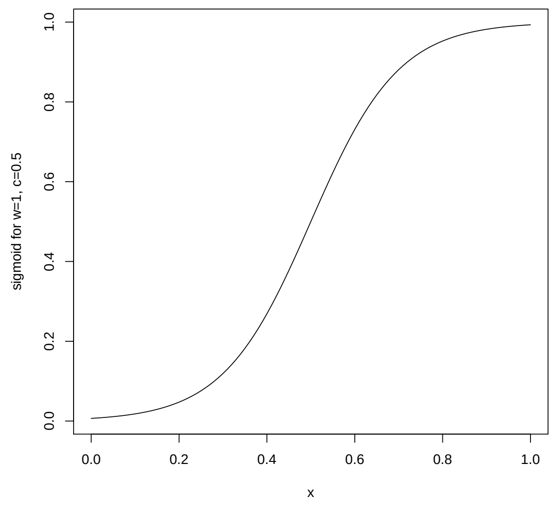


Figure 2.18: A graph showing the sigmoid function with centre and width, with a centre of 0.5 and a width of 1.

2.11.2 Implementation and results

Two implementations of the system were constructed, one with a single hormone, and one using a two hormone cascade.

In the single hormone system, the hormone was fed by the current. The width and centre of the input sigmoid function were set by close inspection of the current levels in the results of slope runs for the 500-roll and lurching gaits, such that the sigmoid was near 0 at the current level where 500-roll was most efficient, and near 1 where the lurching more most efficient. The output sigmoid was set to be extremely narrow, effectively making the output binary, with the centre set to 0.4. The result is that a hormone level of 0.4 should make the rover roll, while a higher hormone level should make it lurch.

To achieve this using the subsumption architecture, two extra machines were added which subsume the outputs of all the wheels:

- The `hormoneroller` is only active when the hormone level is *low*. If so, it waits for all wheels to be forwards (i.e. for all rolls to have finished) and then starts to output plain rolling data and a centred lift position. It will not output any drive to the motors until the wheels are all in the centre position, to avoid current surges. If the hormone level becomes high again, it will stop outputting (or rather, start outputting nil once more). (Note that this refers to the final version, earlier versions switched immediately into rolling when the hormone level became low — see Section 2.11.2.2 below.)
- The `kicker` is used to initiate the lurching gait. If the wheels are centred, it sends signals to move them all to the back position. Because the kicker is itself subsumed by the roller, this will have no effect if the rover is rolling. If the rover stops, however, it will force the wheels into the back position where the lurcher will once again begin its cycle.

The architecture is shown in Figure 2.19, with the parts added to the basic lurcher shown in red. Note that only one side of the rover is shown, but there is only one `kicker` and one `hormoneroller` across the entire rover.

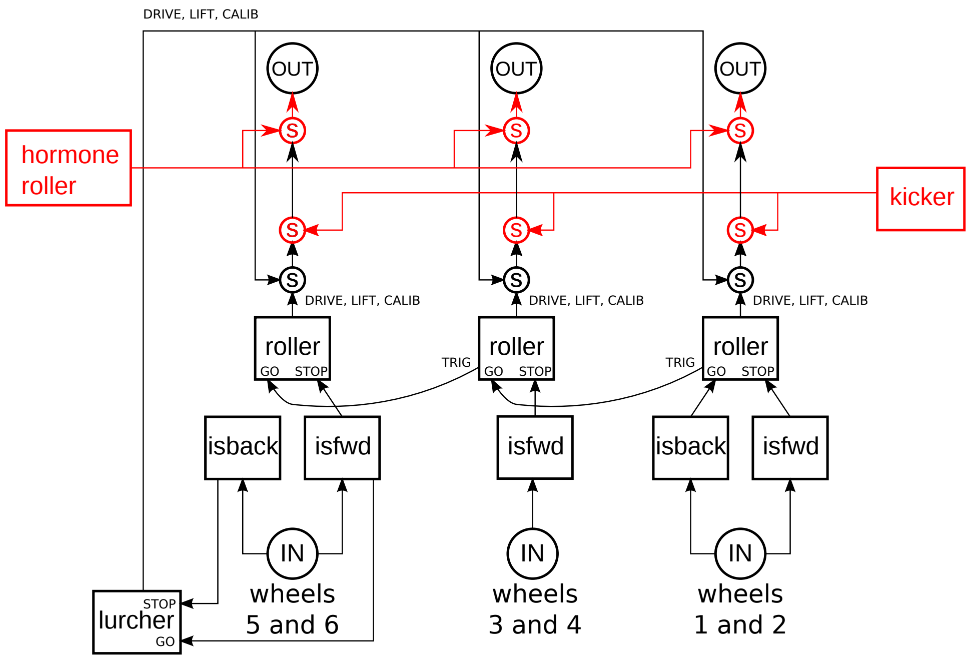


Figure 2.19: The endocrine gait switcher, switching between lurching and rolling, shown as an additional layer built on top of lurching.

The actual endocrine code is run as part of the main loop, which makes the appropriate calls to `updateHormone`. The hormones are global variables which are accessed inside `hormoneroller`. For a small system like this, the lack of encapsulation is acceptable. The entire endocrine controller code is included as Appendix D.2.8.

2.11.2.1 Hormone cascade system

The system oscillated a great deal with with a single hormone, due to the large variation in current readings. A hormone cascade was developed, in which the first hormone smoothed the current by using a very wide sigmoid, a slow decay and low release rate. The output function of this "input hormone" was not used: instead, the "switching hormone" was fed directly by the input hormone's level. This "hormone cascade" is shown in Figure 2.20[7], and the actual parameters are shown in Table 2.1.

This resulted in considerably smoother performance, but still with some oscillation as shown in Figure 2.21. This experiment also has a video[8]: see Table E.1.

The behaviour has several stages, shown on the plot by numbers:

1. the input hormone (in blue) is smoothing the input current (in black), because it is set to an almost linear sigmoid with a low release rate and a very slow decay. The input hormone

[7]It is very common in biology for a hormone to stimulate receptors which lead to (or inhibit) the secretion of other hormones in this way.

[8]`https://www.youtube.com/watch?v=KS-EX95luHs`

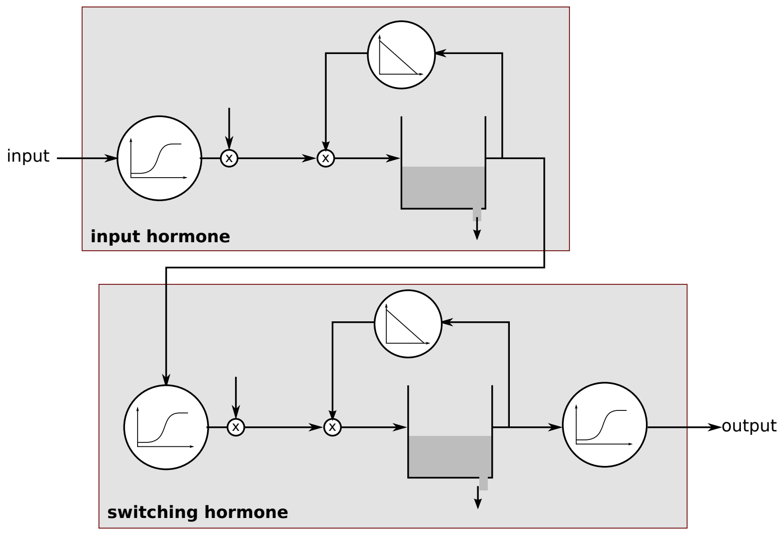


Figure 2.20: A 2-hormone cascade, where the input of the second hormone is fed directly from the level (not the output) of the first. Compare with Figure 2.17.

Parameter	**Input hormone value**	**switching hormone value**
Input sigmoid c	200	0.1
Input sigmoid w	800	0.01
Release rate	0.001	0.1
Decay constant	0.994	0.9
Output sigmoid c	unused	0.4
Output sigmoid w	unused	0.001

Table 2.1: Parameters for the 2-hormone cascade switching gait system. Note the wide input sigmoid, slow release and slow decay for the input smoothing hormone, the narrow input sigmoid and faster rates for the switching hormone, and the near-binary output at a level of 0.4.

begins to build up.

2. The switching hormone (in red) begins to be released as the value of the input hormone enters the significantly non-zero part of its input sigmoid, roughly between the two dotted blue lines on the plot (this hormone's scale has been vertically exaggerated).

3. The switching hormone reaches its threshold (actually a very narrow sigmoid, indicated by the red dotted line), and the rover begins to lurch.

4. The current begins to fall, so less input hormone is released.

5. After about a minute, the input hormone has fallen sufficiently that it no longer "tops up" the switching hormone, which begins to fall rapidly.

6. The input hormone crosses its threshold the other way, the rover begins to roll. This is where the problems begin.

7. The current immediately spikes, causing the input hormone to begin to rise again.

8. This causes the switching hormone to rise, and the rover returns to lurching.

9. The current immediately falls, as does the input hormone, and the switching hormone follows, switching back to roll.
10. Which immediately causes another current spike, and the cycle continues.

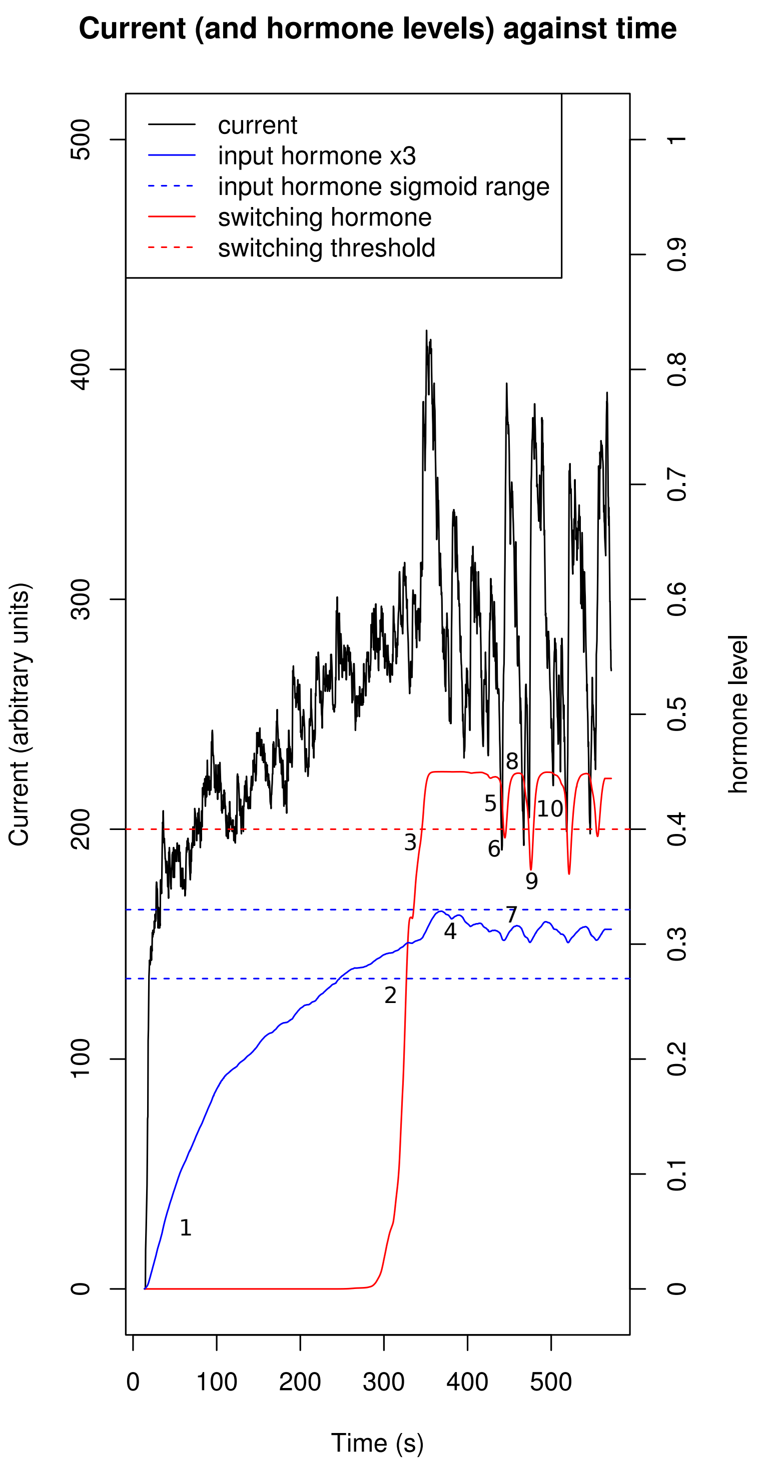


Figure 2.21: Two hormone cascade with simple hormone roller (no waiting), showing oscillation at end of lurch phase. This is from the data file `aes/exp4`. The numbers refer to the enumerated stages in the text on page 31.

2.11.2.2 Resolving the current surges

These problems occur because this early version of `hormoneroller` simply switched the gait from rolling to lurching by immediately commanding the legs to the centre position and starting to drive. This caused a large drive current, as the drive wheels would try to go from standing to full speed. At the same time, the lift motors would draw very large currents as they would try to push in opposite directions to all get to the centre position.

This was resolved by adding the extra logic in `hormoneroller` described above, to ensure that the wheels were all at the same angle before rolling restarted and that the drive motors did not draw current until the lift motors were stopped. Once this was added the system worked well, as demonstrated in Figure 2.22 (also available as an on-line video[9], see Table E.1).

In this plot:

1. The current rises very rapidly, causing the input hormone to rise[10],
2. which in turn causes the switching hormone to be released,
3. and soon the rover switches to lurching.
4. The current drops to a lower level, but not enough to lower the input hormone to levels where the switching hormone begins to fall. Once the rover is on the flat, the current falls further, as does the input hormone, and then the switching hormone.
5. Now, when the switching hormone drops below threshold and rolling starts again, there is no current spike — just a small plateau, enough to cause a small amount of switching hormone to build up, but not sufficiently to cross threshold again.
6. The current continues to fall now the rover is on the flat, and the roll continues.

Unfortunately, there was insufficient time to perform further experiments or analysis on the switching gait.

[9] `https://www.youtube.com/watch?v=7vspI6JLdMA`

[10] The hormone starts high because this run follows on from another recorded in the same Angort session — the script in its current form does not recreate the hormones for each run. This is a bug, but has little effect on the outcome beyond causing the lurch to occur a little earlier.

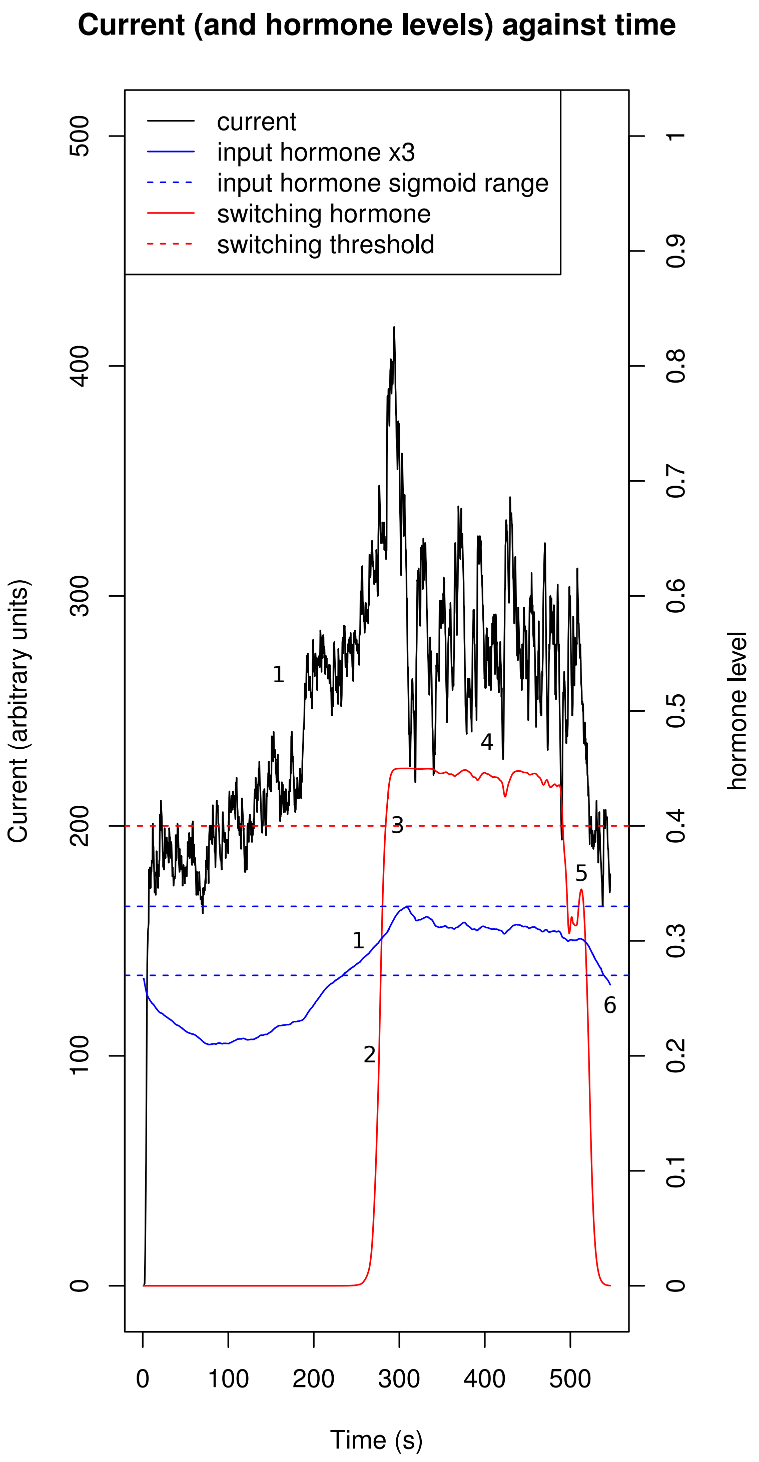


Figure 2.22: Two hormone cascade with final hormone roller, showing successful transitions and no current spike on return to rolling. This is `aes/exp8`. Numbers refer to stages on page 35.

Chapter 3

Experimental Methods

This section will discuss the experimental methods used to answer the research questions. It will not cover details of the technologies used — these are covered in Chapter 2 (which includes both technologies developed for the experiments and pre-existing systems).

3.1 Overview

There are several components to the experimental methodology:

- the environment in which the experiments take place;
- how the surface was prepared;
- the IT setup used (this is covered in Chapter 2);
- how the data from the monitoring system was collated with the motion capture data;
- how and why the data was cleaned up;
- which experiments were performed.

Analyses and results of the experiments can be found in the next chapter.

3.2 The environment and surface

All experiments took place in Aberystwyth's Planetary Analogue Terrain laboratory (PATLab), on a surface of Mars Soil Simulant-D from DLR Germany, which is geophysically analogous to Martian regolith (soil). This is similar to talcum powder in consistency, and has a particular tendency to cause wheel slip and sinkage when not packed down.

Much work has been done on analysing such surfaces for wheeled rovers, based on MG Bekker's terramechanics [10,40], although there has been relatively little research done on legged and hybrid models in such conditions. Hidalgo et al. have performed some preliminary studies, developing a detailed kinematic model with slip for a four-wheeled rover [21], but this has only been done in simulation. They also state that

> "it is not the purpose of this work to accurately model surface conditions, [therefore] the wheel-terrain interaction is based on simple models."

The "Mars yard" in the PATLab has a usable area of approximately 5m by 3m, with one end being made up of an slope with another flat area on top. This slope is not packed regolith: there is a layer of polythene sheeting covering polystyrene blocks about 10cm below the surface. This caused problems in some of the runs on a loose slope when the rover wheels wore through to the sheeting.

Figure 2.5 shows a rough diagram of the yard, with the slope represented by steps. In reality, the slope has a roughly sigmoid profile with a maximum inclination of approximately 20°, although this varied over time due to the movement of the simulant. The precise shape of this profile and how it changed is explored in Section 4.1. A photo of the part of the yard is shown in Figure 3.1, but the field of view is limited. This shows an uphill view of the slope, and underestimates its steepness.

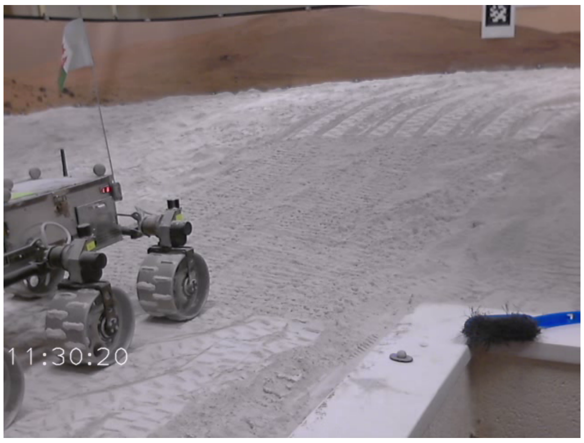


Figure 3.1: A photo of approximately half of the Mars yard, looking up the slope.

3.3 The experimental IT setup

This is described fully in Chapter 2 (see Figure 2.1), but in overview:

- Experiments are run on the rover's computer, to which a laptop is remotely connected via an SSH shell session.
- The rover sends the rover's internal sensor data (odometry, currents etc.) to the laptop, which is also running a monitoring program which captures the data to a file.
- At the same time, the Vicon motion capture system captures the position and orientation to a separate file on another system (because it runs on its own private network).
- Synchronisation of the two processes is done by button press.
- The two resulting data files, one from the monitor and one from the motion capture system, are then collated by a Python script into a single file.
- This file is examined for problems, and cleaned up.

3.4 Method for all experiments

Experiments were performed referring to a checklist, which is given in Appendix G. In summary:

- write, test (using the simulator) and transfer the script to the rover;
- ensure the surface is appropriately prepared — that the surface is packed or loose, and that the slope has been repaired from any slipping which may have occurred;
- start the rover scripting system with the script;
- use remote control commands to move the rover to the correct starting position (marked by a tape marker for the slope, such that the rover runs for approximately 1m before starting to climb; or as far left as possible for flat experiments);
- start the monitor and run the experiment script;
- ensure data is being collected both by the monitor and Vicon;
- start logging and then start the experiment and Vicon capture at the same time;
- stop the experiment;
- retrieve and collate the data.

Each experiment was performed at least five times.

3.4.1 Surface preparation

Surface preparation was performed by raking the soil with a steel rake if a loose surface was required, or packing it down using a pressure of approximately 500Pa[1]. Unfortunately, rollers were not available to pack the surface properly, and no way of measuring the degree of packing was available. This led to problems with consistency of the surface state, affecting the repeatability of experiments.

Additionally, each experiment on the slope packed some simulant under the wheels, while moving more simulant downhill. Attempts were made to ameliorate this by rebuilding the slope after each run, with the result that the slope became less packed. For later packed runs, a fresh part of the slope was used, but this had a lower maximum inclination. The first experiment, for roll at 500, had a notably steeper profile. This is described more fully in Section 4.1.

3.5 Merging and cleaning up the data

As described above, the experimental procedure produces two data files: one containing data captured from the rover's sensors, and one containing rover position and orientation data from the motion capture system. These must be merged into a single file, which is done using a Python script called `collateViconAndRoverCSV` (collate Vicon and rover data into a Comma Separated Values file).

[1]the pressure applied by a man standing on a wooden rectangle

This is a complex operation because the files will have started at different times, and have very different formats. The Vicon capture file starts when the "Capture" button in the Vicon iQ interface is pressed, and consists of a short header followed by comma-separated values consisting of frame number, orientation Euler angles, and position coordinates (relative to an arbitrary but constant point); whereas the rover capture file consists of key-value pairs, with data omitted when a value has not change since the last packet was transmitted. An example can be seen in Appendix B.1.

Synchronisation is possible because the Vicon capture is started at the same time as the EXP GO switch is toggled in the monitor, when the `go` variable becomes 1 in the rover capture data.

The collate code takes both files, producing a comma-separated value for each full update cycle in the rover file, merged with the Vicon data at that point. The algorithm is presented in Appendix C with some explanatory notes.

3.5.1 Cleaning up the data

Data collected from the Vicon system frequently contained errors, which manifested as discontinuities in position or orientation. This typically happened at the far end of a run, when the camera data became ambiguous. An example of this can be seen in Figure 3.2, where the distance travelled (from $\sqrt{x^2 + y^2 + z^2}$ in the collated data) becomes nonsensical after about 350s.

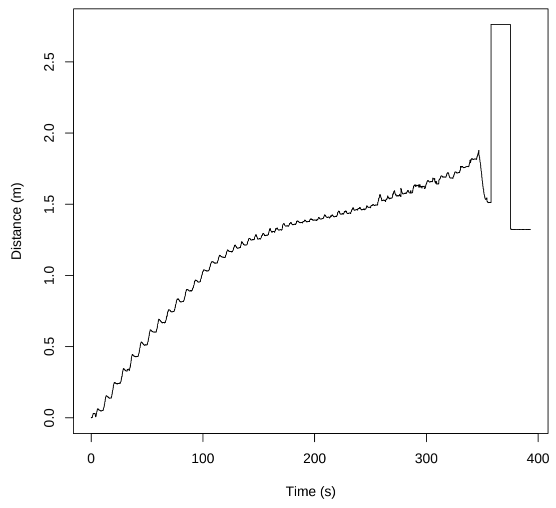


Figure 3.2: Example of motion capture error.

Such errors were detected visually by plotting the distance and inclination (rotation around y) for all runs. In the case of end-of-data errors such as that above, all the data for the after corresponding times in the rover data capture (not the Vicon data) was elided, and the data re-collated. This caused all data after the given time to be removed, because (given the collation algorithm) no data could be found with which to collate the erroneous Vicon data.

In a few cases this required a re-run because so much data was lost (alternating gait runs on the slope proved very tough for this).

Occasionally, an error (typically a rotation) would manifest as brief jump in orientation to another pose. This was dealt with by removing the incorrect data from the Vicon stream. This would cause a small discontinuity, but it was smoothed by the splines used to condition the data (see Chapter 4).

3.6 Experiments performed

The experiments were intended to answer two questions originally given in Section 1.2.4:

- What is a good metric?
- Is there a gait which works which is more efficient than plain rolling when used on the slope?

To answer these questions, the behaviours of the selected gaits both on the flat and on the slope were recorded. Additionally, the condition of the surface (packed or loose) may have an effect, so all experiments were performed for each of these conditions. This gives the 16 experiments shown in Table 3.1. The number in the rolling experiments refers to the commanded rover speed in arbitrary units. 500 is approximately 6mm/s.

Gait	**Type**	**Surface**
roll at 500	flat	loose
roll at 1000	flat	loose
lurch	flat	loose
alternating	flat	loose
roll at 500	slope	loose
roll at 1000	slope	loose
lurch	slope	loose
alternating	slope	loose
roll at 500	flat	packed
roll at 1000	flat	packed
lurch	flat	packed
alternating	flat	packed
roll at 500	slope	packed
roll at 1000	slope	packed
lurch	slope	packed
alternating	slope	packed

Table 3.1: The 16 experiments which were performed (not including neuroendocrine tests).

3.7 Neuroendocrine experiments

Because of a lack of time, little formal experimentation was done on the neuroendocrine controller beyond that of tuning it and demonstrating that it worked. This is described fully in Section 2.11.

Chapter 4

Results and Conclusions

This chapter analyses the results of the experiments listed in the previous section. It deals with:

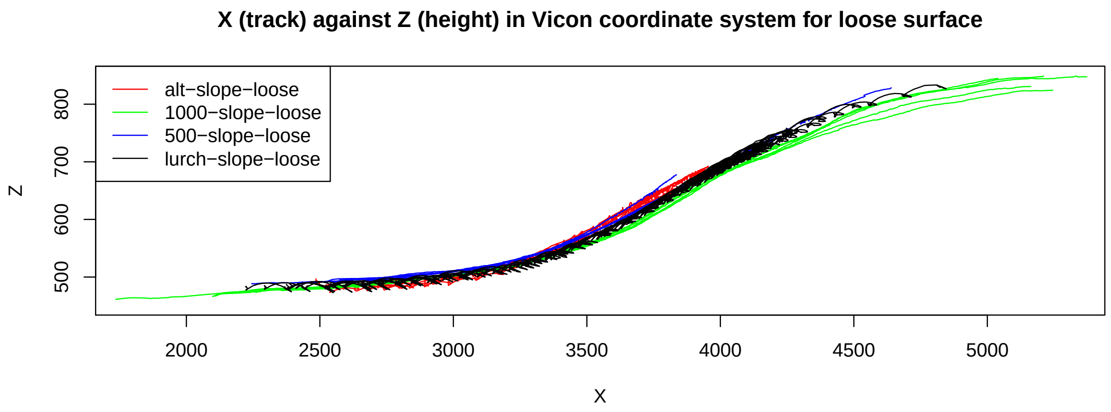
- the profile of the slope;
- verifying that the rover moves differently on the slope and the flat;
- finding, testing and comparing metrics;
- comparing the various gaits using these metrics;
- finding a metric and gaits which are suitable for building an AES-based switching gait.

The analysis was done using the R programming language [30]. Initially, Python with the NumPy and SciPy libraries was used, but R was found to be better integrated (being designed for the analysis of vectorised data from the ground up), better supported and to have a wider range of better-documented packages. There was a steep initial learning curve, however. All the R scripts written are included in the electronic submission in the `project/data` directory.

4.1 Profile

The first use made of the data was to determine the profile of the slope. This was done by plotting y against x in the Vicon's coordinate system for every run. The results for both loose and packed experiments are shown in Figures 4.1 and 4.2. It can be seen that:

- The shape of the slope was slightly different for each experiment, because the slope changed over the course of the study and there were no facilities to reshape it beyond manually moving soil from one part of the yard to the other. It is also possible the rover occasionally deviated course slightly, either onto a less sloped part of the environment or to traverse the slope at an oblique angle, giving a lower apparent slope.
- It was not possible to run all experiments on the entire slope on the loose surface — the alternating gait and 500 roll in particular were unable to proceed after a certain point, getting stuck in the soil somewhere between 17° and 20°.



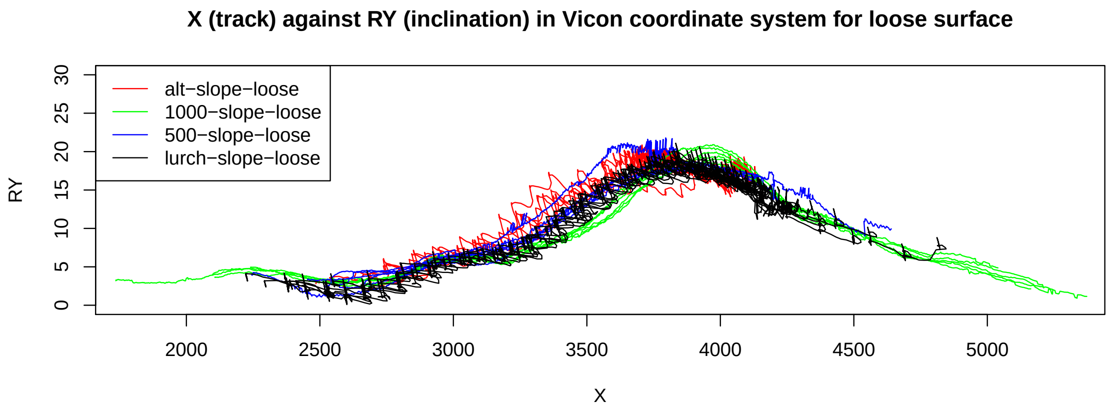
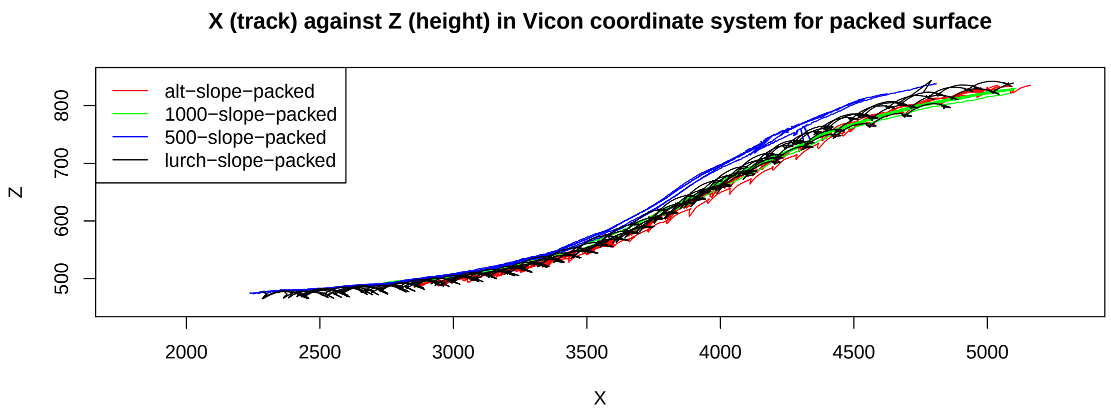


Figure 4.1: Slope profiles as Z (height) against X (track) for all runs (loose). Generated by the **plotProfile.r** script.

- The inclination of the packed slope was more variable than that of the loose, most notably for the roll 500 experiments which faced inclines of over 20°. This is because the packed surface gradually became less sloped, and it was extremely difficult to reform and repack it into the same profile. It would have been better to run all the packed experiments before all the loose experiments, but the problems with repacking the surface were not foreseen.
- The inclination data is very noisy for the walking gaits, because the walking actions modify the inclination of the rover.

This variability in the slope should be largely dealt with by analysing the metrics against inclination, so it should not effect the validity of the findings. However, some experiments were not run on the full range of inclinations — for example, roll 1000 was not run with an incline of much greater than 15° on a packed surface.



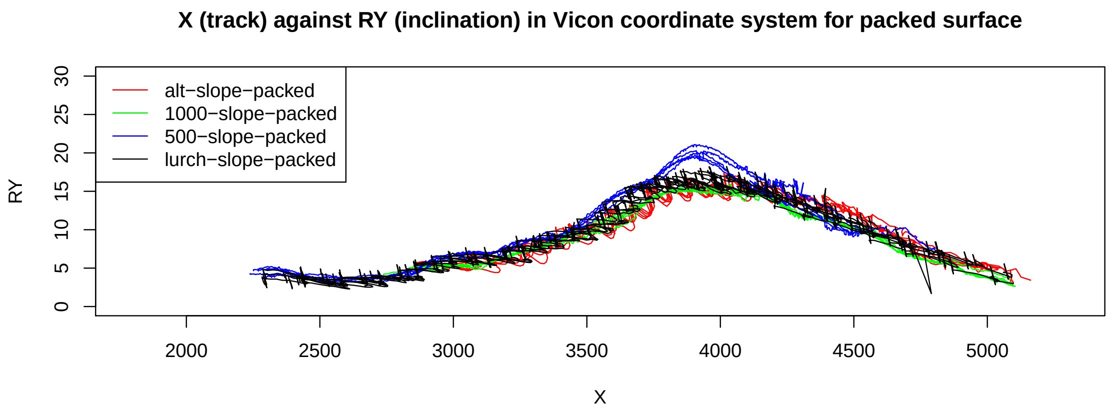


Figure 4.2: Slope profiles as Z (height) against X (track) for all runs (packed). Generated by the **plotProfile.r** script.

4.2 Velocities

To plot the velocities, the distance values from the origin were calculated for each run as $\sqrt{x^2 + y^2}$ (the Vicon's vertical axis being z). These were then approximated by a polynomial smoothing spline of order 5, with a low smoothing parameter [31]. This allowed the derivative of the distance (i.e. the velocity) to be determined. The results for all runs on the both packed and loose surfaces are shown in Figures 4.3 and 4.4.

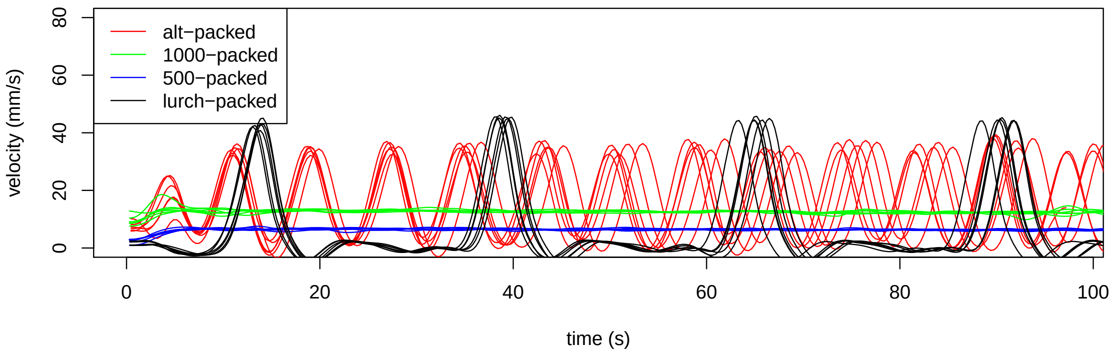


Figure 4.3: Velocities of all experiments on the flat packed surface, approximated by the derivative of a polynomial smoothing spline applied to the distances. Generated using the **plotAllVelAgainstTime.r** script.

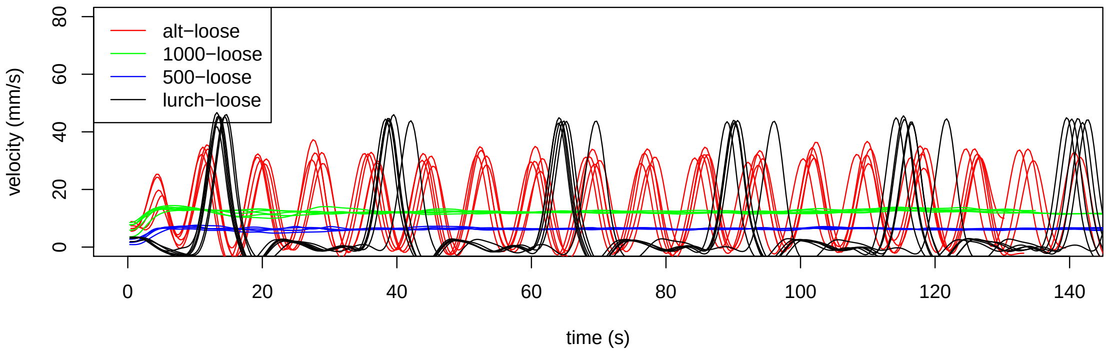


Figure 4.4: Velocities of all experiments on the flat loose surface, approximated by the derivative of a polynomial smoothing spline applied to the distances. Generated using the **plotAllVelAgainstTime.r** script.

It can be seen that

- the velocities of the rolling gaits are fairly constant, as should be expected;
- the velocities of the two walking gaits oscillate as expected (particularly in the lurching gait, given its nature);
- as do the circumference velocities of the wheels, given the oscillatory nature of th gaits. of the gaits.

Some smoothing may therefore be required on the underlying data to calculate the metrics.

4.3 Experiments on the flat - finding a metric

4.3.1 Calculating the slip

The initial hypothesis is:

> *Some metric based on how much the wheels slip on the surface is a suitable metric for efficiency.*

This is based on the notion that any energy expended driving the wheels is wasted if it is not moving the wheels forwards. Slip is a common metric in the analysis of rover motion [10, 21, 26, 40]. Some analyses of the data from the flat experiments were made to test slip-based metrics.

There were several slip metrics available. The first is the *simple slip ratio,* calculated as the ratio of the mean of circumferential velocities of the wheels $v_{c_i}, i \in [1, 6]$, and the rover's velocity v_r as measured by the Vicon:

$$s_{simple} = \frac{\bar{v_c}}{v_r}$$

This metric has a problem: an immobile rover with wheels turning has an infinite slip. Moreland et al. [26] defines the slip ratio in a similar way, but by referring to the rover velocity alone:

$$s_{moreland} = \frac{\textit{current rover speed}}{\textit{baseline flat rover speed}}$$

This metric deserves further study, but was not used here — it was felt more useful to use a metric which did not rely on a fixed measurement of an ideal baseline speed, and which used at least one interoceptive sensor.

Yoshida and Hamano [40] propose an alternative slip ratio metric for each wheel[1]:

$$s_i = \begin{cases} (v_{c_i} - v_r)/v_r & (v_{c_i} < v_r) \\ (v_{c_i} - v_r)/v_{c_i} & (v_{c_i} \geq v_r) \end{cases}$$

[1]Ding et al. [10] use a similar measure, but assume that the the wheel velocity is always greater than the rover velocity.

where v_r is the velocity of the rover, and v_{c_i} is the circumferential velocity of wheel i. This metric was modified to act over the whole rover:

$$S = \begin{cases} (\bar{v}_c - v_r)/v_r & (\bar{v}_c < v_r) \\ (\bar{v}_c - v_r)/\bar{v}_c & (\bar{v}_c \geq v_r) \end{cases}$$

This is compared with the simple slip ratio in Figure 4.5.

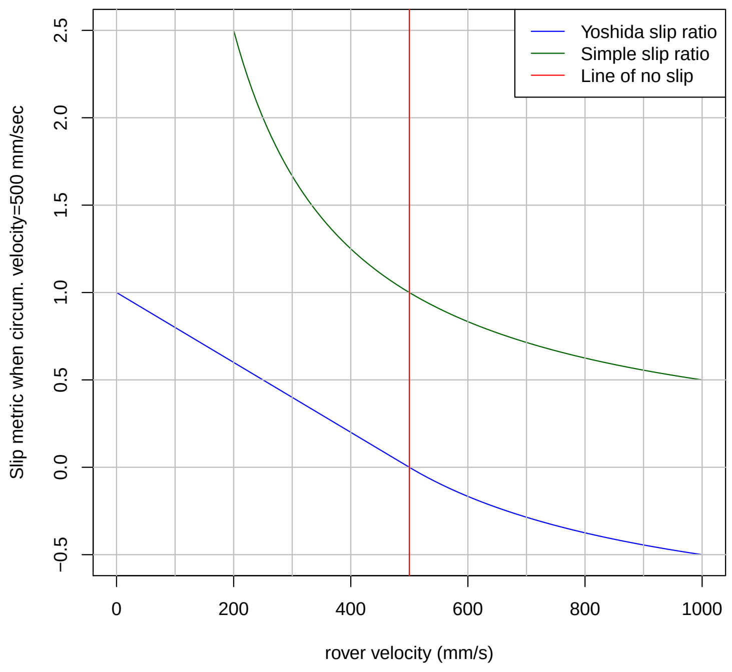


Figure 4.5: A comparison of Yoshida's slip ratio [40] with the simple slip ratio, by fixing the average wheel velocity at 500 mm/s and varying the rover velocity between 0 and 1000 mm/s. Generated using the **slipRatioComparison.r** script.

The simple slip ratio is highly non-linear where the rover velocity is slower than the wheel velocity — i.e. the rover is moving slower than it should be, which is the usual case in the current study. The Yoshida slip ratio becomes linear at this point, making it more tractable to analysis and comparison. Therefore, we will use the Yoshida slip ratio instead of the simple slip ratio.

One issue remains: calculating the circumferential velocity. Work towards this was done during the initial work on the rover, where a conversion factor from odometry ticks to millimetres was estimated, based on the measured circumference of the wheel and the number of encoder ticks in a complete revolution.

4.3.2 Measuring slip on the flat

If the Yoshida slip ratio is calculated for all the runs on the flat over time the plot in Figure 4.6 is obtained. With no smoothing, it is clear that the slip value varies too much to extract any useful information. In the case of the walking gaits, this is because of the different slip amounts during the gait cycle. For example, during the "rolling" phases of the lurch gait, the rover is not moving but the wheels are showing odometry. In the "push" or "lurch" phase the rover moves, but the wheels do not show odometry.

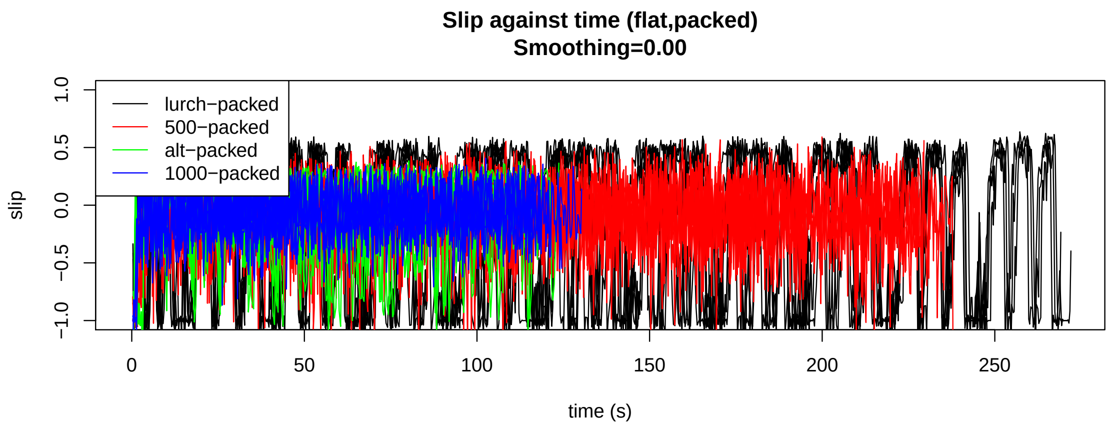


Figure 4.6: Slip over time for all runs of all gaits on the flat on a packed surface, showing extreme slip variation when no smoothing is applied, generated by the **plotAllSlipAgainstTime.r** script.

Perhaps a more useful view of this data is a histogram, as shown in Figure 4.7.

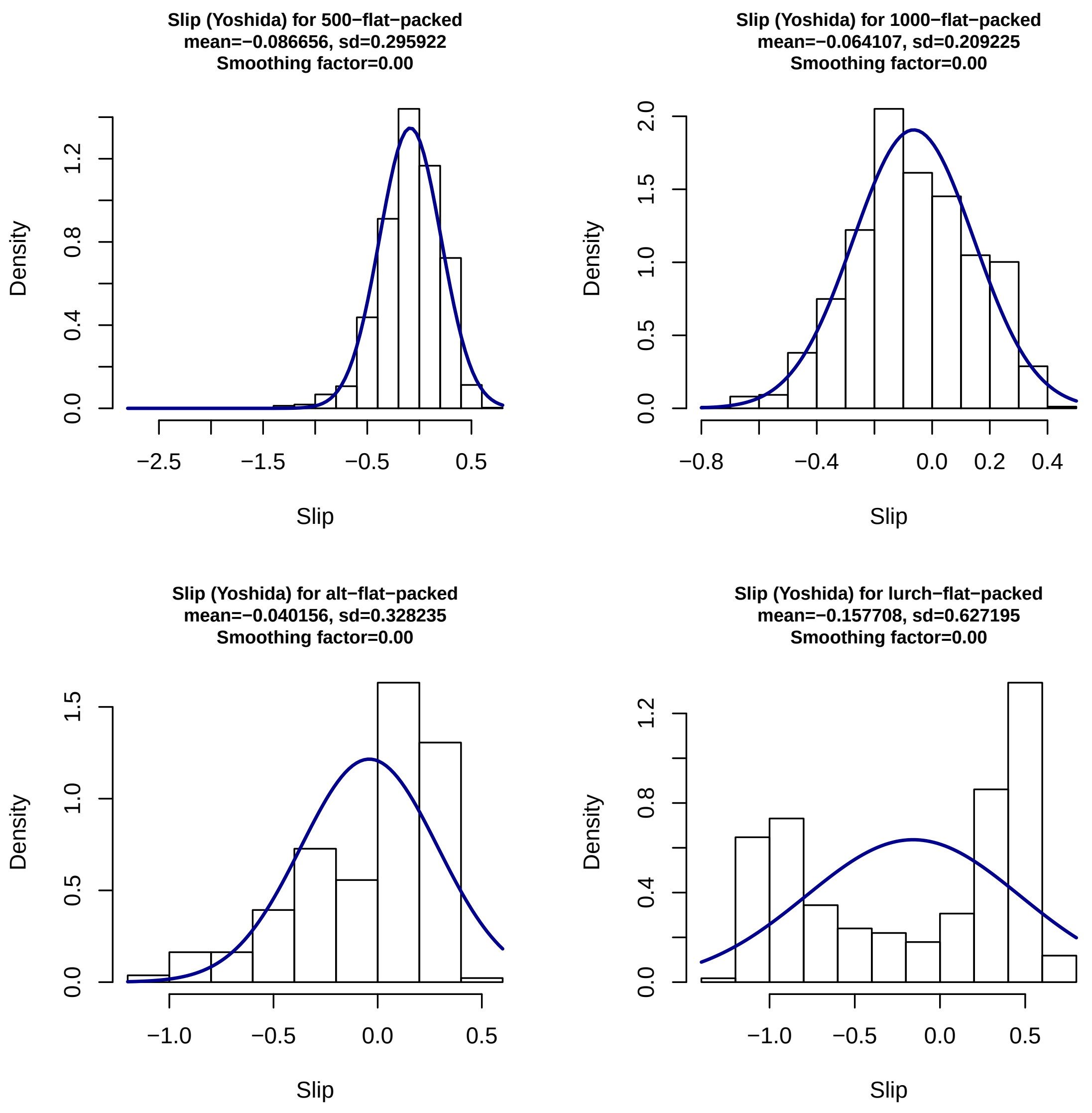


Figure 4.7: Histograms showing the distributions of different slip values during packed runs on the flat, for all experiments, generated by the **getBaseSlip.r** script.

This shows some interesting features more clearly:

- The two rolling gaits both show normally-distributed slips, as is to be expected given the complex nature of the wheels' interactions with the soil and the inherent oscillations in the control system (see Figure A.1).
- The lurching gait shows two distinct peaks — in the larger peak, the rover is slipping backwards a very large amount, approaching -1 (not moving at all, see Figure 4.5). This corresponds to the "rolling" phase, when the wheels are rolling forwards in pairs while the rover remains stationary. In the smaller peak, the rover is moving faster than the wheels are rotating, corresponding to the "lurching" phase. The true slip ratio over time may be a weighted mean of these values.
- The alternating gait is a more complex case, given the nature of the gait and how it was implemented, but the fundamental problem is still the same — the slip varies wildly over the gait cycle, and the true slip is some mean of these values.

It can be seen that in order to measure the effect of the incline on the slip ratio, the underlying data must be smoothed — it is the slip as a whole which is interesting, not its value in different parts of the gait cycle. In the above results, the odometry and distance data are approximated by polynomial splines (in order that they be differentiable). Increasing the smoothing factor to a much higher value should achieve this.

Figure 4.8 shows the effect on the slip of a large smoothing value imposed on the splines which approximate circumferential distance and rover distance. If histograms are plotted from this data,

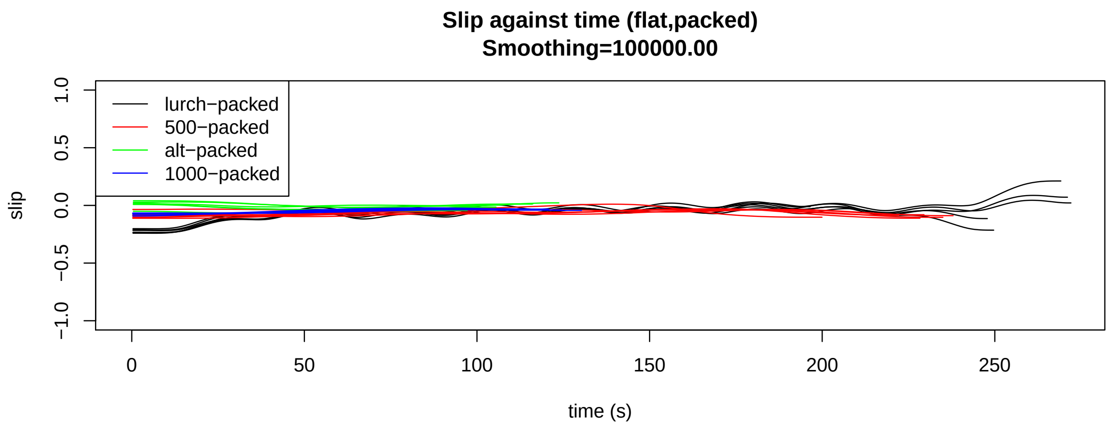


Figure 4.8: Slip over time for all runs of all gaits on the flat on a packed surface as in Figure 4.6, but with a much higher smoothing factor on the polynomial splines for the distances.

the distributions in figure 4.9 are obtained.

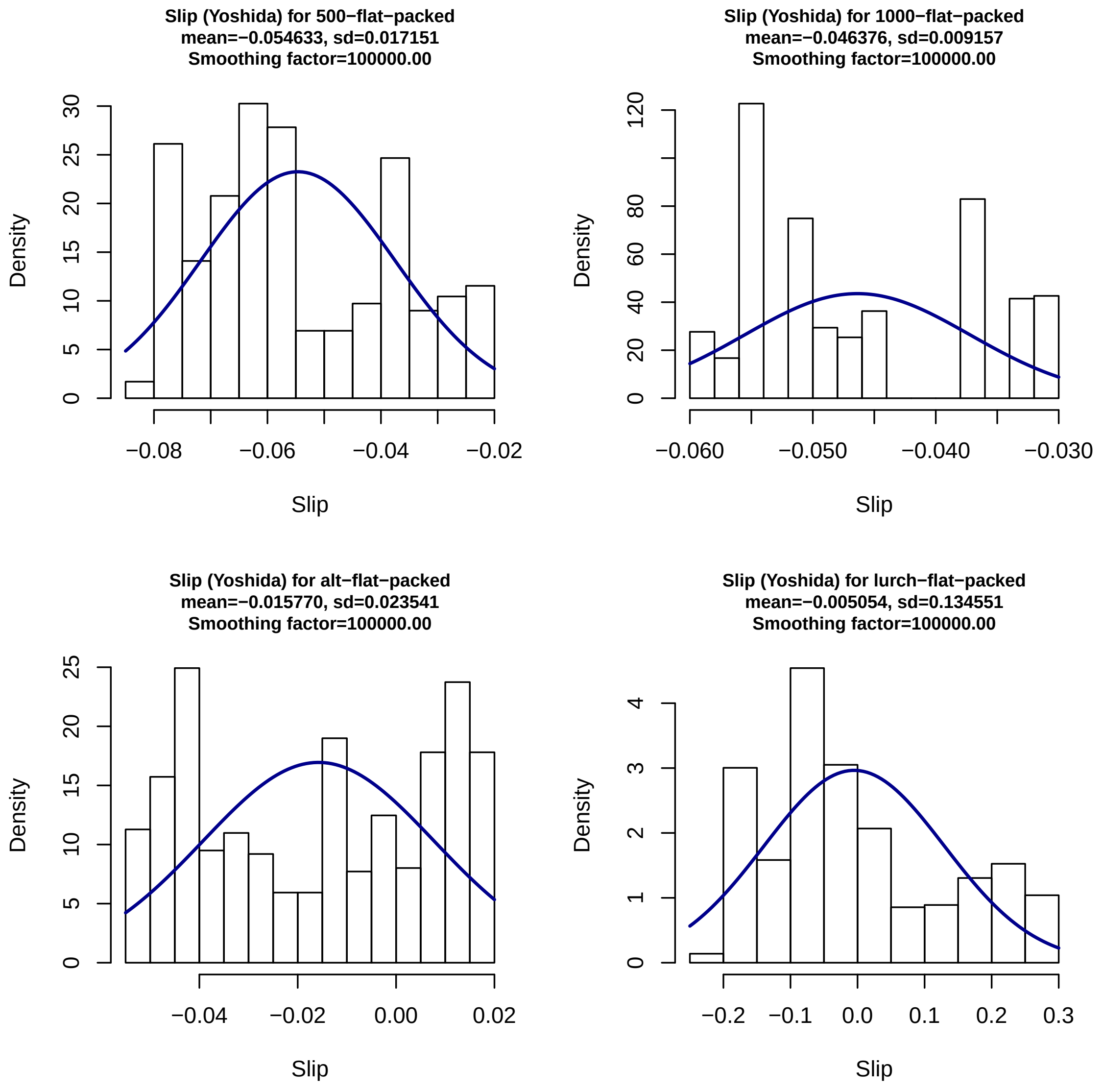
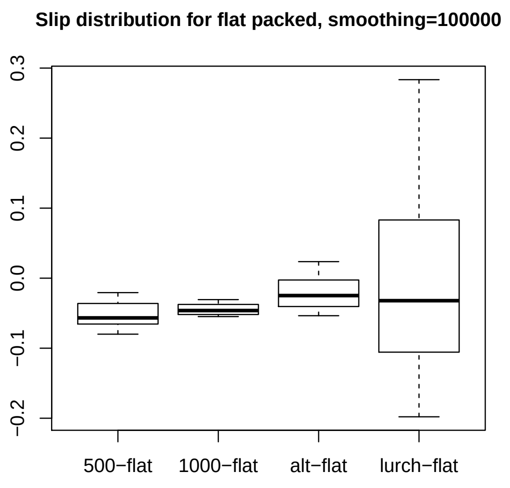


Figure 4.9: Histograms showing the distributions of different slip values during packed runs on the flat as in Figure 4.7, but with much larger smoothing values.

The normal distribution appears to have been lost, but the variation is considerably lower. This data is presented as a box plot in Figure 4.10.

There are some interesting features:

- The standard deviation of the slip is still high on the walking gaits, even when heavily smoothed, particularly on the loose. There is still a very large variation with considerable overlap between the experiments.
- Additionally, negative slip is present. This normally means that the rover is moving faster than the wheel odometry — this is unlikely for smoothed data, particularly in the rolling gaits. It is probable that the measurement of the factor which converts encoder ticks to odometry is incorrect, relying as it does on parameters which are difficult to measure (en-



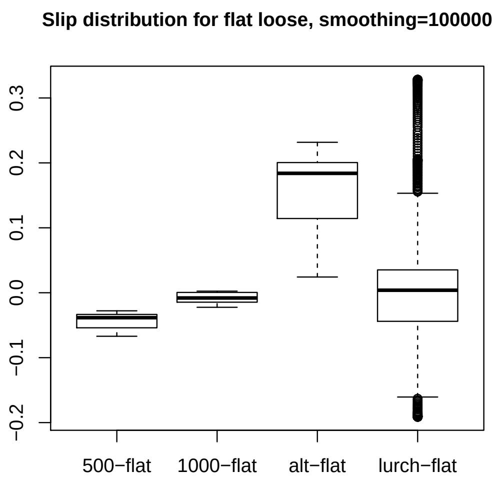


Figure 4.10: Box plots showing distributions of slips for flat runs, with a high smoothing factor.

coder ticks per revolution, for example). It should therefore be borne in mind that a zero Yoshida slip may not equate to zero slip in the system.

This may have significant impact on the metric. If the true "zero slip" point is close to zero, the magnitude of the Yoshida slip may be higher when there is no slip than when there is slip. However, this should only occur at very small slips of the order of those seen in experiments on the flat, and may not have an effect on the slope. Unfortunately, time did not permit a more thorough investigation of this problem.

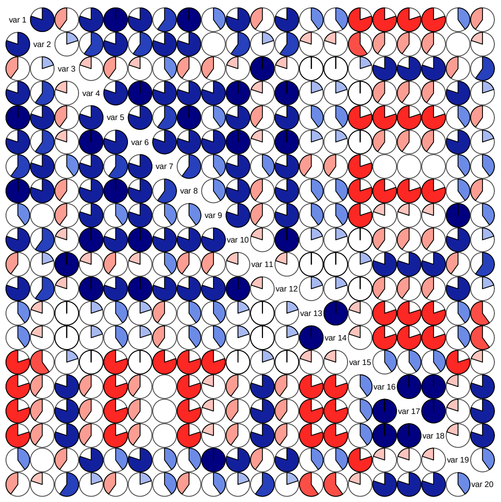
4.3.2.1 Correlation of Yoshida slip for different experiments across runs

For the Yoshida slip to be a useful metric, it should be consistent — that is, if a gait is better than another gait it should always be better than that gait. Given the variance of the metric, this is unlikely, but it should be tested.

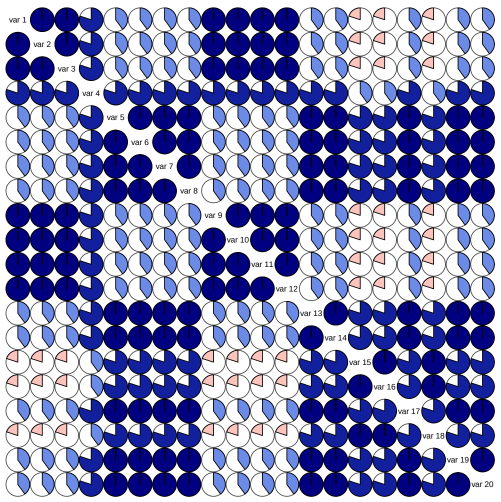
To do this, a correlation matrix based on Spearman's rank coefficient was used. Four random samples of the slip were taken from each of the flat runs (which should ideally have constant slip) for all the experiments, leading to a total of 20 samples per experiment. These were arranged into 20 data rows, each with four slip values — one for each experiment. For each possible combination of rows, the slips were ranked, and the two rankings correlated. Ideally, they should all agree with each other, and the resulting matrix should be filled with 1.

Correlation matrices for 20 variables are difficult to visualise, so correlograms (or corrgrams) were used. This technique, devised by Friendly [16], shows the matrix visually with a pie chart at each intersection showing the correlation — full blue for 1 (maximum positive correlation) and full red for -1 (maximum negative correlation) with an empty pie for no correlation. The correlations and plots were generated using the `corrgram` R package [39].

The results are shown in Figure 4.11. They clearly show that the Yoshida slip across the packed experiments is quite inconsistent, which is to be expected given the closeness of the medians in



(a) Correlation matrix for packed



(b) Correlation matrix for loose

Figure 4.11: Correlation matrices for 20 random samples from each experiment, ranked by Yoshida slip. Generated by the **rankSlipFlat.r** script.

Figure 4.10. The loose experiments are a little better, but still show a good deal of disagreement in the ranking with many combinations correlating only to $\rho = 0.6$.

4.3.3 Slip-based metrics: a conclusion

While slip provides us with a measure of the energy used in turning the wheels which is not used in moving the rover forwards (i.e. is wasted), it has several problems:

- Slip is a metric with an exteroceptive component — in order to calculate the rover velocity, one must first accurately measure its position.
- The direct slip ratio $\frac{\bar{v_c}}{v_r}$ shows highly non-linear behaviour the velocity is low.
- The Yoshida slip ratio corrects for this, but is neither consistent nor sensitive enough to distinguish between the different gaits on the flat.
- Finally, the rover motion in walking gaits is dependent not only on the drive wheels, but also on the rotation of the lift motors. It is not clear whether the slip takes this into account correctly.

Therefore slip may not be a suitable metric.

4.3.4 Current-based metrics

A more direct metric could involve measuring the amount of energy the rover is using. The rover, as noted in Appendix A.5, has current sensors in each of the motor driver chips. Because of the pulsed nature of the current actually delivered to the motors, this reading must be smoothed. This is done internally over short timescales within the rover firmware, as discussed in the appendix referenced above. Using current has the advantage of taking into account all the motors, both drive and lift.

Current itself cannot be used as a metric for efficiency — if it were, the rover would be at its most efficient when switched off. Efficiency implies striking a balance between work done and power consumed. The rover is required to travel as far as possible (in our simple model) for the least amount of power.

Therefore, it may be appropriate to use a simple ratio of current and velocity. A rover which draws current but does not move is infinitely inefficient, whereas a rover which moves while drawing no current will produce zero. This matches the ordering of the Yoshida slip values investigated so far — high values are bad, low values are good.

Under this metric, the velocity (if low) is much more significant that the current. This is appropriate, because whether the rover is moving is far more important than how much current it draws — if it cannot move, it cannot improve its situation.

4.3.5 Current/velocity on the flat

A raw plot of current/velocity on the packed flat is shown in Figure 4.12. It is clear once again that the velocity, which is highly variable because of the walking gait cycles, needs to be smoothed. Once this was done using the same technique and parameter as with the Yoshida slip, the ratio in Figure 4.13 resulted.

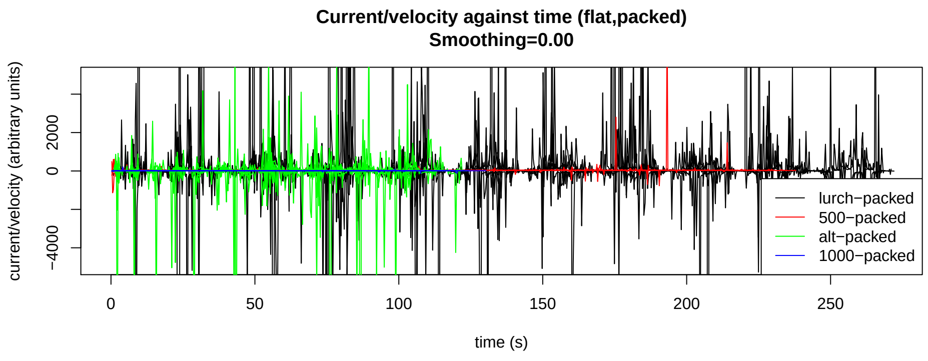


Figure 4.12: Unsmoothed current/velocity for packed flat runs. Note the y axis range — the spikes are very large.

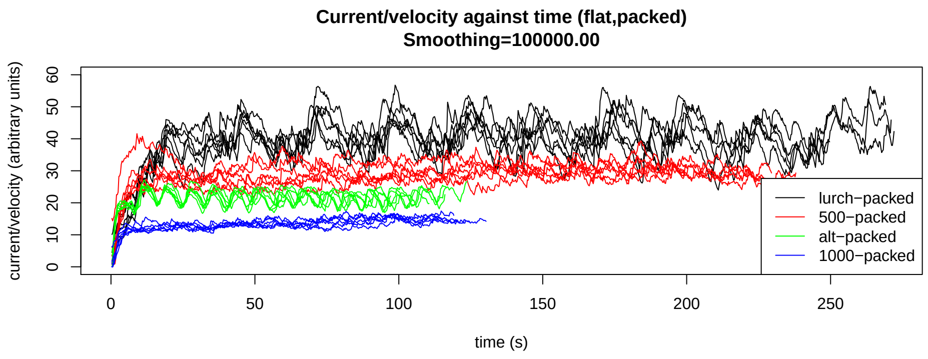


Figure 4.13: smoothed current/velocity for packed flat runs.

There is still a fair amount of variation, because the smoothing on the current (in firmware) is considerably less than that imposed on the velocity. However, clear differences between each gait are now visible. A boxplot for both packed and loose runs on the flat is shown in Figure 4.14.

Although the lurching gait shows some high outliers (corresponding to the peaks in Figure 4.13), there is much less variation and clear separation between the gaits. The source of these peaks in lurching is shown in Figure 4.15: they occur when the rover is rolling the wheels forwards and not actually moving — the velocity drops down to near zero, resulting in a very high current/velocity ratio. This is ameliorated by the smoothing (shown in green). Without the smoothing, the division by values close to zero results in the very high values seen in Figure 4.12.

There are several other interesting features in the box plot:

- The rankings of the various gaits are very different from those seen tentatively with the Yoshida slip: under that metric, the alternating gait did badly — unequivocally so on the loose surface. With the current/velocity metric, it is second only to roll-1000.
- Rolling at 1000 slips more than rolling at the lower speed (at least on the packed surface; the slip correlations are too weak for conclusions to be drawn on the loose), but appears to draw considerably less current. In fact, rolling slowly appears to be extremely inefficient, second only to lurching. We will discuss this further below.

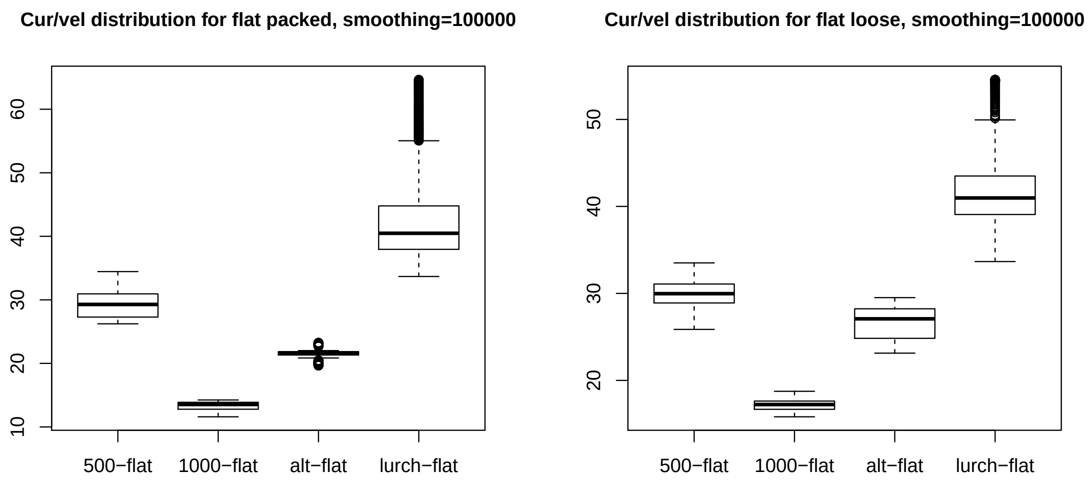


Figure 4.14: Box plots showing distributions of current/smoothed velocity for flat runs. Generated by the **boxPlotCurVelFlat.r** script.

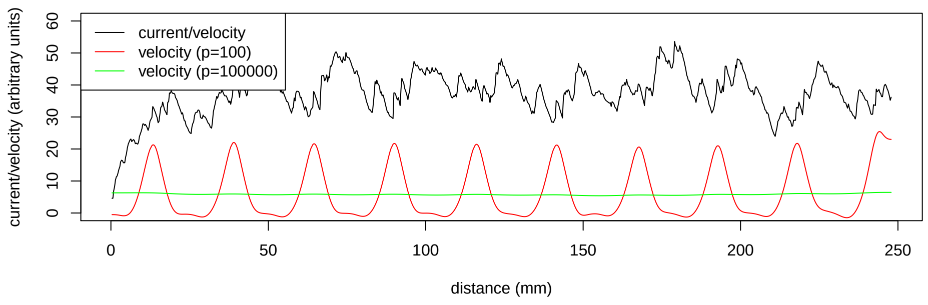


Figure 4.15: Current/velocity and smoothed/unsmoothed velocities for a single lurching run on the flat, demonstrating the peaks in current/velocity against low velocity.

- Lurching does not perform well on the flat — this may be because the lift motors have a very high current draw when used alone. In the alternating gait, the motion of the legs is always assisted by some rolling, either from the same wheel (rolling) or from the wheels in the opposing tripod (pushing).

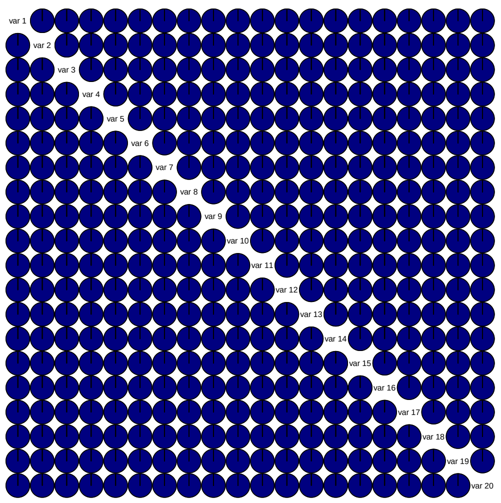
4.3.5.1 Poor performance of slow rolling

It is notable that rolling at a speed of 500 performs so badly. This is an extremely slow speed, roughly 6mm/s. At these speeds, the motor may be quite close to its stall speed — this can be seen from the plot in Figure A.1 in the appendices: the motor is oscillating a great deal around the speed requested. Such a speed may draw significantly greater current per unit rotation than higher speeds.

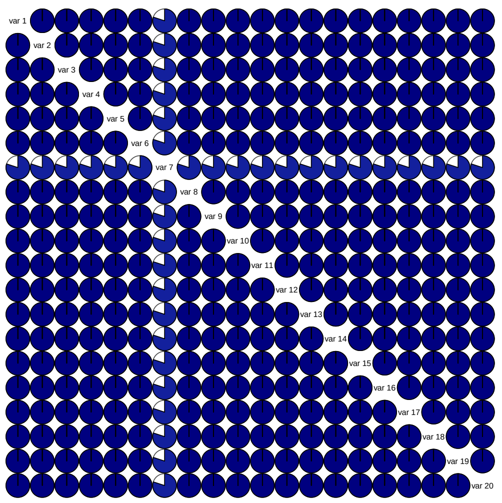
Unfortunately, these extremely slow speeds are often those at which planetary rovers are required to move, both for control reasons and to avoid mechanical shock — particularly on difficult terrain [8].

4.3.5.2 Correlation of current/velocity for different experiments across runs

Using the same procedure as in Section 4.3.2.1, the rankings of the gaits were correlated across random samples in different runs. The correlograms are shown in Figure 4.16. Clearly, this metric



(a) Correlation matrix for packed



(b) Correlation matrix for loose

Figure 4.16: Correlation matrices for 20 random samples from each experiment, ranked by current/velocity. Generated by the **rankCurrentVelFlat.r** script.

is strongly consistent across all the runs, and thus makes a good candidate.

4.3.6 Current-based metrics: a conclusion

The most obvious current-based metric, the current/velocity, turns out to be a good candidate for an efficiency metric. It is consistent, directly measures the power expended per unit of work, and is dependent upon all the motors. The slip, in contrast, is an approximate measure of wasted energy, and may not deal with walking gaits correctly. Current/velocity is, therefore, the metric which shall be most used in studying the slope. However, the Yoshida slip metric is still of interest — as shall be seen, it can be used to gain insight into the nature of the problems experienced by particular gaits.

4.4 Experiments on the slope

The plots in Figures 4.17 and 4.18 show the current/velocity plotted against x (in Vicon space). Some interesting features:

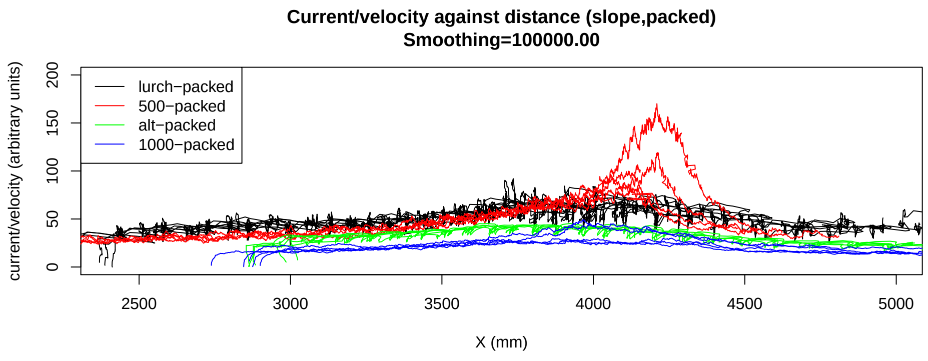


Figure 4.17: Current/velocity against x for packed runs on the slope.

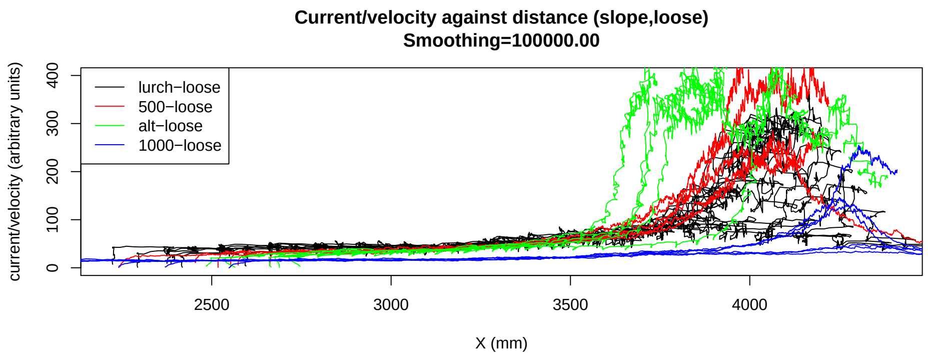


Figure 4.18: Current/velocity against x for loose runs on the slope.

- Roll-500 performs poorly on both surface types;
- Roll-1000 performs best, but there are two runs on the loose slope where it draws more current;
- Alternating performs very badly indeed on a loose surface, but better on the flat than all but roll-1000;
- Lurching also performs badly on a loose surface (although better than alternating), and is inefficient on the flat.

4.4.1 Packed slope experiments

A better analysis can be made by plotting the metric against the inclination (rotation around the Vicon y axis). For the packed surfaces, this gives us Figure 4.19. Each individual curve here is a single run, ordered by time, and a small amount of smoothing has been applied to the current to make the plots legible. Note that the graphs are logarithmic on the y axis. It can be seen that:

- Current/velocity is roughly exponential against inclination, hence the near linearity on a log plot.

- Lurching, alternating and roll-1000 behave consistently on the packed slope, with alternating performing slightly better at higher inclinations, although the data set for roll-1000 is limited, as mentioned in Section 4.1. It seems that current draw for roll-1000 is still rising exponentially at the higher inclinations for at least some of the runs, while other runs are better-behaved. This may be predictive of the failure modes seen in roll-500 (see below), which may occur in roll-1000 at higher inclinations.
- Rolling at 500 gets into difficulties at high but decreasing inclinations, where the rover has just crested the hill. This will be referred to as the *crest effect*, and will be analysed later in Section 4.5.1 — it is much more significant on loose surfaces.

The same transformation is applied to the Yoshida slip data in Figure 4.20. There are some interesting comparisons to be made with current/velocity:

- The Yoshida slip is often considerably lower on the upper flat, after the slope — often becoming negative. This may be an artifact of the miscalibrated zero slip (see Section 4.3.2), but it is also present (to a small degree) in the current/velocity data. Alternatively, it may correspond to a difference in packing between the two flat sections.
- Roll-500 shows the crest effect under this metric too, indicating that the high corresponding current/velocity is likely to be due to slip causing low velocities. Figure 4.21 shows slip, current/velocity and current for all wheels in a single run, and shows that while all wheels experience high slip on cresting the hill, wheels 3 and 5 (middle and back left) show current surges indicating an uneven load, and that wheel 3 may have stalled. This wheel recently had its motor replaced due to encoder failure, which involved disassembly and reassembly of the gearing. This may be affecting its performance: its current is consistently higher than all the others, indicating a possible gearing problem.

4.4.1.1 Conclusions

On a packed slope, rolling quickly is the best gait; although the walking gaits may be useful at higher inclinations: the general trend of roll-1000 indicates that it may begin to fail at slightly higher inclinations than those recorded. Rolling at 500 performs badly, probably because the motors are running very close to their stall speeds — particularly on wheel 3, which is known to draw high currents and may have gearing problems.

The walking gaits perform similarly to each other, with the current draw likely to be higher because of the use of the lift motors. The alternating gait performs differently on the top flat than on the bottom, possibly indicating a sensitivity to surface effects.

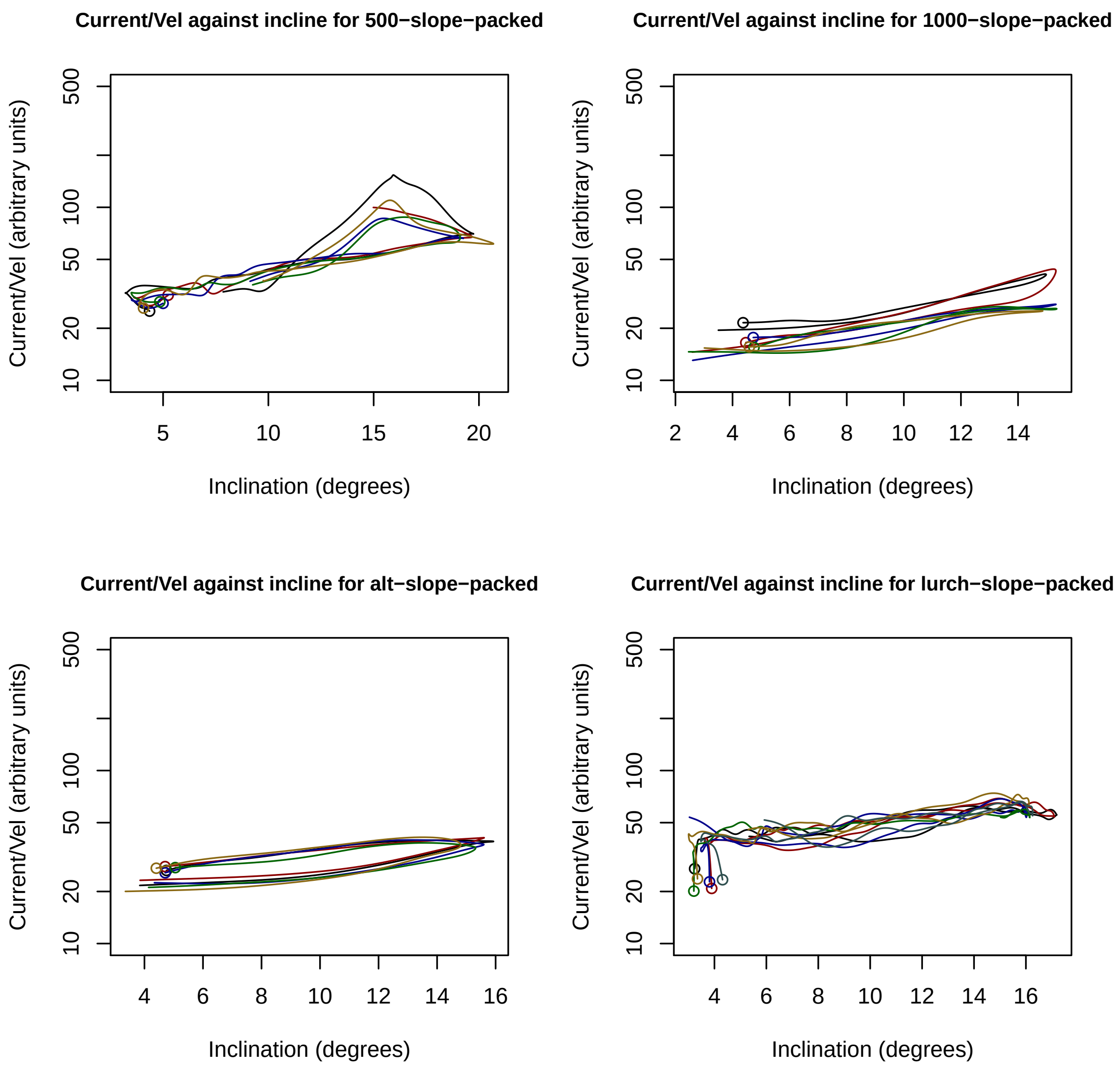


Figure 4.19: Current/velocity against incline for the packed surface. Individual runs are shown in dark colours. The first point in time is marked by a circle. This is **logarithmic in current/velocity.** Generated using the **currentVelAgainstIncline.r** script.

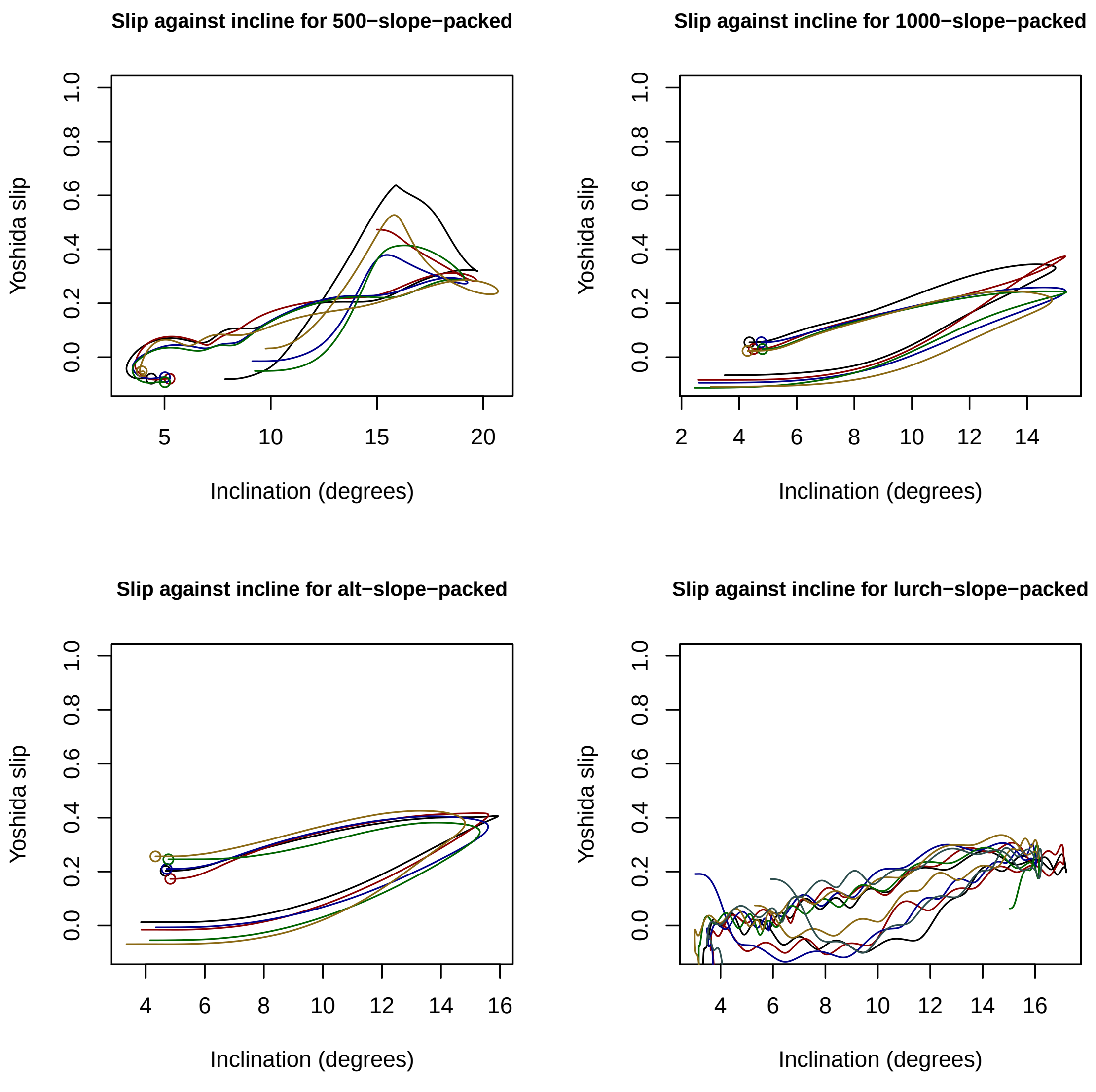


Figure 4.20: Yoshida slip against incline on the packed surface.

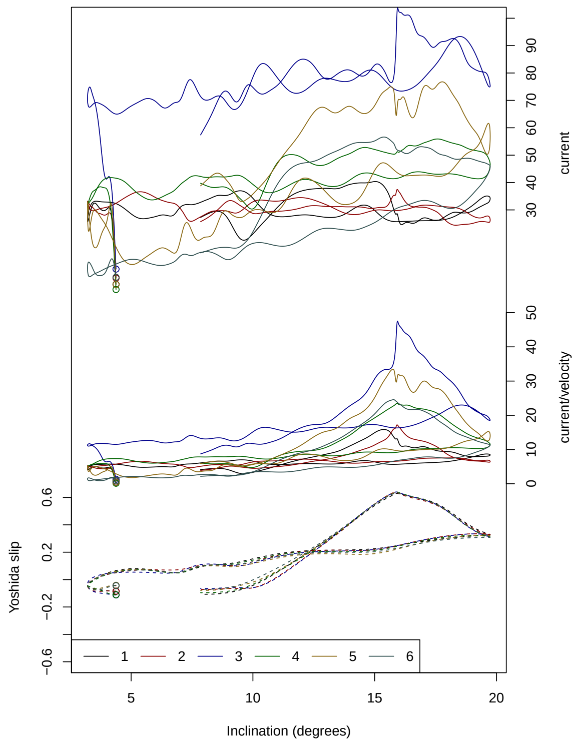


Figure 4.21: Yoshida slip and current for all wheels in a single, packed run of roll-500.

4.4.2 Loose slope experiments

Figure 4.22 shows current/velocity against incline on a loose surface. We can make the following

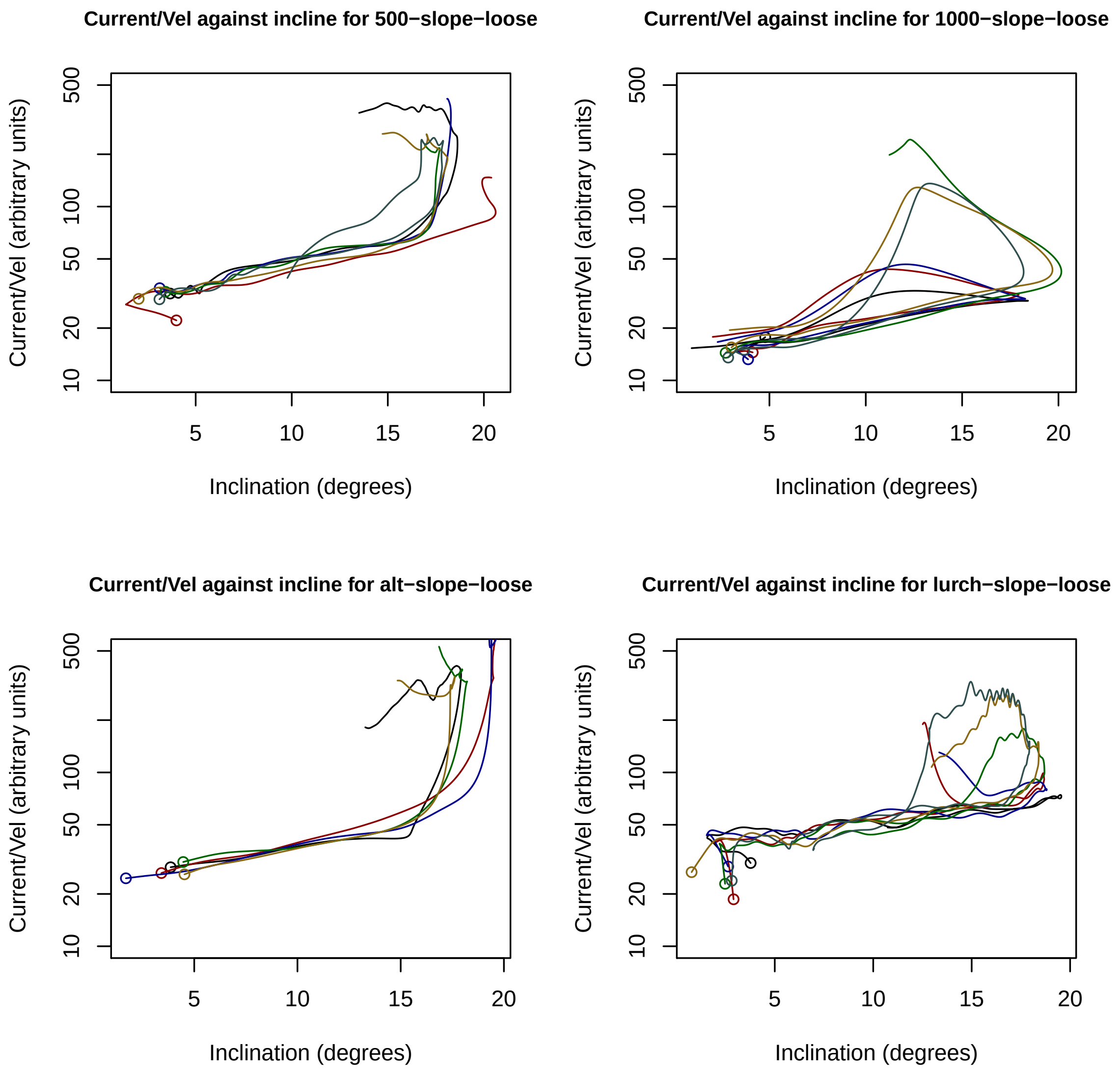


Figure 4.22: Current/velocity against incline for the loose surface. See Figure 4.19 for legend.

observations:

- There is much greater variation between runs.
- At low inclinations, the performance is roughly comparable to packed surfaces, perhaps slightly better. This may be because the surface is so loose that it either packs easily under the wheels or forms a "bow wave" in front consisting of loose material which is easily pushed aside. This can be seen in a video[2]: the bow wave is pushed to either side of the wheel.

[2]`https://www.youtube.com/watch?v=oCkwf8nakkc`

- The "crest effect" is much stronger here — all runs in alternating and all but one in roll-500 had to be aborted because the rover became stuck shortly after cresting the hill.
- At higher inclinations, roll-500 and alternating perform adequately until a given point, where the current/velocity suddenly increases. This can also be seen in the Yoshida slip plot shown in Figure 4.23. This happens at different points in each run, above a particular threshold inclination. Roll at 1000 is less prone to this effect, but two of the runs show it at an earlier threshold.

  Note that is is not the same as the crest effect — in the crest effect, the metrics rise as the inclination decreases after its peak. This is seen most clearly in the plot for roll-1000. However, the threshold effect involves the metrics rising sharply while the inclination is still increasing, as seen in roll-500.

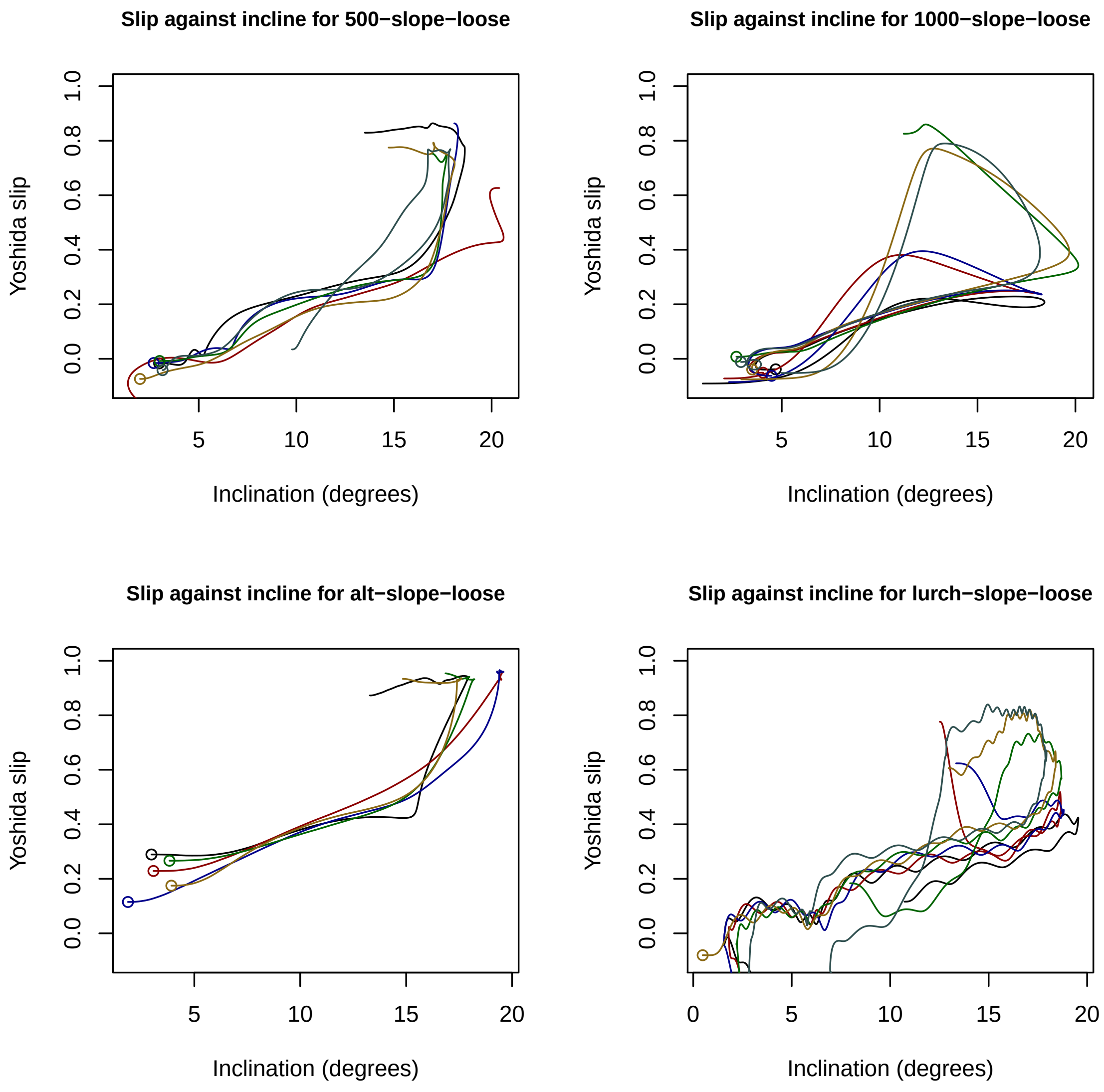


Figure 4.23: Yoshida slip against incline on a loose surface.

4.4.2.1 Causes of sudden increase in current/velocity on the loose for roll-500

An initial hypothesis is that this is due to stalling, as in the previous poor performance at roll-500. However, Figure 4.25a demonstrates that this is not the sole cause — the current/velocity increases sharply, but the Yoshida slip is also increasing rapidly.

It is currently hypothesised that this is due to *stick-slip* behaviour: at higher inclinations, gravity acts on the surface as the wheels rotate, causing a shearing force to be exerted on the loose regolith. Below a certain threshold, the soil remains a single mass. Above this threshold, the soil separates into layers which slip against each other, with the boundary layer typically being within the transition zone between loose and packed regolith. This is borne out by the fact that the run had to be aborted: once the surface starts to slip, it will remain fluid and the rover will be unable to progress. This is also shown in a video[3].

Figure 4.24 shows the forces acting on the loose regolith: the weight of the wheel (and rover) pushes down, as does the weight of the regolith itself. On a slope, these forces will shear the soil because they each have a component parallel to the boundary between loose and packed layers. Finally, the wheel itself applies a strong shearing force.

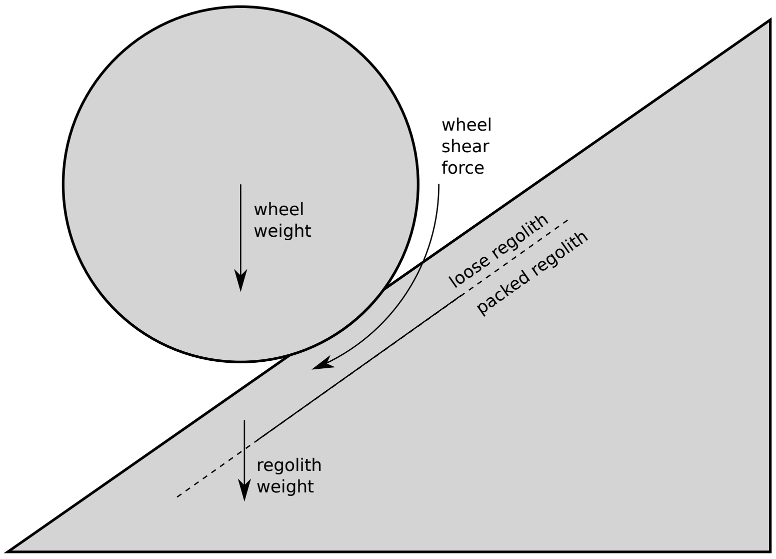
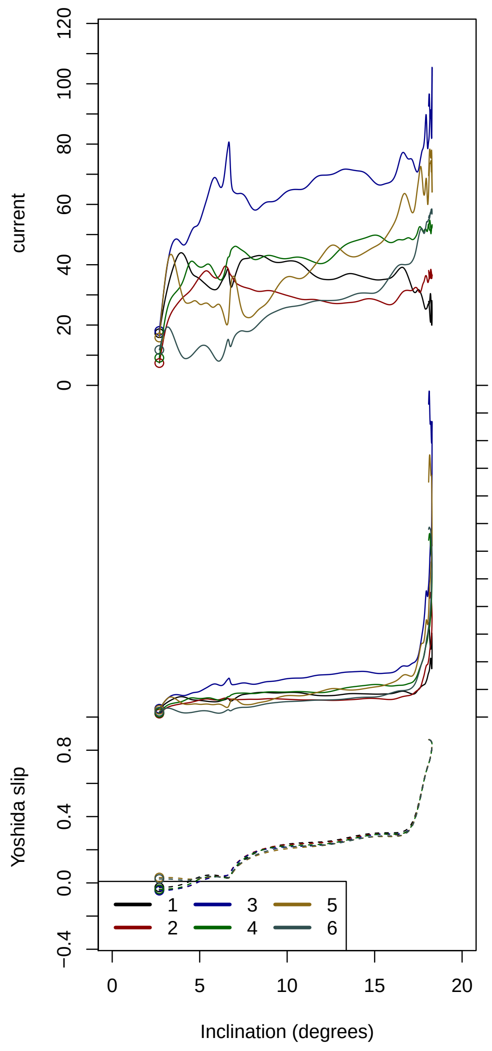


Figure 4.24: The forces acting on the boundary between packed and loose regolith on the slope with a moving rover.

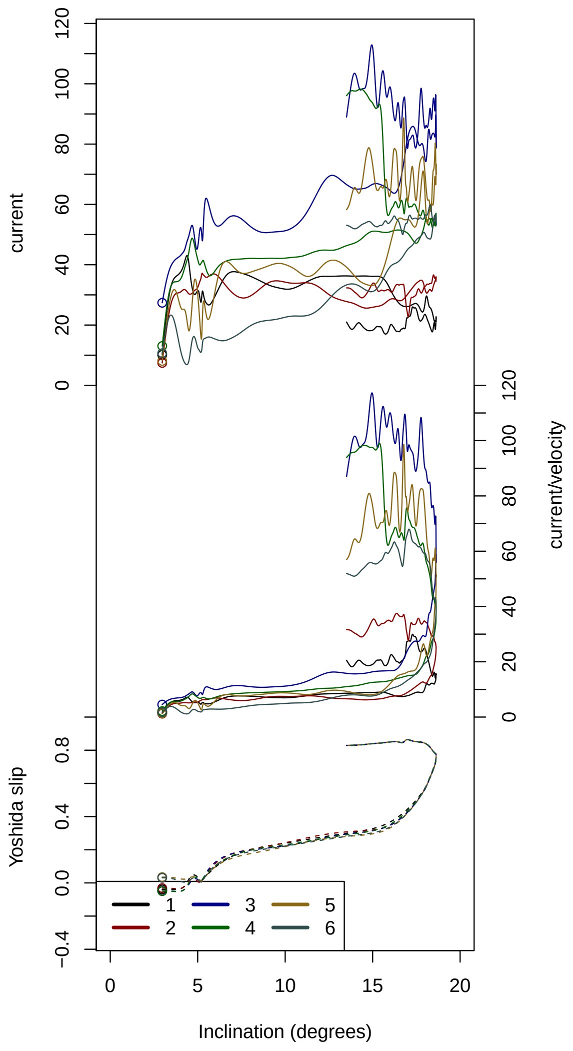
However, shearing is not the only effect — or even a major effect in all runs, as can be seen from another roll-500 run in Figure 4.25b. Here, the poor behaviour is associated with current surges, similar to those seen in Figure 4.21. This run did not have to be aborted, and the current was decreasing at the end.

It is possible that stalling and stick/slip behaviour are working together to different extents in each run: repeated stalls of the motors, plus the oscillatory nature of the motor control (exacerbated by the stalls) seen in Figure A.1 will apply frequent high shearing forces to the regolith, causing more slip. The next "stick" phase, as the layers bind again, will cause the motors to stall once more. This is consistent with the oscillatory currents seen in both figures.

[3]`https://www.youtube.com/watch?v=dajwkfjjbF8`



(a) Run in which slip-stick occurs



(b) Run in which stall may occur

Figure 4.25: Slip, current and current/velocity for two runs of roll-500 on a loose slope. Generated using **plotRun.r**.

4.4.2.2 Causes of sudden increase in current/velocity on the loose for alternating

Similar threshold behaviour occurs in the alternating gait, at roughly the same inclination. An initial hypothesis is that this is also due to a mixture of stall and stick/slip, but the data shows that this is always due to stick/slip or similar behaviour. Examples of this are shown in Figure 4.26. Neither run shows the sudden increase in current commensurate with a stall condition, although

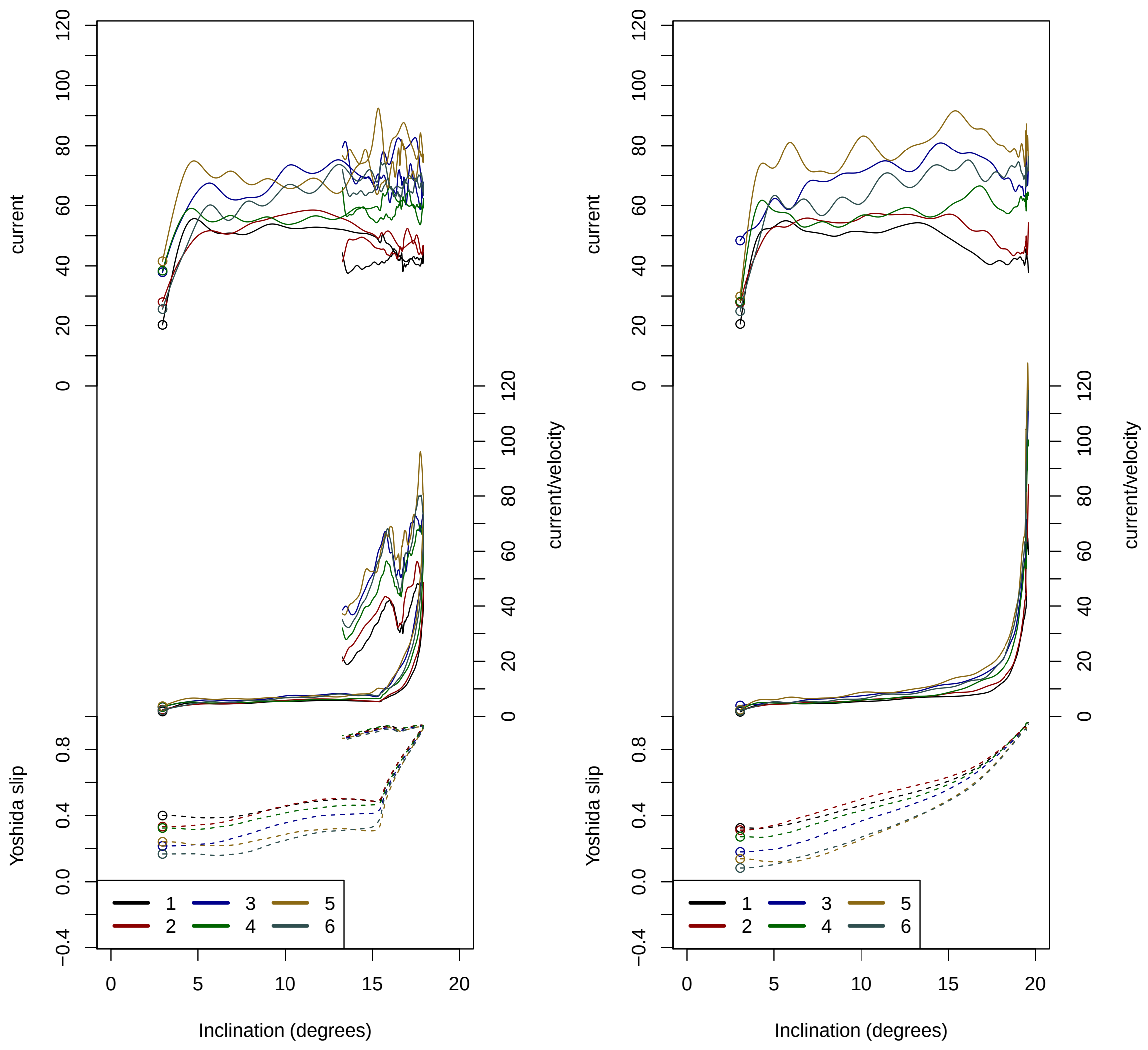


Figure 4.26: Slip, current and current/velocity for two runs of alternating gait on a loose slope.

one of the runs had to eventually be aborted because the rover's wheels had worn down to the plastic sheeting under the regolith.

An examination of the video evidence [4] shows the cause: as well as slippage on the rolling wheels moving forward, the pushing wheels are not moving the rover forwards, they are merely

[4] `http://youtu.be/V4u1E2g6orE`

pushing soil from underneath the rover behind it. This eventually digs the rover into the regolith, until the wheels cannot roll their way out. It is likely that the “push” action, combined with gravity, causes too strong a shear on the regolith for it to maintain cohesion.

4.4.2.3 Behaviour of lurching gait

The runs for the lurching gait vary a great deal, but most runs appear to show a slightly higher transition point from efficient to inefficient behaviour — around 18° of inclination, as opposed to 16° for roll-500 and alternating. We will see that only roll-1000 performs better. In addition, in about half of the runs the “crest” effect (high inefficiency at relatively low inclines on cresting the hill) appears to be lower or almost non-existent. Even in the runs which exhibit a crest effect, the rover recovers rapidly compared with other gaits. The possible cause of the variation between runs will be discussed in Section 4.5.1.

Figure 4.27 shows two runs. In Figure 4.27a, the crest effect is still very much in evidence, but recovery is rapid. In Figure 4.27b there is no crest effect and the rover is is efficient throughout, on both metrics.

4.4.2.4 Behaviour of roll-1000

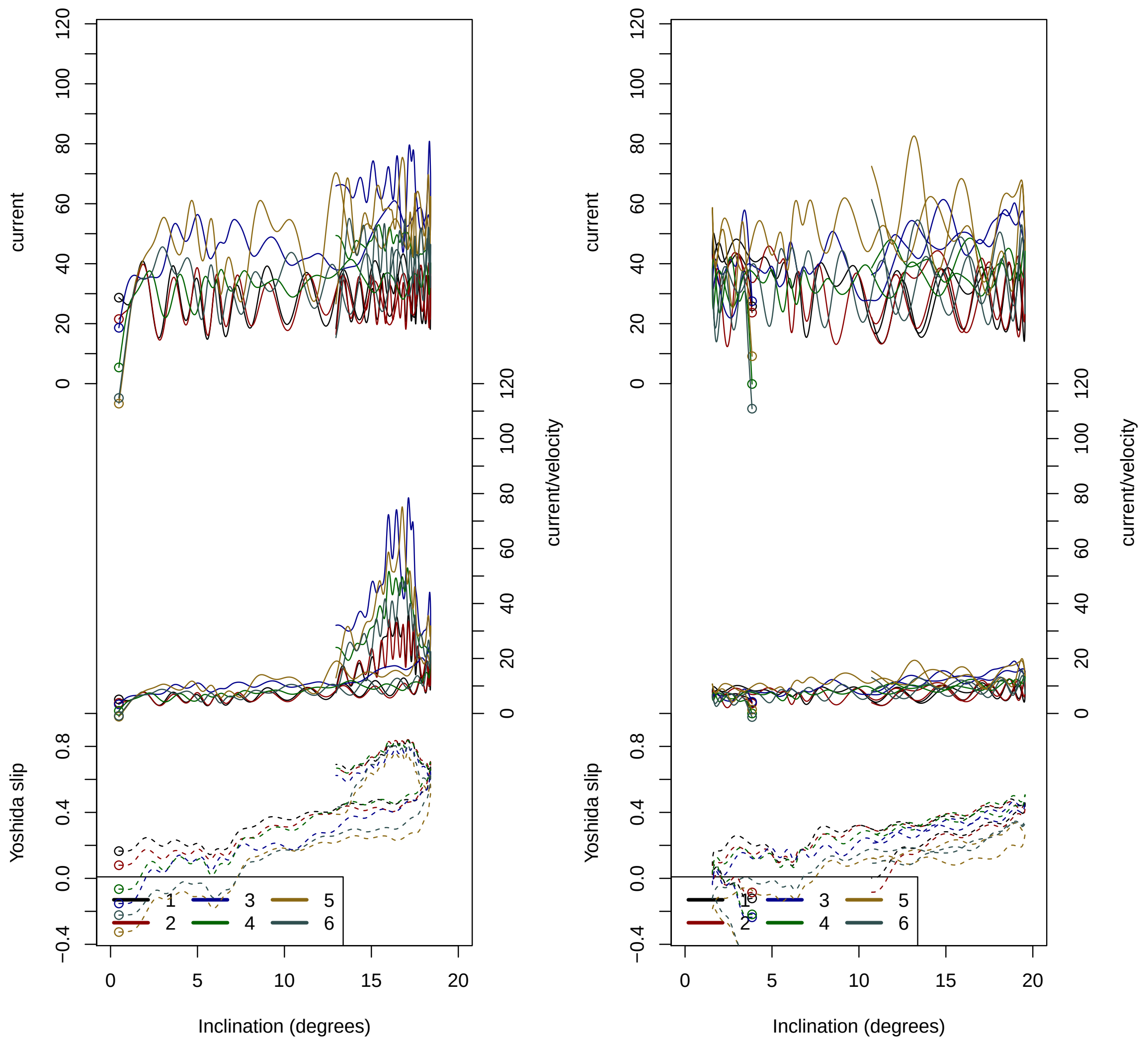
Figure 4.28 shows two runs of roll-1000. The left-hand figure shows a run in which the gait behaved well, with the current/velocity always low. The right-hand figure shows a worse run. All other runs fall somewhere between these two extremes.

Note that since all these individual run plots use a linear current/velocity scale, we can see that roll-1000 does behave much better than roll-500 — this can be obscured by the logarithmic axis in Figure 4.22.

It’s clear that the current usage is higher, as we would expect — nearly as high as the putative stall currents in Figures 4.21 and 4.25b. However, the current/velocity is still low, so the rover is still moving (current/velocity places a premium on the rover moving, as stated in Section 4.3.4).

Current usage varies a great deal across the wheels, with wheels 3 and 5 notably higher: wheel 3 is the wheel which has been reassembled as noted earlier, and wheel 5 is a rear wheel. Interestingly, wheel 6 (the other rear wheel) shows a low metric — one rear wheel is drawing far more current than the other, although both are slipping equally. Looking at other experiments (Figures 4.26 and 4.25) shows that wheel 5 has a tendency to draw more current at higher inclines (i.e. under load) than all other wheels except 3. Perhaps this wheel also has a mechanical problem.

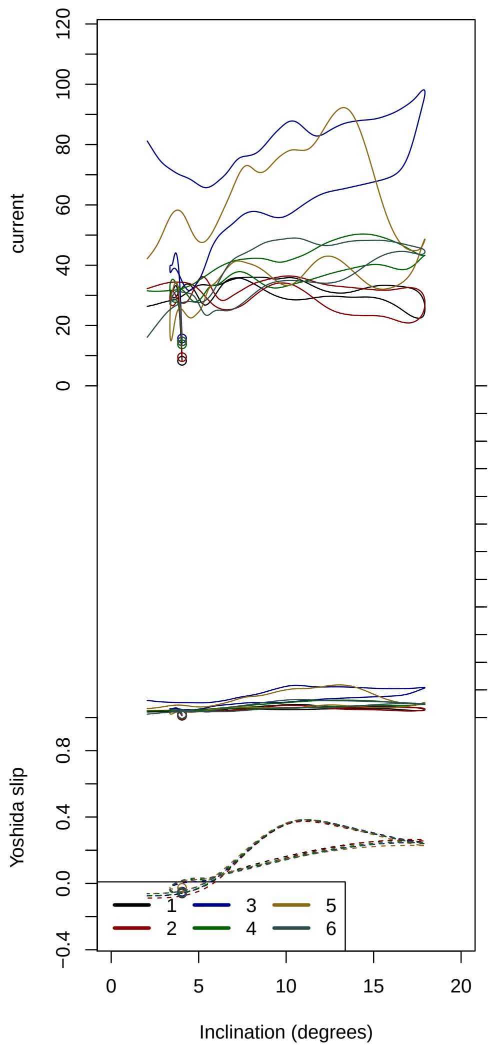
There is considerable crest effect on all runs which reached a sufficient inclination to trigger it, and this is a potential problem. The crest effect and its causes are dealt with in Section 4.5.1. In general, however, it is clear that a faster roll is more efficient than all other gaits on a constant incline.



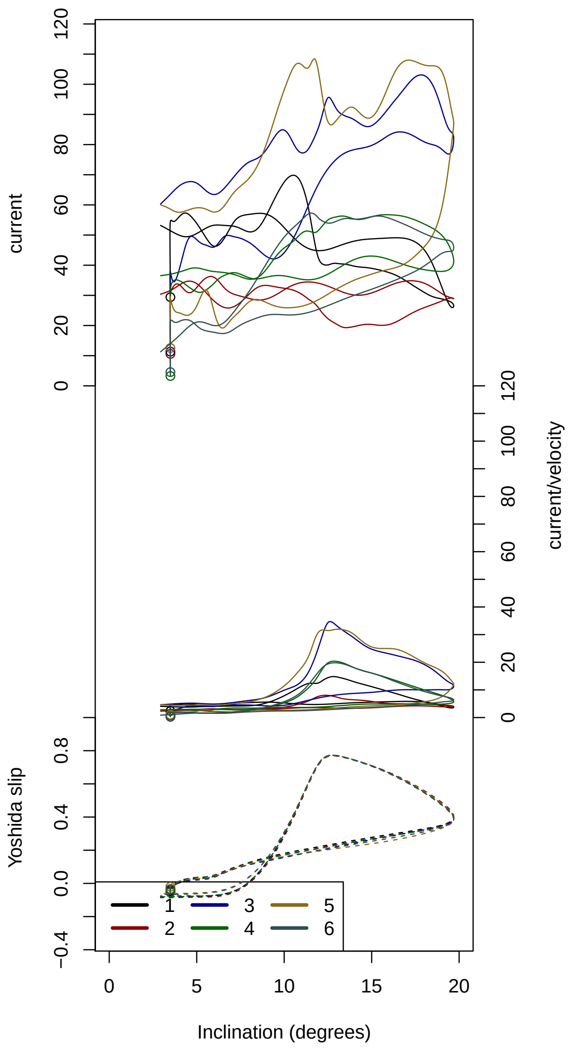
(a) Run 5, an poor run with a significant crest effect, although recovery is quicker than other gaits

(b) Run 1, an efficient run with no crest effect

Figure 4.27: Slip, current and current/velocity for two runs of lurch on a loose slope.



(a) Run with no crest effect due to low maximum incline.



(b) Run where crest effect occurs.

Figure 4.28: Slip, current and current/velocity for two runs of roll-1000 on a loose slope.

4.5 Comparison of gaits

Comparing the gaits is difficult statistically, because each experiment contains some runs which behave differently from others. A simple comparison is to calculate the mean of each metric for each run, giving a simple measure of how efficient the run was, and then rank the gaits by the maxima and minima of those means. Here, the mean can be thought of as an integral of the metric, corrected for different run durations. This will give us a comparison of how well the gaits do at their best and at their worst, and gives Table 4.1 for current/velocity, and Table 4.2 for Yoshida slip.

	minima	maxima
1000	21.316	93.803
lurch	53.156	123.050
500	63.077	210.167
alt	213.629	324.168

Table 4.1: Minima and maxima of current/velocity means of all runs of all experiments. Generated by **summarise.r**.

	minima	maxima
1000	0.108	0.429
lurch	0.213	0.478
500	0.263	0.565
alt	0.705	0.771

Table 4.2: Minima and maxima of Yoshida slip means of all runs of all experiments. Generated by **summarise.r**.

We can see that the ordering is the same for both metrics, sorted by either maximum or minimum. Fast rolling is best, followed by lurching, then slow rolling and alternating. This analysis only considers the situation over the slope as a whole, however — some gaits deal with changes in the inclination (the crest effect) better than others.

4.5.1 Crest effect

The "crest effect" is the observation that as the inclination of the rover begins to fall after cresting the hill, the metrics are at their worst. This can clearly be seen in roll-1000, whereas the poor performance in roll-500 is due to the rising inclination reaching a threshold.

The pose of the rover, and the forces acting on the rear wheel (and also all the other wheels) in this situation are shown in Figure 4.29. As well as the weight of the rover pushing the wheels into the surface, the rear wheels are being pulled into the surface by the forward motion of the rover. This will serve to increase the effect of the shear force acting on the surface under the wheel, increasing the likelihood that the wheel will slip. This effect is much more notable on a loose surface, as can be seen by comparing Figures 4.19 and 4.22, because shearing forces only cause regolith to slip on such surfaces.

Moreland states that a static wheel creates a "unified deep soil mass" beneath it which acts as a solid base, preventing slip [26]. If this is the case, then any gait which can keep the rear wheels

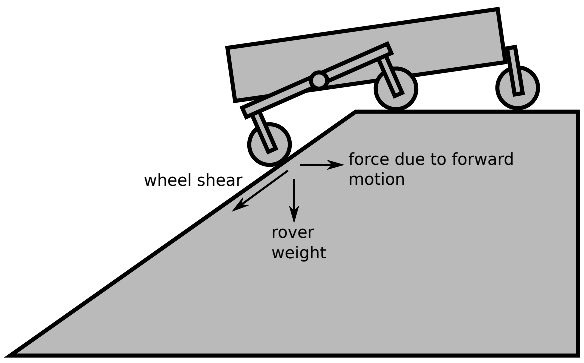


Figure 4.29: The pose of the rover when cresting the slope, showing that the rear wheels carry much of the weight (incline of slope is exaggerated).

static should crest well.

The lurching gait appears to show this behaviour for most of the runs, but this varies. There are likely to be only one or two gait cycles involved in cresting a hill, and given the dynamics of stick/slip behaviour the soil will either slip or not — there is no “partial slippage” — with the probability dependent on several factors, such as the precise packing and depth profile of the point under the rear wheels, and whether those wheels are static at the critical point where the cresting effect is strongest.

This leads to some lurch runs having almost no crest effect, while others show it strongly. However, even in these runs the rover soon returns to the “pushing” phase of the gait (see Section 2.10.2) and regains its solid base, allowing it to proceed where other gaits might remain in difficulty. This explains both the variation and the quick recovery of this gait.

Rolling gaits, however, both crest poorly. Roll-500 is already in trouble before it reaches the crest, because the soil is already starting to stick/slip at those high inclinations, possibly exacerbated by the oscillatory movement of the wheels and repeated stalls. Roll-1000 also shows high slip at the crest where sufficiently high inclinations were achieved. This is because the wheels are always moving. Alternating performs badly because the gait causes the rover to rotate slightly around the vertical axis for every movement, so a steady base is never formed; and because of the digging action mentioned in Section 4.4.2.2.

4.5.2 Conclusion of slope analysis

We can see that on any constant incline, rolling as fast as possible is best. However, rolling will suffer from crest effect on reaching the top of the hill. Lurching, while not efficient most of the time, may do better on the crest because at least two pairs of wheels are static at all times, providing the solid mass mentioned in [26].

It is possible that improvements to the lurching gait — perhaps using direct current control rather than PID for the drive motors to reduce oscillation — may increase its performance, allowing the wheels to roll smoothly across the surface without oscillation causing shearing.

The most important finding is that the pose of the rover, and its relationship to the surface — particularly how this relates to the suspension and forces acting on the wheels — is as important as the inclination and condition of the surface.

4.6 Preparing for a neuroendocrine system

To create an AES, we need to find two gaits to switch between and an interoceptive metric to use as input.

The gaits selected were the lurching gait, which behaves well on the crest of a hill; and roll-500, which performs badly, but runs at the very slow speeds likely to be required by a real rover on difficult terrain. Another factor is that roll-1000 performs sufficiently well to get to the top of the hill without requiring lurching, so an AES is not required — so for demonstration purposes we need to use the slower gait.

We cannot use current/velocity as a metric within the algorithm because velocity is exteroceptive — there is no access to it while the rover is running because of the system architecture. Therefore, we must find a proxy: an interoceptive metric which is strongly correlated to current/velocity.

An obvious candidate is the current. To test this, the data points in all runs were ranked by both current/velocity and current, and correlated using Spearman's rank coefficient, with the results shown in Table 4.3. A strong correlation is shown for both roll-500, and a fair correlation

	500	1000	alt	lurch
1	0.816	0.985	-0.180	-0.483
2	1.000	1.000	-0.775	0.747
3	1.000	1.000	-0.316	0.623
4	0.970	1.000	0.924	0.888
5	0.999	0.891	-0.348	0.915

Table 4.3: Spearman's ρ for current/velocity and current, for all data points in all runs on the loose slope.

for lurch (except on run 1). The strong correlation in roll-500 should allow for a good transition to the lurch mode. The poorer correlation for lurch mode should be relatively unimportant, but may in retrospect have caused some of the oscillations seen in the runs. It is fortuitous that lurching was chosen: the correlation for the alternating gait is much worse.

This selection of metric and gait worked well when the AES was built and briefly tested — see Section 2.11.2.

4.7 Conclusions

With the current gaits, the answers to the research questions are:

Question 1: *Is there a metric based on interoceptive sensor data which is a sensible measure of efficiency, is sufficiently accurate, and is practical on our ExoMars Concept-E chassis-based prototype rover?* **Yes and no**. The current works fairly well — provided we're not using the alternating gait! However, some oscillation (see Section 2.11.2) may be caused by the poor current-current/velocity correlation in the lurching gait. In reality, current/velocity would be far better, but would require fairly accurate external velocity measurement (though not as accurate as slip would require). Perhaps optic flow could be used, or some more complete visual localisation.

Question 2: *Is there a locomotion technique (or "gait") for the our rover, involving both the drive and walking motors, which is more efficient (under some metric) than plain rolling when used on an incline, on both packed and loose soil?* **Not with the current gaits on this rover**, where rolling fast appears to do best. It's possible that it fails at higher inclinations, or that lurching can be improved to a point where it competes (using direct current control motors to avoid oscillations).

Question 3: *If so, is it possible to design an artificial endocrine or neuroendocrine system, controlling a subsumption architecture, which will switch between locomotion techniques automatically as the given metric demands?* **Yes**, if a better gait can be found, or a steeper slope upon which rolling may fail is used. For demonstration purposes, rolling slowly was used instead to show the system would work.

The behaviour of the rover on the surface was found to be far more complex than anticipated. In particular, two important effects were noted on the slope:

- **Threshold effect**: certain gaits tolerate the slope well up to a threshold, when the surface starts to slip because of shear forces from the rover's weight and wheels.
- **Crest effect**: the rover is less efficient when the inclination is falling at the top of the slope, when it is cresting the hill.

While some attempts have been made to analyse this behaviour, much more work needs to be done using physics models of the soil.

Chapter 5

Critical Evaluation

This was an extremely tough project, with many challenges in engineering, experimentation, and data analysis. I'm pleased with how it went, but there's a great deal I would do differently if I had the chance.

5.1 Initial change of direction

My initial project was an attempt to built a genetic algorithm to construct artificial neuroendocrine systems — Mark and I rapidly realised this was far too ambitious[1]. A chance conversation with Dave Barnes led to a change of direction to the current project.

However, about two weeks were spent on this initial idea, including getting a good way into an ODE simulation of the rover. I suspect blundering straight into this was probably a bad idea, but I had to start somewhere.

5.2 Preliminary research

I'd already worked with artificial endocrine systems, but hadn't built one from scratch, so I had an idea of the literature in that field before I started. Rover gaits were harder to research — there was a great deal of work on wheeled robots, and on legged robots, but relatively little on hybrids. I was lucky to find Moreland's paper [26] which gave me my first idea for a workable gait, and Yoshida's work [40] helped me with a workable and enlightening metric.

However, the gait analysis would have benefitted more from some proper mechanics. Unfortunately, it was very difficult to find a way into this field ("terramechanics") — the maths was far beyond me. I was only able to describe what was going on in each gait, and guess at the reason, without being able to analyse it properly. This was disappointing.

Another blind alley was the Libkoki work — I suspected the Vicon was going to be unavailable, so worked on an alternative for a few days. My time could have been better spent.

[1]It's now the subject of my PhD research!

5.3 Approach

I believe my overall approach was sound: my primary research question ("can I build an AES switching rover?") led naturally into the other two questions ("can I identify a metric?" and "are there suitable gaits for switching?").

That naturally led to a series of experiments — I'm glad I ran some preliminary tests, which at least told me that the packed and loose slopes might be different. I had no idea how different they were until after the experiments had been finalised, and the depth of the differences wasn't revealed until the analysis. The experiments provided me with enough data to test my metrics, and then to analyse the gaits.

By far the largest part of the work was the analysis of the gaits — building the gaits was straightforward, as was the AES, but analysing the complex behaviour took a great deal of work. This could have benefitted from more soil mechanics theory, and certainly from more control in the experiments (see below).

The final part of the process should have been analysing the completed AES, but unfortunately I ran out of time.

5.4 Technology

Although I kept worrying that it was foolish, I'm glad I used my own language Angort to run the experiments. Several times I started work on implementations of gaits and the AES in C++, and each time the code became very messy and I switched back. I'm a very experienced C++ programmer, and it just isn't suited to that kind of coding.

Angort already existed, and was already linked to the rover, so it was a natural choice. Since I originally wrote it as a control language, it has gained a great deal of power — I've worked on it a lot over the couple of years as a pet project. I was very surprised to find just how expressive and powerful it was in a real project.

My only concern is that using a novel language limits the comprehensibility of my code. To that end, I've added a partial description of the language as an appendix and made sure everything is commented (although it could be commented even more).

The Vicon motion capture system was very difficult to use, with a complex and unreliable Windows program to capture the data. It took quite some time to get to grips with. Most of my problems involved having to re-run an experiment because Vicon had crashed or was giving spurious results.

5.5 Experimentation

I'm fairly happy with the experimental protocol, and the use of the monitor to keep an eye on the rover. Probably about 20 hours of experiments were done, not including the preparation and analysis time, which probably doubles this. Having a checklist helped avoid re-runs and permitted some multitasking.

However, repeatability was a big problem. I should have been more careful to ensure the rover was always started at the same point (in the y axis as well as x in Vicon space), and that the condition of the soil was always the same. The latter proved extremely difficult — rollers would have helped immensely.

The entire project would have benefited from better control over the environment, with properly measured and calibrated slopes and soil profiles. I imagine this would be extremely hard to achieve. Also, I should have tried once more to get a definitive value for the encoder ticks / circumferential velocity ratio; but by the time I realised it was a problem, it was too late.

5.6 Analysis

One problem with this project is that despite the title, it's mostly about analysing gaits and their failure modes. The AES part was actually completed in about a week at the end of the experimental time allocated, and it shows. More time would have permitted an analysis of the AES performance.

The data has been smoothed a great deal, which always looses some information. A separate analysis of what goes on inside a gait cycle would have been useful, using data which isn't smoothed, but that would have been a huge piece of work.

The analysis of the gaits is fair, but there's not enough mathematical rigour. I'd like have discussed the "crest" and "threshold" effects in more detail, but time and space were limited. However, the effects are interesting, and come nicely out of the metrics. I'm pleased with my qualitative analysis of them.

Statistics could have been used to provide more quantitative data, but given the distribution of the data on the slope, non-parametric statistics had to be used. I'm happy with using Spearman's ρ to test the metrics, and also to correlate the current/velocity with current; but surely more could have been done. My statistics knowledge is near nil (but increasing) so it was hard to know exactly what.

5.7 Achievements and aims

This project has achieved its aim: we have an AES-controlled switching gait, and a demonstration that it works. We also have a useful pair of metrics, which have been used to analyse four gaits in some depth — in fact, this took up most of the project.

However, given that the answer to "is there a gait which is more efficient on the slope than plain rolling?" is actually "no" (roll-1000 is always best) the AES feels like a bit of a cheat: I had to use roll-500 to get it to work. I'm not happy with that, even though it proves the concept.

The alternative would have been to try to improve the lurching gait so that it slipped less. I'm convinced this is possible, but it would require modifying and re-flashing the firmware on the rover to permit direct current control. There was no time, and it could have failed.

Appendices

Appendix A

The Blodwen rover

In this section, some of the subsystems of the ExoMars rover prototype introduced in section 2.2 are described in more detail. These are specific to our prototype, not to the final ExoMars rover.

A.1 Introduction

As supplied, the rover had no control systems whatsoever. Control was achieved by manually powering the motors via a tethered switch box. As part of industrial year work, the author designed and built a complex microcontroller-based control system.

A.2 Motors and gearing, and PID control problems

Concept-E uses three motors on each wheel. These are small Maxon motors, drawing approximately 0.5A at full load, and are very highly geared. This, combined with the friable simulated regolith, which changes the force reacting against the motor drastically under different conditions, made tuning the motors very challenging. There are still a number of problems — the actual drive motor speed oscillates rapidly around the required speed, and both steer and lift motors overshoot markedly. This can clearly be seen in Figure A.1.

It is likely that the current simple PID control is insufficient, and improvements such as feed-forward, gain scheduling or cascaded control should be considered for future work.

These problems impact considerably on the experiments — walking requires changing the speeds and positions of the motors reasonably accurately and quickly, and in practice this was very difficult. It is likely that the walking gaits analysed would be far more efficient with improved motor control.

A.3 Control

The rover originally had no onboard electronics, being controlled simply by manual switches on a tethered remote control unit. A control system was built by the author during his industrial

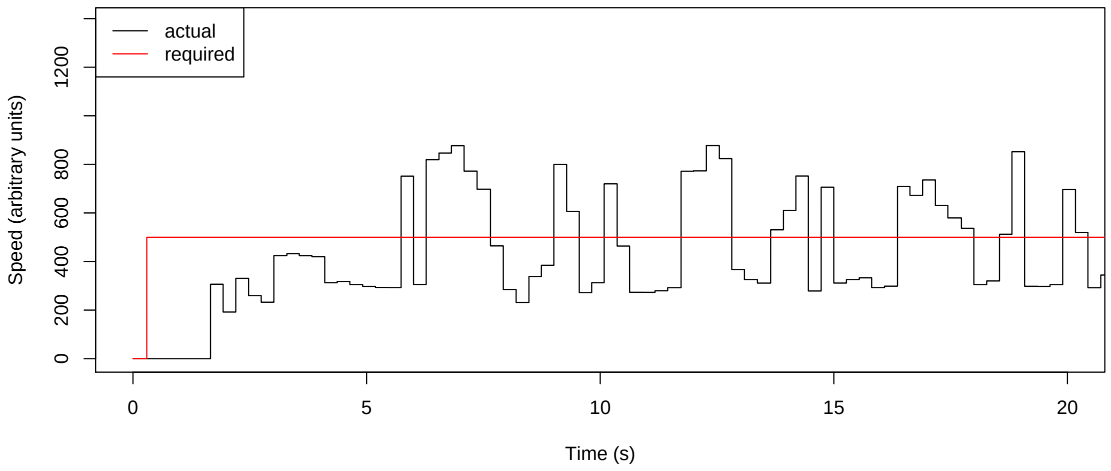


Figure A.1: Actual drive motor speed for wheel 1 on packed soil, commanded to a speed of 500 arbitrary units, demonstrating the typical extreme oscillations.

year, based around a three-layer architecture: a set of nine microcontroller/motor driver boards[1] communicating over an I^2C bus to an Arduino Uno master, which communicates over USB with a C++ library on the onboard PC. This is described fully in the relevant appendix. For ease of control, a simple interface was written to to connect this library to the Angort scripting/control language. This language surprisingly proved very powerful for experimentation.

A.4 On-board equipment and power

The rover is powered by a 8800mAh lithium polymer battery — it has room for two, but it was considered more efficient to use only one, while the other recharged. On board, there are two Fit PC low power computers (only one of which is in use) and a wireless router. These typically draw 1.5A when the rover is not moving. With drive and lift motors actuated, this can reach up to 4.5A.

A.5 Current measurement

Conveniently, each L293D motor driver chip is equipped with current sense lines. These are connected to analogue inputs on the microcontroller, providing us with a current measurement. Unfortunately the PWM control applied to the motors, and the innate oscillation of the motor due to the tuning problems, meant this measurement oscillated wildly. Therefore, the firmware

[1]Sparkfun ROB-09571 units, consisting of an L293D dual H-bridge motor driver and an ATMega328P microcontroller

imposes a large amount of smoothing on the measurement:

$$I_t = ki_t + (1-k)I_{t-1}$$
$$k = \begin{cases} 0.1 & \text{if } \frac{di}{dt} > 0 \\ 0.01 & \text{if } \frac{di}{dt} \leq 0 \end{cases}$$

where I_t is the output measurement at time t, i_t is the input measurement at that time, and k is the smoothing constant. This is the *exponential moving average* [5], because it is is an average of all previous values, weighted by factors which decrease exponentially with distance in time.

It should be noted that k is different if the input measurement is increasing or decreasing. Thus, the hysteresis on increasing current is much smaller than that on decreasing current. This allows rapid upward changes to be quickly spotted, so overcurrent can be avoided, while continuing to smooth the output.

A.6 Other sensors

Other sensors aboard the rover are:

- **temperature sensors** on each of the motor driver chips, measuring the temperature associated with powering pairs of motors;
- **suspension module potentiometers** measuring the angles of the suspension joints.

Further experiments could make use of the chassis potentiometers, perhaps to monitor when the rover is starting or finishing an incline, and could be used in an interoceptive metric for control. However, none of these sensors are used in the current study.

A.7 Software

As mentioned above, the rover is controlled by a set of microcontrollers (one for each pair of motors), all of which are sent commands via a single master Arduino Uno, which in turn receives commands from the PC via USB, configured as a simple serial device. All the microcontrollers have custom software written in C++, documented in more detail in [14].

The PC to master commands, in their raw form, consist of requests to read or write "registers" — variables stored in the microcontrollers. For example, the following command writes four values to four different registers:

PC to master: `0d 32 04 00 ff 01 ff 02 01 00 03 02 00`
Response: `00`

- `0d 32`: 13 bytes, command 2 (write) for slave 3
- `04`: contains 4 writes
- `00 ff`: write `ff` to register 0
- `01 ff`: write `ff` to register 1

- `02 01 00` : write `0001` to register 2
- `03 02 00` : write `0002` to register 3

(Response is just zero to acknowledge).

This system is clearly very complex, so a C++ library was written to encapsulate the commands.

This system uses a top-level `Rover` class, which is instantiated as a singleton. This can then be used to gain access to objects in the `Motor` class hierarchy to control the motors, which can in turn be requested for data objects. Calling `update()` on the `Rover` singleton will cause request to read all sensors to be sent, updating these data objects.

A brief example:

```
int main(int argc,char *argv[]){
    Rover r;

    try {
        // set up the rover given the comms port and the baud rate.
        r.init("/dev/ttyACM0",115200);

        // send default calibration
        r.calibrate();

        // some parameter data we're going to change (for illustration)
        MotorParams params = {
            0.004,0,0,   //PID
            0,0,         //integral cap and decay
            300,         //overcurrent threshold
        };

        // change parameters on the drive motors
        // and set a speed for them

        for(int i=1;i<=6;i++) { // motors are 1 to 6 as in the documentation
            Motor *m = r.getDrive(i); // get each drive motor

            // get a pointer to its parameters
            MotorParams *p = m->getParams();

            // copy some other data into them
            *p = params;

            // and send the changes
            m->sendParams();

            // and set a speed
            m->setRequired(1000);
        }

        for(;;){
            usleep(10000); // wait 1/100 s
            r.update(); // update the rover

            // get drive motor 1 data
            DriveMotorData *d = r.getDriveData(1);
            printf("%f\n",d->actual); // print actual speed
        }

    } catch(SlaveException e) {

        // slave exceptions are thrown by protocol and comms errors
        printf("Error in rover communication: %s\n",e.msg);
        return 0;
    }
}
```

Appendix B

Information on the rover monitor system

See Section 2.3 for general information about the monitoring system.

B.1 Incoming data protocol

The monitor reads a UDP port, which is fed variable data as lines of text in key-value form by the remote system. Not every line contains all the possible variables; if no data is received for a given variable, its value is assumed not to have changed. This helps minimize bandwidth. All lines must contain a variable called "time", however, which is typically the UNIX timestamp. Here are some example lines, captured from this project's experiments:

```
time=1393848445.992669 go=1.00
time=1393848445.992846 ptime=146.352093
time=1393848445.992959 actual1=0.00 req1=-2200.00 current1=0.00 lift1=-1.114 steer1=-0.036 liftcurrent1=17.00 odo1=0
time=1393848445.993090 actual2=0.00 req2=-2200.00 current2=0.00 lift2=-0.700 steer2=-1.068 liftcurrent2=8.00 odo2=0
time=1393848445.993221 actual3=0.00 req3=0.00 current3=0.00 lift3=-0.946 steer3=0.927 liftcurrent3=3.00 odo3=0
time=1393848445.993342 actual4=0.00 req4=0.00 current4=0.00 lift4=0.134 steer4=0.061 liftcurrent4=0.00 odo4=0
time=1393848445.993464 actual5=0.00 req5=0.00 current5=0.00 lift5=-0.476 steer5=0.671 liftcurrent5=1.00 odo5=0
time=1393848445.993598 actual6=0.00 req6=0.00 current6=0.00 lift6=0.427 steer6=0.366 liftcurrent6=15.00 odo6=0
time=1393848445.993717 temp1=6.500443
time=1393848445.993815 temp2=8.500015
time=1393848445.993914 temp3=9.999695
time=1393848445.994012 temp4=7.000336
time=1393848445.994109 temp5=8.000122
time=1393848445.994205 temp6=8.500015
time=1393848445.994299 temp7=8.000122
time=1393848445.994396 temp8=8.500015
time=1393848445.994490 temp9=7.500229
time=1393848446.270672 go=1.00
time=1393848446.270839 ptime=146.630085
```

Inside the rover's Angort application a simple function called `udpwrite()` allows any part of the code to write data to the monitor, automatically preceded by a timestamp. For example, the code in the rover which produces most of the data above is the following, which is called periodically by the application in idle mode, and by the Angort `handleudp` command when in a script:

```
/// send a standard block of UDP data, and
/// process any incoming messages

void handleUDP() {

    udpServer.poll(); // check for incoming
    /// send the special properties, whose
    /// values came from the monitor in
    /// the first place, for confirmation.

    extern void sendUDPProperties();
    sendUDPProperties();

    // get elapsed time since program stat

    struct timespec t;
    clock_gettime(CLOCK_MONOTONIC,&t);
    double diff=time_diff(progstart,t);

    // send this time as "ptime"

    udpwrite("ptime=%f",diff);

    // now send all sensor data

    for(int w=1;w<=6;w++){
        DriveMotorData *d = r->getDriveData(w);
        DriveMotor *dm = r->getDrive(w);
        SteerMotorData *s = r->getSteerData(w);
        LiftMotorData *l = r->getLiftData(w);

        udpwrite("actual%d=%f req%d=%f current%d=%f lift%d=%f steer%d=%f\
          liftcurrent%d=%f odo%d=%d",
                w,d->actual,
                w,dm->getRequired(),
                w,d->current,
                w,l->actual,
                w,s->actual,
                w,l->current,
                w,d->odometer
                );
    }

    // also send temperature data

    MasterData *m = r->getMasterData();

    for(int i=1;i<10;i++){
        udpwrite("temp%d=%f",i,m->temps[i] - m->temps[0]);
    }
```

In addition, extra variables can be output at any time through the scripting system; the "go" variable, for example, reflects whether the rover believes it is currently running an experiment — this is an example of a feedback variable (see below).

B.2 Annotated example monitor configuration file

This is a sample of a very simple monitor script, showing some of the available widgets. Other useful widgets include a compass and a map (which can control a powerful waypoint management system), which were used in robotic boat control. A full description of the syntax of monitor configuration files can be found at [12].

```
#monitor

# the above line sets up syntax highlighting and indenting in
# my editor, MicroEmacs.

# first comes a list of data variables the system expects in the
# packets being sent from the remote system.

var {

    # Currently only float values are supported - we can emulate
    # others with these.

    # "a" is a variable with range -1 to 1. We\u2019ll store 100 values
    # for it in a cyclic buffer, so we can graph 100 values back.

    float a 100 range -1 to 1

    # Similarly, "b" is a 100-long buffer which takes values from -1
    # to 1.

    float b 100 range -1 to 1
}

# now a list of windows and the widgets in them. A common configuration
# is multiple fullscreen windows, switched between using key presses.

window size 800,800
{
    # this frame is at 0,0 (top left) and made up of 8x4 squares

    frame 0,0,8,4 {

        # at top left in this frame, a gauge for variable "a". The
        # range is obtained from the variable.

        gauge 0,0 { var a }

        # to the right of it, a gauge for variable b.
        gauge 1,0 { var b }

        # and to the right of that, a gauge for the expression "a+b". When
        # we use an expression as a data source we must specify
        # a range.

        gauge 2,0 { expr "a+b" range -2 to 2}

        # below all these, a graph which is 8x2 so it fills the row

        graph 0,2,8,2 {
            time 30
            var a { col red width 2}
            var b { col yellow width 2}
        }
    }
}
```

This graphs two variables a and b in the bottom part, and in the top part shows gauges of the current values and their sum. The running monitor with this configuration is shown in Figure B.1.

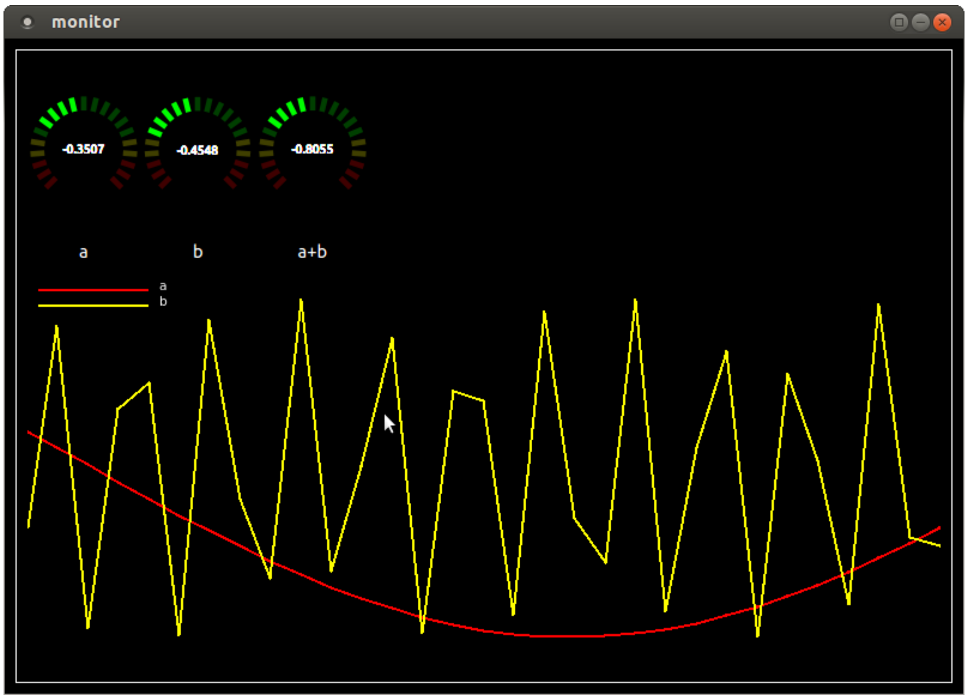


Figure B.1: Example of a simple monitor configuration.

B.3 The rover monitor configuration file

This listing shows the monitor configuration file for the experiments in the current study. See Figure 2.3 for a screenshot of this configuration.

```
#monitor

# This is a configuration file for the (originally) minty2 monitor program.
# It sets up monitoring for the rover

# variables are considered invalid if no data has been received for
# 10 seconds, and any widgets involving those variables will show the "no data" state
    .

validtime 10


#
# A list of the variables expected from the rover, how many
# values should be stored in the associated buffer, and the
# range of the values. The only type currently supported is
# floating point.
#

var {
    # how long the scripting system has been running

    float ptime 10 range 0 to 100000

    # actual drive motor speeds

    float actual1 10000 range -2500 to 2500
    float actual2 10000 range -2500 to 2500
    float actual3 10000 range -2500 to 2500
    float actual4 10000 range -2500 to 2500
    float actual5 10000 range -2500 to 2500
    float actual6 10000 range -2500 to 2500

    # required drive motor speeds

    float req1 10000 range -2500 to 2500
    float req2 10000 range -2500 to 2500
    float req3 10000 range -2500 to 2500
    float req4 10000 range -2500 to 2500
    float req5 10000 range -2500 to 2500
    float req6 10000 range -2500 to 2500

    # actual lift motor positions

    float lift1 10000 range -45 to 45
    float lift2 10000 range -45 to 45
    float lift3 10000 range -45 to 45
    float lift4 10000 range -45 to 45
    float lift5 10000 range -45 to 45
    float lift6 10000 range -45 to 45

    # actual steer motor positions

    float steer1 10000 range -60 to 60
    float steer2 10000 range -60 to 60
    float steer3 10000 range -60 to 60
    float steer4 10000 range -60 to 60
    float steer5 10000 range -60 to 60
    float steer6 10000 range -60 to 60

    # drive motor currents
```

```
float current1 10000 range 0 to 100
float current2 10000 range 0 to 100
float current3 10000 range 0 to 100
float current4 10000 range 0 to 100
float current5 10000 range 0 to 100
float current6 10000 range 0 to 100

# lift motor currents

float liftcurrent1 10000 range 0 to 100
float liftcurrent2 10000 range 0 to 100
float liftcurrent3 10000 range 0 to 100
float liftcurrent4 10000 range 0 to 100
float liftcurrent5 10000 range 0 to 100
float liftcurrent6 10000 range 0 to 100

# wheel odometry in encoder ticks/256

float odo1 10000 range 0 to 1000
float odo2 10000 range 0 to 1000
float odo3 10000 range 0 to 1000
float odo4 10000 range 0 to 1000
float odo5 10000 range 0 to 1000
float odo6 10000 range 0 to 1000

# driver chip temperatures. Chip to motor mappings are:
#  1 = drive/steer for wheel 1
#  2 = drive/steer for wheel 2
#  3 = lift/lift for wheels 1 and 2
#  4 = drive/steer for wheel 3
#  5 = drive/steer for wheel 4
#  6 = lift/lift for wheels 3 and 4
#  7 = drive/steer for wheel 5
#  8 = drive/steer for wheel 6
#  9 = lift/lift for wheels 5 and 6

float temp1 10000 range 2 to 30
float temp2 10000 range 2 to 30
float temp3 10000 range 2 to 30
float temp4 10000 range 2 to 30
float temp5 10000 range 2 to 30
float temp6 10000 range 2 to 30
float temp7 10000 range 2 to 30
float temp8 10000 range 2 to 30
float temp9 10000 range 2 to 30

# confirmation of "go" switch, relays the rover's copy
# of "go" which is a variable sent from the laptop.

float go 10 range 0 to 2

# confirmation of the "scf" value setting, also sent from
# outside. It's used in hormone switching simulation tests
# to modify the drive speed -> current increase mapping.
# It stands for "Simulation Current Factor."

float scf 10 range 0 to 20

# level of the primary "switching" hormone

float hlevel 10000 range 0 to 1

# output of the switching hormone's output neuron;
# above a certain level gait switch occurs

float hout 10000 range 0 to 1

# level of the input hormone, which is released by the current
```

```
    # and serves to smooth it. This in turn releases the switching
    # hormone.

    float hlevel2 10000 range 0 to 1
}


#
# There now follow widget definitions.
#

window
    { frame 0,0 borderless {
        frame 0,0 {

            # gauges showing required and actual drive speeds

            gauge 0,0 { var req1 }
            gauge 1,0 { var req3 }
            gauge 2,0 { var req5 }
            gauge 0,1 { var req2 }
            gauge 1,1 { var req4 }
            gauge 2,1 { var req6 }

            gauge 0,2 { var actual1 }
            gauge 1,2 { var actual3 }
            gauge 2,2 { var actual5 }
            gauge 0,3 { var actual2 }
            gauge 1,3 { var actual4 }
            gauge 2,3 { var actual6 }

        }
        frame 1,0 {

            # gauge currents

            gauge 0,0 { var current1 }
            gauge 1,0 { var current3 }
            gauge 2,0 { var current5 }
            gauge 0,1 { var current2 }
            gauge 1,1 { var current4 }
            gauge 2,1 { var current6 }
            gauge 0,3 { var liftcurrent1 }
            gauge 1,3 { var liftcurrent3 }
            gauge 2,3 { var liftcurrent5 }
            gauge 0,4 { var liftcurrent2 }
            gauge 1,4 { var liftcurrent4 }
            gauge 2,4 { var liftcurrent6 }
        }

        frame 2,0 {

            # actual lift motor positions

            gauge 0,0 { var lift1 }
            gauge 1,0 { var lift3 }
            gauge 2,0 { var lift5 }
            gauge 0,1 { var lift2 }
            gauge 1,1 { var lift4 }
            gauge 2,1 { var lift6 }

            # three momentary buttons, which each perform
            # "special" actions in the monitor rather than
            # sending messages to the rover. These start
            # and stop the logging system, and quit the
            # monitor

            momentary 0,2 { title "startlog"
```

```
                special "startlog"
                key "s"
                size 80,80}

            momentary 1,2 { title "stoplog"
                special "stoplog"
                key "k"
                size 80,80}

            momentary 2,2 { title "quit"
                special "quit"
                key "q"
                size 80,80}

            # a gauge which shows the average temperature -
            # the range must be set explicitly because the
            # monitor currently cannot infer it.

            gauge 3,0 {
                expr "(temp1+temp2+temp3+temp4+temp5+temp6)/6.0"
                range 0 to 30
                title "AVG TEMP"
            }

            # a number display showing the average wheel odometry

            number 3,1 {
                expr "(odo1+odo2+odo3+odo4+odo5+odo6)/6.0"
                # range irrelevant to number widgets, but must
                # be supplied because of the syntax.
                range 0 to 1
                title "AVG ODO"
            }

            # this switch toggles the "go" value, and sends the
            # new value to the rover. The switch also shows
            # whether the switch setting agrees with the value
            # of the "go" variable being sent from the rover.

            switch 3,2 {
                out go
                var go
                title "EXP GO"
                immediate
                key "g"
            }

            # the SCF slider, which sends "scf" to the rover
            # and confirms it with "scf" being received from the rover.

            slider 4,0,1,3 {
                title "SCF"
                out scf
                var scf
                vertical immediate
                initial 0.07
                range 0.02 to 0.2
            }
        }

        # the bottom half of the window

        frame 0,1,3,1 {

            # a graph showing total drive and lift currents
            # and hormone levels

            graph 0,0,5,1 {
                time 100
                expr "(current1+current2+current3+current4+current5+current6)/6.0"
                    range 0 to 100 {col red width 3}
                var hlevel2 {col green width 3}
                var hlevel {col white width 3}
                expr "(liftcurrent1+liftcurrent2+liftcurrent3+liftcurrent4+
                    liftcurrent5+liftcurrent6)/6.0" range 0 to 100 {col blue width 3}

                var lift1 {col blue width 1}
                var lift2 {col blue width 1}
                var lift3 {col red width 1}
                var lift4 {col red width 1}
                var lift5 {col yellow width 1}
                var lift6 {col yellow width 1}
            }

            # gauge showing switching hormone level

            gauge 5,0 { var hlevel }

            # gauge showing total current

            gauge 6,0 {
                expr "current1+current2+current3+current4+current5+current6+
                    liftcurrent1+liftcurrent2+liftcurrent3+liftcurrent4+liftcurrent5+
                    liftcurrent6" range 0 to 1000
                title "current"
            }

            # gauge showing input hormone level

            gauge 7,0 { var hlevel2 }

            # a status indicator which is black if
            # the switching hormone output is low,
            # and green if it is high. Intermediate values
            # are blue.

            status 8,0 {
                size 1,1
                floatrange {
                    pos 0,0
                    title "HACT"
                    var hout
                    bands
                    <0.1 black
                    <0.7 blue
                    else green
                }
            }
        }
    }
}
```

Appendix C

The collation algorithm

This appendix describes the algorithm used by the **collateViconAndRoverCSV** Python script to merge the data from the rover capture file with the Vicon motion capture data.

It produces a file containing a line of comma separated values for each unique value of `ptime` in the rover capture data (i.e. each rover update), with the Vicon position and orientation data at that time added in, both absolute in the Vicon coordinate system, and relative to the start position of the rover.

- Read all the Vicon data into a list consisting of time, position and rotation. The list is ordered by time, and the time is in seconds since the start of the file (obtained from the frame number and known frame rate.)
- First pass: read the entire rover file, generating a list of keys, and a hash of keys and their values, set to the first values. Ignore any lines for which `go` is false.
- Add the Vicon data keys to the list of keys, and set their first values. Add both an (x, y, z) triplet for the relative position from the start of the run and a $(viconx, vicony, viconz)$ triplet for absolute positions relative to the Vicon's origin.
- Output a header consisting of a comma separated list of keys.
- Create an index for the "current Vicon datum" — set this to zero
- Second pass: for each line in the rover data file,
  - Set the current time to the `time` value of the line minus the time's first value (time is a UNIX timestamp; this will set it to start from zero at the start of the experiment).
  - Set all other values in the hash to the values received for each key in the line.
  - While the current time is greater than or equal to[1] that of the current Vicon datum, set the values for the Vicon elements in the hash to those of the current Vicon datum, calculating the relative positions (x, y, z) by subtracting the position of the first entry; and increment the current Vicon datum index.

    This will run through all the Vicon entries before the current time and set the position and rotation values in the collated data hash to each entry. There is a significant amount of unnecessary assignment for simplicity's sake, but the algorithm still runs quickly.

[1]Floating point values being what they are, "equal to" never happens.

  - If `go` in the hash is true (i.e. an experiment is running), and some Vicon data has been set, and `ptime` has changed since the last output, output all the values as a CSV line with the same ordering as the header. (The purpose of this last condition is to ensure that duplicate data are not output unnecessarily — `ptime` is guaranteed to change once per experiment tick.)

The result of this algorithm is to output a single line of data for each complete experiment tick, containing all the variables, with the correct rover position at that time.

Appendix D

Angort code listings

This appendix contains the listings of some key Angort scripts used in the experiments. The files can be found in `experiments`.

D.1 script.ang

This is the "default" script — it is loaded automatically by the rover scripting environment, before any files specified on the command line. It contains both remote control function definitions and useful functions common to all experiments.

```
# Angort script file for basic rover experiments.

# define a constant range, for the wheel numbers, over which we
# can iterate. These are wheels 1-6, and we say 1-7 because the
# actual interval is [1,7).

1 7 range const wheels

# Disable leg interference checking - normally if the legs
# get too close the rover will go into an exception state.
# In certain gaits (such as alternating), this is required behaviour.

0 setlegchecks

##################################################################
##
## Remote control code
##
## Note that the first part of the word's actual definition,
## which is in the form :"...." is the help text for the word.
## Where the word is simple, any comments are omitted in favour
## of this help text.
##
##################################################################


:setsteerall
    :"(pos --) set the steer position on all wheels"
    wheels each {dup i!steer}drop;

:a :"(pos --) shorthand for setsteerall, useful in remote control"
    setsteerall;

:setliftall
```

```
    :"(pos --) set the lift position on all wheels"
    wheels each {dup i!lift}drop;

:setdriveall
    :"(pos --) set the required drive speed on all wheels"
    wheels each {dup i!drive}drop;

:d :"(pos --) shorthand for setsteerall, useful in remote control"
    setdriveall;

:turn |t:|
    :"(angle --) turn the front wheels one way and the back wheels the opposite way"
    ?t dup 1!steer 2!steer
    ?t neg dup 5!steer 6!steer
    0 0 3!steer 4!steer
;

:t :"(angle --) shorthand for turn" turn;

###################################################################
##
## Configuration words, for when the rover configuration (such
## as control parameters for the motors) needs to be changed.
##
###################################################################

:setiall |gain,cap,decay:|
    :"set the integral gain, cap and decay on all drive motors"
    wheels each {
        ?gain i digain
        ?cap i dicap
        ?decay i didecay
    }
;

:setpall
    :"(pgain --) set the proportional gain on all drive motors"
    wheels each { dup i dpgain}drop;

:setdall
    :"(dgain --) set the differential gain on all drive motors"
    wheels each { dup i ddgain}drop;

:setpid |p,i,d:|
    :"(pgain igain dgain --) set PID gains, also sets Icap and Idecay to fixed values
        "
    ?p setpall
    ?i 200 1 setiall
    ?d setdall
;

:zerolift
    :"zero all lift motor gains"
    wheels each {0 i ligain};

:zerosteer
    :"zero all steer motor gains"
    wheels each {0 i sigain};



###################################################################
##
## UDP properties - values which are modified by packets sent
## from the monitoring system, but look like global variables
## to Angort scripts.
##
## Note that this occurs only when the interpreter is idle,
## or when handleudp is called.
##
##################################################################


# This is set when the user flips the EXP GO switch on the monitor,
# thus starting the experiment.

"go" addudpvar

# this is used only by the simulator - it's a factor which multiplies
# the simulated current generated by drive motors.

"scf" addudpvar

##################################################################
##
## These two words set up configurations (PID gains and other
## control system properties) used by all gaits.
##
##################################################################


#
# "Rolling calibration" is used in a walking gait when a wheel
# needs to be rolled forwards. We set the wheel's drive gain to
# very low, so the driving wheel is almost uncontrolled and unbraked,
# but a small amount of torque is still provided.
# We set the lift I-gain also quite low, but increase the decay
# constant and the cap - this increases the torque provided
# by the lift motor.
#
# This causes the lift motor to push the wheel to
# the desired position, while the drive motor rolls freely and
# provides a little help. The aim is for the leg to move to the
# required position while the drive wheel rolls smoothly along the
# surface.
#

:rollcalib |w:|
    :"(wheel--) calibrate a given wheel for rolling"
    0.007 ?w dpgain
    0 10 0 12 0.95 500 255 1 ?w setliftparams
;

# "Standard calibration" is used when a wheel is pushed or in
# normal rolling - it's the same as produced by the "calib" word,
# the default settings for the rover software.

:stdcalib |w:|
    :"(wheel--) calibrate a given wheel with standard gains"
    0.01 ?w dpgain
    0 5 0 50 0.9 500 255 1 ?w setliftparams
;


##################################################################
##
## Experiment helper functions used to manage the experiments
## themselves, primarily concerned with handling the "go" UDP
## property.
##
##################################################################

# This word centres all the lift and steer motors (waiting
# 2 seconds for this to be reflected in the actual positions),
# resets all odometry, and then enters a loop waiting for "go"
# to become true.
```

```
:expStart
    :"( -- ) experiment start code. Resets and waits for 'go'"

    "Ready to run, press EXP GO on the monitor".
    # zero all positions and wait
    wheels each {0 i!lift 0 i!steer} 2 delay
    # reset odometry
    wheels each {i resetodo}
    # loop waiting for "go" - note the update and handleudp.
    {
        0.1 delay
        update handleudp
        ?go ifleave
    }
    "Starting..." .
;

# this is called periodically inside the experiment.
# It updates the system, sends/receives UDP, and then
# return true if the experiment should leave. Typically
# an experiment will have a loop consisting of
# { ...do stuff... expUpdate ifleave}


:expUpdate
    :"(-- leave) experiment update code. Updates and returns true if containing loop
        should exit"
    0.1 delay update handleudp ?go not;
```

D.2 Experiment setup words, and an introduction to Angort

D.2.1 The `expStart` word

This word is used to reset the rover and wait for the "go" variable sent by the monitor to become non-zero[1]. In full, it reads as follows:

```
:expStart
    :"( -- ) experiment start code. Resets and waits for 'go'"

    "Ready to run, press EXP GO on the monitor".
    # zero all positions and wait
    wheels each {0 i!lift 0 i!steer} 2 delay
    # reset odometry
    wheels each {i resetodo}
    # loop waiting for "go" - note the update and handleudp.
    {
        0.1 delay
        update handleudp
        ?go ifleave
    }
    "Starting..." .
;
```

The first line begins the definition of the word, with no parameters or local variables. The second line defines a help text — if the user types

```
"expStart" help
```

[1]Booleans in Angort are integers where zero is false and non-zero is true, as in C.

they will be shown that string. The next line prints the string in quotes (the word "." converts the value on top of the stack into a string and prints it with a newline following).

The next line is an iterator loop, which is written less tersely as

```
wheels each {
    0 i!lift
    0 i!steer
}
2 delay
```

The `wheels` word will stack the constant of that name, which is a range value covering the integers from 1 to 6: the wheel numbers[2]. The compound word `each {` introduces an iterator loop, which iterates over the range or collection on top of the stack. It is terminated with }.

Within the loop, the sequence `0 i` will stack the number 0 and the current value of the iterator (i.e. the wheel number). The word `!lift` is a "property write" — `lift` is defined as a property, something which looks like a global variable to Angort, but has C++ get and set methods. The exclamation mark means "set", so the set method will run. This will first pop the wheel number, then the value, and set the required position of that lift motor to that position.

The next line does a similar thing with the steer motor, so the entire loop contents will set all the wheels' lift and steer motors to zero. The final line will just delay for two seconds, while the wheels recentre.

The line

```
wheels each {i resetodo}
```

will reset the odometry counter on each wheel, so we can measure the rover's odometry from zero.

The final loop:

```
{
    0.1 delay
    update handleudp
    ?go ifleave
}
```

is not an iterator loop — it is an standard infinite loop, delimited by { } without the `each`. Such loops run forever unless terminated by `leave` or the conditional `ifleave`, which jumps out of the loop at the point at which it executes. In this case, the loop pauses for 0.1s, updates the rover data, retrieves and sends UDP data, and then the words

```
?go ifleave
```

will exit the loop if `go` is true. The question mark indicates a get operation, and `go` is a UDP property — a value which is essentially a global variable but can also be set by `handleudp` receiving a packet from the monitor program, containing a new value. The final line simply prints a message to the console, and the trailing semicolon completes the word definition.

[2]Wheel numbers are in the range 1 to 6 (rather than 0 to 5) because that is how they are labelled in the documentation which was supplied with the rover.

D.2.2 The `expUpdate` word

This word is much simpler:

```
:expUpdate
    :"(-- leave) experiment update code. Updates and returns true if containing loop
        should exit"
    0.1 delay update handleudp ?go not;
```

This code is straightforward: delay for 0.1 seconds, then update, then send/receive UDP data. Finally, retrieve the `go` value and negate it, leaving the value on the stack for the calling code to examine (since any containing loop should exit when `go` is false).

This is indicated in the *stack picture* in the help text: "(– leave)" shows the state of the stack before and after the word's execution, separated by the double dash. In this case, there is nothing on the stack required by the word on entry, but a value called "leave" is added on exit.

D.2.3 sub.ang

This file contains the core of the subsumption architecture as used by the lurching and alternating gait.

```
#
# The core of the subsumption architecture.
#
# The design of the system is in the report proper,
# but it is based around "machines", each of which contains
# a number of states, and is implemented as a hash.
#
# States contain the state functions `entry, `exit and `update
# (in a hash of those symbols to anonymous functions).
#
# Machines also contain hashes of input and output values,
# the current state, and the time spent in the current state.
# They may also store any other value they like in their hash.
#
# Most of the functions is here are very short, and their
# help text describes them, so they are not commented.
#
# Note the frequent use of the "symbol shortcut get"
# syntactic sugar to get a value keyed by a symbol from a hash.
# Instead of
#
#    `mysymbol ?somehash get
#
# it's possible to type
#
#    ?somehash?`mysymbol

# This is used at the start of building a new architecture
# to clear the list of machines

:newmachines
    :"(--) clear the machine list for new machines"
    # "..better than those on Richese"
    [] !Machines
;

# Slightly ugly convention - the "This" global is the current
# state machine, which is initially nil (or "none" in Angort)

none !This

# this returns the value of This, asserting that it is not nil.

:this
    :"(-- current machine) get the current machine"
    ?This dup isnone not "This is none" assert
;

# A useful routine to show the states, inputs and outputs
# of a machine. Printing in Angort can be ugly, but "p" outputs
# the value on the stack without a trailing newline, and "." outputs
# it with one. Note heavy use of shortcut symbol get, and the nested
# hashes.

:dumpmachine |m:|
    :"(m --) dump a machine to console"
    "Machine " p ?m?`name p ", State: " p ?m?`state .
    "  Inputs:".
    ?m?`inputs each {
        "    " p i p ": " p i ?m?`inputs get .
    }
    "  Outputs:".
    ?m?`outputs each {
        "    " p i p ": " p i ?m?`outputs get .
    }
;

:curstate
    :"(-- s) get current state of machine"
    this?`state # current state key
    this?`states # state hash
    get;

# run one of the named functions within the current machine:
# exit, entry or update.

:runstatefn |f:|
    :"(f --) run function f in current machine"
    ?f curstate get
    call;

:statetime
    :"(-- t) get time since state start time"
    time this?`statestarttime - ;

:resetstatetime
    :"(--) reset the state start time of this machine"
    time this!`statestarttime ;

# all machines have a arbitrarily large set of timers, which
# is cleared on every state transition, as well as the main "statetime"
# timer - these are typically used by "ifsettledgo".

:resettimer |tid:|
    :"(tid -- ) reset a timer inside the machine, cleared on transition"
    time ?tid this?`timers set
;

:gettimer |tid:|
    :"(tid -- timer) get a timer inside the machine, cleared on transition"
    ?tid this?`timers in not if
        ?tid resettimer
    then
    time ?tid this?`timers get -
;


:cleartimers
    :"(--) clear all timers in the machine, called on transition"
    [%] this!`timers
;

:nnn |s:|
    :"(s -- s) deal with none, replacing with string 'none'"
    ?s isnone if "none" else ?s then
;

# for sending state data over UDP (as was done in debugging)
# we need to associate symbols (i.e. statenames) with numbers.
# This is done using the SymN hash.

[%] !SymN       # symbol ID number keyed on hash
0 !SymnCt       # symbol ID number counter
:symn |s:|
    :"(s -- n) replace a symbol with a number, unique to that symbol"
    ?s ?SymN in not if
        # if there isn't a number for this symbol, make one
        # from the counter and increment said counter.
        ?SymnCt ?s ?SymN set
        ?SymnCt 1+ !SymnCt
    then
    # return the symbol's number
    ?s ?SymN get
;

:gostate |s:|
    :"(s m --) Tell machine m to transition to state s"

    # uncomment this line to see state changes on console
    #    "transitioning " p this?`name p " to state " p ?s.

    # write the state change to UDP (using symn above)
    this?`name "=" + ?s symn + udpwrite

    # check the state exists
    ?s this?`states in
    "state \"" ?s nnn "\" not in machine \"" this?`name nnn "\""+ + + + assert

    # run the exit function of the current state (if there is one)
    this?`state isnone not if
        `exit runstatefn
    then

    # set the new state
    ?s this!`state
    # set the "statestarttime" to now
    resetstatetime
    # clear all auxiliary timers
    cleartimers
    # run the entry function of the new state if there is one
    this?`state isnone not if
        `entry runstatefn
    then
;

:initmachine |m,s:|
    :"(machine initstate -- machine) initialise a machine, called at end of
        definition"
    ?m?`name isnone if
        ?m each {i.}
        0 "machine has no name" assert
    then

    ?m!This           # set This
    ?s gostate        # goto the initial state s
    ?m ?Machines push # add machine to the global list
    ?m                # return the machine
;


:ifsettledgo |c,s,tid:|
    :"(c s timerid --) if cond is true and has been true for a bit, go to new state"
    # this will reset the timer if the condition is false, but if the condition
    # is true will check that the timer was reset sufficiently long ago, and if
    # so, will transition to a new state. If the timer is not present, it is created.

    ?c if
        ?tid gettimer SETTLETIME > if
            ?s gostate
        then
    else
        ?tid resettimer
    then
;


:updatemachine |m:|
    :"(m -- ) set the given machine to this, and then run its update"
    ?m!This
    `update runstatefn
;

:updateallmachines
    :"(--) update all machines in the global list"
    ?Machines each {i updatemachine}
;

# words to read and write data inside a machine's update function

:input |i:|
    :"(i --) read input named i in the current machine"
    ?i this?`inputs in
    "input \"" ?i nnn "\" not in machine \"" this?`name nnn "\""+ + + + assert
    ?i this?`inputs get
;

:output |v,o:|
    :"(v o --) set output named o to v in the current machine"
    ?o this?`outputs in
    "output \"" ?o nnn "\" not in machine \"" this?`name nnn "\""+ + + + assert
    ?v ?o this?`outputs set
;

# words to route data between machines

:readout |o,m:|
    :"(o m -- v) read a value from an output of a machine"
    ?o ?m?`outputs in
    "output \"" ?o nnn "\" not in machine \"" this?`name nnn "\""+ + + + assert
    ?o ?m?`outputs get
;

:writein |v,i,m:|
    :"(v i m --) write a value to the input of a machine"
    ?i ?m?`inputs in
    "input \"" ?i nnn "\" not in machine \"" this?`name nnn "\""+ + + + assert
    ?v ?i ?m?`inputs set
;

# routing shortcuts
:directnamed |o,i,n:|
    :"(outmach inmach names --) connect named outputs to inputs with same name on
        another machine"
    ?n each {
        i ?o readout i ?i writein
    }
;

:directall |o,i:|
    :"(outmach inmach --) connect all inputs of a machine to outputs in another, if
        they exist"
    ?i?`inputs each {
        i ?o?`outputs in if
            i ?o readout i ?i writein
        then
    }
;






# handle named calibration presets
[%
 `std (stdcalib),
 `roll (rollcalib)
] const CALIBRATIONS

:recalibrate |c,w:|
    ?c isnone not if
        ?c CALIBRATIONS in if
            ?w ?c CALIBRATIONS get call
        then
    then
;

:subsume |a,b:|
    :"(a b -- out) if b is not none, b; else a"
    ?b isnone if ?a else ?b then
;

:inhibit |a,b:|
    :"(a b -- out) if b is true, none; else a"
    ?b if none else ?a then
;
```

D.2.4 An example subsumption machine (or behaviour)

This example shows a very simple machine, one of the elements which make up a subsumption architecture.

A hash is created and put on the stack in Angort with the following syntax:

```
[% key value, key value, ..., key value]
```

and an anonymous function is simply a fragment of Angort code in brackets — when compiled, this is replaced with a word to stack a reference to the code in the brackets, which can then be called with the `call` word.

Once created, `initmachine` must be called on the hash and the symbol for the initial state, returning the new initialised machine.

Here is an example machine, or rather code to generate and return an example machine:

```
:mkmymachine
    [%  # begin defining the hash

        `name "mymachine",

        # the machine has two inputs, both of which are initially nil
        `inputs [% `input1 none, `input2 none]

        # it has no outputs - perhaps it controls the rover directly
        `outputs [%]

        # it has one variable set to 4.0 initially
        `myvariable 4.0,

        # this is the hash containing the states

        `states [%

            # the initial and only state, which switches the drive motors on
            # if either input is not nil.

            `init [%
                `entry (), `exit (),  # the entry and exit functions do nothing
                `update (             # runs every tick

                    # Note that in Angort, "if(COND){CODE}else{CODE}"
                    # becomes "COND if CODE else CODE then"

                    `input1 input isnone
                    `input2 input isnone and if
                        # both inputs are nil, set the drives to zero
                        0 setdriveall
                    else
                        # otherwise turn the drives on
                        1000 setdriveall
                    then
                )
            ] # end of init state hash
        ] # end of states hash
    ] # end of machine hash

    # pass the hash we just made and the initial state's symbol
    # into initmachine

    `init initmachine
;
```

D.2.5 The lurch experiment

This is the Angort code for the lurch experiment, which is started with the `tst` word.

```
#
# Lurch gait experiment
#

# include the subsumption architecture, and the standard input/output
# machines

include "sub.ang"
include "machines.ang"


-20 const FRONTANGLE   # forward wheel position in degrees
20 const BACKANGLE     # backward wheel position
0.2 const SETTLETIME   # time for debouncing position checks
-2200 const DRIVESPEED # how fast the wheel is commanded to roll

# how close a lift angle must be to the target to be considered
# to have achieved it

8 const LIFTEPSILON

:nonefilter |a,deflt:|
    :"(a default --) if a is none, replace with a default"
    ?a isnone if ?deflt else ?a then
;

:withinliftepsilon |a,b:|
    :"(a b--bool) are quantities a and b within LIFTEPSILON of each other,
        disregarding NONE?"
    ?a isnone ?b isnone or if
        0
    else
        ?a ?b - abs LIFTEPSILON <
    then
;

# this machine outputs true if the input indicates that the wheel is fully back
:mkisbackward [%
   `name `isback,
    `inputs [% `lactual none ],
    `outputs [% `out 0 ],
    `states [%
        `init [%
               `entry (), `exit (),
               `update (
                   `lactual input BACKANGLE withinliftepsilon
                   `out output
               )
               ]
        ]
    ]
    `init initmachine
;

# this machine outputs true if the input indicates that the wheel is fully forwards
:mkisforward [%
   `name `isfwd,
    `inputs [% `lactual none ],
    `outputs [% `out 0 ],
    `states [%
        `init [%
               `entry (), `exit (),
               `update (
                   `lactual input FRONTANGLE withinliftepsilon
```

```
                    `out output
                )
                ]
        ]
    ]
    `init initmachine
;

# this machine has three states:
# init: no output. On receipt of "go", go to roll
# roll; outputs to make the wheel roll forwards. On "stop", go to finished
# finished: turn off drive motor, output "trigger", and after a short interval return
     to init.

:mkrollfwd [%
   `name `roller,
    `inputs [% `go 0, `stop 0 ],
    `outputs [% `drive none,`lift none,`calib none, `trigger none ] ,
    `states [%
        `init [%
                `entry (), `exit (),
                `update (
                    `go input
                    `roll 0 ifsettledgo
                    # default outputs
                    none `trigger output
                    none `drive output
                    none `lift output
                    none `calib output
                )
                ],
        `roll [%
                `entry (), `exit (),
                `update (
                    `stop input
                    `finished 0 ifsettledgo
                    # rolling outputs
                    none `trigger output
                    DRIVESPEED `drive output
                    FRONTANGLE `lift output
                    `roll `calib output
                )
                ],
        `finished [%
                `entry (), `exit (),
                `update (
                    1 `trigger output
                    0 `drive output
                    statetime 0.2 > if
                        `init gostate
                    then
                )
                ]

        ]
    ]
    `init initmachine
;

#
# On "go", output lurch (zero drive, all lift motors to back position).
# On "stop", go back to outputting nothing.
#

:mklurchback [%
   `name `lurcher,
    `inputs [% `go 0, `stop 0 ],
    `outputs [% `drive none,`lift none,`calib none ] ,
    `states [%
        `init [%
               `entry (), `exit (),
               `update (
                   `go input
                   `lurch 0 ifsettledgo
                   # default outputs
                   none `drive output
                   none `lift output
                   none `calib output
               )
               ],
        `lurch [%
               `entry (), `exit (),
               `update (
                   `stop input
                   `init 0 ifsettledgo
                   # rolling outputs
                   0 `drive output
                   BACKANGLE `lift output
                   `std `calib output
               )
               ]
        ]
    ]
    `init initmachine
;

#
# Build the architecture for the lurching system
#
:build
    newmachines  # clear the machine list

    # create the per-wheel machines, adding them to a list of
    # hashes, keyed by symbols for the type of machine they are

    [] wheels each {
        [%
         `sensor     i mkwheelsensor,
         `isfwd        mkisforward,
         # mkbackward is redundant in the case of wheels 3,4
         # but it's retained here for simplicity
         `isback       mkisbackward,
         `roller       mkrollfwd,
         `output     i mkwheeloutput
         ], # "," will add to the list
    }

    # store the list
    !WheelList

    # and the two lurchers (we do each side separately)
    mklurchback !LurchEven
    mklurchback !LurchOdd
;

#
# Helper functions for accessing machines inside the wheel list
#

:getwh |w:|
    :"(w -- )get a wheel's machine hash in the canonical on-board numbering scheme"
    ?w 1- ?WheelList get
;

:rd |w,o,m:|
    :"(w o m -- v) get the output of a given machine in the given wheel"
    ?o ?m ?w getwh get readout
;

:wr |v,w,i,m:|
    :"(v w i m --) write the input of a given machine in the given wheel"
    ?v
    ?i ?m ?w getwh get writein
;


#
# Run a single iteration of the architecture
#
:run
    updateallmachines
    # here we do the routing and subsumption/inhibition

    # first, read the inputs into the isfwd and isback machines
    ?WheelList each {
        i?`sensor dup i?`isfwd directall i?`isback directall
    }

    # The front wheels start rolling forwards when they are fully backwards.
    # The other wheels wait for "trigger" from the wheel in front, which
    # happens (briefly) when they complete their rolls.

    1 `out `isback rd           1 `go `roller wr
    1 `trigger `roller rd       3 `go `roller wr
    3 `trigger `roller rd       5 `go `roller wr

    2 `out `isback rd           2 `go `roller wr
    2 `trigger `roller rd       4 `go `roller wr
    4 `trigger `roller rd       6 `go `roller wr

    # and any rolling is stopped when the wheel is fully forward
    ?WheelList each {
        `out i?`isfwd readout    `stop i?`roller writein
    }

    # when the back wheel on each side is in the forward position, start a lurch on
        that side
    5 `out `isfwd rd    `go ?LurchOdd writein
    6 `out `isfwd rd    `go ?LurchEven writein
    # we stop all the lurching when the back wheels are in place
    5 `out `isback rd   `stop ?LurchOdd writein
    6 `out `isback rd   `stop ?LurchEven writein


    # connect to the lurchers to the output via a subsumption action; actually three
        of them.
    # More than that, because it's duplicated for each side.

    [1,3,5] each {
        i `drive `roller rd    `drive ?LurchOdd readout subsume   i `drive `output wr
        i `lift  `roller rd    `lift  ?LurchOdd readout subsume   i `lift  `output wr
        i `calib `roller rd    `calib ?LurchOdd readout subsume   i `calib `output wr
    }
    [2,4,6] each {
        i `drive `roller rd    `drive ?LurchEven readout subsume   i `drive `output
            wr
        i `lift  `roller rd    `lift  ?LurchEven readout subsume   i `lift  `output
            wr
        i `calib `roller rd    `calib ?LurchEven readout subsume   i `calib `output
            wr
    }
;
```

```
:tst
    :"(--) run the experiment")
    build    # build machines
    expStart # start experiment
    run run  # a couple of dummy runs to clear the decks
    BACKANGLE setliftall # set all wheels to back
    {
        run  #run the machines and the plumbing
        expUpdate ifleave
    }
    "Stopping.".

    # recalibrate normally and reset everything
    calib
    0 setdriveall
    0 setliftall
;
```

D.2.6 The alternating experiment

This is the Angort code for the alternating gait experiment, which is started with the `tst` word. Note the use of the `reduce` function and lists inside the code for triggering lurchers and rollers:

```
1                   # - seed the reduce with 1 (true)
i gettri            # - get a list of all the wheels in this triangle
(                   # - start anonymous function compilation: input is a
                    #   boolean and the wheel number
    1-              # - subtract 1: wheel numbers start from 1, angort lists from 0
    [2,1,4,3,6,5]   # - stack a list of the opposite wheels for each wheel 1-6
    get             # - get the opposite wheel number
    `out `isfwd rd  # - get whether the opposite wheel is forward
    and             # - AND this with whatever the boolean passed in was
)                   # - end anonymous function and stack it
reduce              # - starting with 1, call the anonymous function repeatedly
                    #   with each wheel in the triangle, storing the result in
                    #   an accumulator. As "reduce" in many languages, or "foldl"
                    #   in Haskell.
```

The result is whether all the wheels in the opposite triangle are forwards.

```
#
# Alternating gait experiment
#

# include the subsumption architecture, and the standard input/output
# machines

include "sub.ang"
include "machines.ang"


# set the constants
-20 const FRONTANGLE   # forward wheel position in degrees
20 const BACKANGLE     # backward wheel position
0.2 const SETTLETIME   # time for debouncing position checks
-3200 const DRIVESPEED # how fast the wheel is commanded to roll - FAST

# how close a lift angle must be to the target to be considered
# to have achieved it

8 const LIFTEPSILON

:nonefilter |a,deflt:|
    :"(a default --) if a is none, replace with a default"
    ?a isnone if ?deflt else ?a then
;

:withinliftepsilon |a,b:|
    :"(a b--bool) are quantities a and b within LIFTEPSILON of each other,
        disregarding NONE?"
    ?a isnone ?b isnone or if
        0
    else
        ?a ?b - abs LIFTEPSILON <
    then
;

# this machine outputs true if the input indicates that the wheel is fully back
:mkisbackward |w:| [%
   `name "isback" ?w +,
    `inputs [% `lactual none ],
    `outputs [% `out 0 ],
    `states [%
        `init [%
                `entry (), `exit (),
               `update (
                   `lactual input BACKANGLE withinliftepsilon
                   `out output
               )
               ]
        ]
    ]
    `init initmachine
;

# this machine outputs true if the input indicates that the wheel is fully forwards
:mkisforward |w:| [%
   `name "isfwd" ?w +,
    `inputs [% `lactual none ],
    `outputs [% `out 0 ],
    `states [%
        `init [%
               `entry (), `exit (),
               `update (
                   `lactual input FRONTANGLE withinliftepsilon
                   `out output
               )
               ]
        ]
    ]
    `init initmachine
;


# this machine has three states:
# init: no output. On receipt of "go", go to roll
# roll; outputs to make the wheel roll forwards. On "stop", go to finished
# finished: turn off drive motor, and after a short interval return to init.

:mkrollfwd |w:| [%
   `name "rollfwd" ?w +,
    `inputs [% `go 0, `stop 0 ],
    `outputs [% `drive none,`lift none,`calib none ] ,
    `states [%
        `init [%
               `entry (), `exit (),
               `update (
                   `go input
                   `roll 0 ifsettledgo
                   # default outputs
                   none `drive output
                   none `lift output
                   none `calib output
               )
               ],
        `roll [%
               `entry (), `exit (),
               `update (
                   `stop input
                   `finished 0 ifsettledgo
                   # rolling outputs
                   DRIVESPEED `drive output
                   FRONTANGLE `lift output
                   `roll `calib output
               )
               ],
        `finished [%
               `entry (), `exit (),
               `update (
                   0 `drive output
                   statetime 0.2 > if
                       `init gostate
                   then
```

```
                )
                ]

        ]
    ]
    `init initmachine
;


#
# On "go", output lurch (zero drive, all lift motors to back position).
# On "stop", go back to outputting nothing.
#

:mklurchback |w:| [%
   `name "lurch" ?w +,
    `inputs [% `go 0, `stop 0 ],
    `outputs [% `drive none,`lift none,`calib none ] ,
    `states [%
        `init [%
                `entry (), `exit (),
                `update (
                    `go input
                    `lurch 0 ifsettledgo
                    # default outputs
                    none `drive output
                    none `lift output
                    none `calib output
                )
                ],
        `lurch [%
                `entry (), `exit (),
                `update (
                    `stop input
                    `init 0 ifsettledgo
                    # rolling outputs
                    0 `drive output
                    BACKANGLE `lift output
                    `std `calib output
                )
                ]
        ]
    ]
    `init initmachine
;

:build
    newmachines
    # per-wheel machines
    [] wheels each {
        [%
         `sensor     i mkwheelsensor,
         `isback     i mkisbackward,
         `roller     i mkrollfwd,
         `isfwd      i mkisforward,
         `lurcher    i mklurchback,
         `output     i mkwheeloutput
         ],
    }

    !WheelList
;

:getwh |w:|
    :"(w -- )get a wheel in the canonical on-board numbering scheme"
    ?w 1- ?WheelList get
;
```

```
:rd |w,o,m:|
    :"(w o m -- v) get the output of a given machine in the given wheel"
    ?o ?m ?w getwh get readout
;

:wr |v,w,i,m:|
    :"(v w i m --) write the input of a given machine in the given wheel"
    ?v
    ?i ?m ?w getwh get writein
;

# this constant defines the wheels for one triangle
[1,4,5] const Triangle1
# and this for the other triangle
[2,3,6] const Triangle2
# this defines which wheel is in which triangle
:gettri
    1- [Triangle1,Triangle2,Triangle2,
        Triangle1,Triangle1,Triangle2] get
;

:run
    updateallmachines
    # here we do the routing and subsumption/inhibition

    # first, read the inputs into the various machines
    ?WheelList each {
        i?`sensor i?`isfwd directall
        i?`sensor i?`isback directall
    }


    wheels each {
        # each wheel starts rolling forward when all the wheels in the
        # opposite triangle are fully forward and all the wheels in this
        # triangle are fully back
        1   i gettri (
            1- [2,1,4,3,6,5] get # get opposite wheel
            `out `isfwd rd and) reduce
        1   i gettri (`out `isback rd and) reduce and
            i `go `roller wr

        # each wheel's lurch is started when the opposing wheels are all
        # fully back, and all the wheels in this triangle are fully forwards

        1   i gettri (
            1- [2,1,4,3,6,5] get # get opposite wheel
            `out `isback rd and) reduce
        1   i gettri (`out `isfwd rd and) reduce and
            i `go `lurcher wr

        # any rolling is stopped when the wheel is fully forward,
        # and any lurching is stopped when the wheel is fully back.
        i `out `isfwd rd        i `stop `roller wr
        i `out `isback rd       i `stop `lurcher wr

    }


    # connect the output; lurch subsumes roll.

    wheels each {
        i `drive `roller rd     i `drive `lurcher rd subsume    i `drive `output wr
        i `lift `roller rd      i `lift `lurcher rd subsume     i `lift `output wr
        i `calib `roller rd     i `calib `lurcher rd subsume    i `calib `output wr
    }

;
```

```
:tst
    :"(--) run the experiment")
    expStart
    build
    run run
    ?Triangle1 each {BACKANGLE i!lift}
    ?Triangle2 each {FRONTANGLE i!lift}
    {
        run
        expUpdate ifleave
    }
    "Stopping.".
    calib
    0 setdriveall
    0 setliftall
;
```

D.2.7 Hormone framework code

This is the Angort code for the AES — it consists of code to create a hormone object (a hash table) given the parameters, and code to update the hormone. Each hormone encapsulates the architecture in Figure 2.17. Rather than using accessors, code which uses this framework reads the values directly from the hash. For example, to read the output of hormone `h1`, we use `?h1¿out` (using the "shortcut symbol get" syntactic sugar).

```
#
# System for managing hormones in an artificial endocrine system
#


# a hormone has the following parameters:
# * input sigmoid centre and width - used to process the input value
#   to generate a value between 0 and 1
# * release rate - the input value, after sigmoid, is multiplied
#   by this and added to the current level (initially zero)
# * decay rate - the current level is multiplied by this
# * output sigmoid centre and width - the current level is processed
#   through this sigmoid to produce the final output value. Set the width
#   very small to produce a boolean output, and check for <0.5.
# The maximum level of the hormone is assumed to be 1, and this
# is enforced by an exponential function.
#
# The parameters are stored in a hash generated by the following
# function, which also stores the current level.

:mkhormone |insigcent,insigwidth,release,decay,outsigcent,outsigwidth:|
    :"(insigcent insigwidth release decay outsigcent outsigwidth -- h) make a hormone
       "
    # create the hash and store data in it
    [%
     `level      0,             # the current hormone level
     `output     0,             # the current output value
     `inSigWidth ?insigwidth,   # input perceptron sigmoid parameters
     `inSigCent  ?insigcent,
     `release    ?release,      # release rate
     `decay      ?decay,        # decay rate
     `outSigWidth ?outsigwidth, # output perceptron sigmoid parameters
     `outSigCent  ?outsigcent
     ]
;

#
# Helper maths functions used in hormone processing
#


# sigmoid function, takes the input and the width and centre parameters
# of the sigmoid. Produces a value from 0 to 1.

:sigmoid |in,w,c:|
    :"(in w c -- out) sigmoid function with centre and width factor"
    # first subtract the centre from the input
    ?in ?c -
    # multiply the width by 0.1 (an arbitrary factor)
    ?w 0.1 *
    # and divide the input by that
    / !in

    # this does 1-1/(e^x+1) - the actual sigmoid function
    1 1 ?in exp  1 +  / -
;
```

```
#
# the actual hormone update function
#


:updateHormone |in,h:|
    :"(in h -- output) update the hormone given an input, and return the output"
    # process the input through the input sigmoid
    ?in ?h?`inSigWidth ?h?`inSigCent sigmoid
    # multiply this by the release value
    ?h?`release *
    # multiply this by the capping factor, so as there's more hormone
    # the concentration goes up more slowly (from the Endover code,
    # adapted)
    # The idea is that you never reach max concentration, and the more
    # there is, the less diffuses out of the gland.
    0.95 ?h?`level - *
    # add this to the current level
    ?h?`level +
    # decay the new level and store back (but keep on stack)
    ?h?`decay *
    dup ?h!`level
    # process through the output sigmoid and store in output (duplicating
    # so we also get a return value
    ?h?`outSigWidth ?h?`outSigCent sigmoid
    dup ?h!`output
;
```

D.2.8 Endocrine gait switcher

This code uses the hormone framework in the previous section to implement a switching gait, between roll-500 and lurching.

```
-20 const FRONTANGLE
20 const BACKANGLE
0.2 const SETTLETIME
-2200 const DRIVESPEED # how fast the wheel is commanded to roll
8 const LIFTEPSILON

include "sub.ang"
include "machines.ang"
include "hormone.ang"


:nonefilter |a,deflt:|
    :"(a default --) if a is none, replace with a default"
    ?a isnone if ?deflt else ?a then
;

:withinliftepsilon |a,b:|
    :"(a b--bool) are quantities a and b within LIFTEPSILON of each other,
        disregarding NONE?"
    ?a isnone ?b isnone or if
        0
    else
        ?a ?b - abs LIFTEPSILON <
    then
;

# create a hormone

0.10 0.01       # input sigmoid centre and width (current)
0.1             # release rate high!
0.9             # decay rate high!
0.4 0.001       # output sigmoid (boolean)
mkhormone !Hormone

# actual input hormone
200 800         # input sigmoid centre and width (current)
0.001           # release rate
0.994           # decay rate
0.5 1           # output sigmoid (boolean) (UNUSED)
mkhormone !Hormone2

# this machine outputs true if the input indicates that the wheel is fully back
:mkisbackward [%
   `name `isback,
    `inputs [% `lactual none ],
    `outputs [% `out 0 ],
    `states [%
        `init [%
               `entry (), `exit (),
               `update (
                   `lactual input BACKANGLE withinliftepsilon
                   `out output
               )
               ]
        ]
    ]
    `init initmachine
;

# this machine outputs true if the input indicates that the wheel is fully forwards
:mkisforward [%
   `name `isfwd,
    `inputs [% `lactual none ],
    `outputs [% `out 0 ],
    `states [%
        `init [%
               `entry (), `exit (),
               `update (
                   `lactual input FRONTANGLE withinliftepsilon
                   `out output
               )
               ]
        ]
    ]
    `init initmachine
;

# this machine outputs commands to lurch the wheels back if they are all centered.
# To fit the uncentralised model of the other machines, this should work by examining
# the adjacent wheels somehow, and should use the input machines rather than direct
# reading - but time is short.

:mkkicker [%
   `name `isfwd,
    `inputs [%],
    `outputs [%  `drive none, `lift none, `calib none ],
    `states [%
        `init [%
               `entry (), `exit (),
               `update (
                   1 wheels each {i lactual 0 withinliftepsilon and} if
                       BACKANGLE
                   else
                       none
                   then
                   `lift output
               )
               ]
        ]
    ]
    `init initmachine
;


# this machine has three states:
# init: no output. On receipt of "go", go to roll
# roll; outputs to make the wheel roll forwards. On "stop", go to finished
# finished: turn off drive motor, output "trigger", and after a short interval return
     to init.

:mkrollfwd [%
   `name `roller,
    `inputs [% `go 0, `stop 0 ],
    `outputs [% `drive none,`lift none,`calib none, `trigger none ] ,
    `states [%
        `init [%
               `entry (), `exit (),
               `update (
                   `go input
                   `roll 0 ifsettledgo
                   # default outputs
                   none `trigger output
                   none `drive output
                   none `lift output
                   none `calib output
               )
               ],
        `roll [%
               `entry (), `exit (),
               `update (
                    `stop input
                    `finished 0 ifsettledgo
                    # rolling outputs
                    none `trigger output
                    DRIVESPEED `drive output
                    FRONTANGLE `lift output
                    `roll `calib output
                )
                ],
        `finished [%
                `entry (), `exit (),
                `update (
                    1 `trigger output
                    0 `drive output
                    statetime 0.2 > if
                        `init gostate
                    then
                )
                ]

        ]
    ]
    `init initmachine
;

#
# On "go", output lurch (zero drive, all lift motors to back position).
# On "stop", go back to outputting nothing.
#

:mklurchback [%
   `name `lurcher,
    `inputs [% `go 0, `stop 0 ],
    `outputs [% `drive none,`lift none,`calib none ] ,
    `states [%
        `init [%
                `entry (), `exit (),
                `update (
                    `go input
                    `lurch 0 ifsettledgo
                    # default outputs
                    none `drive output
                    none `lift output
                    none `calib output
                )
                ],
        `lurch [%
                `entry (), `exit (),
                `update (
                    `stop input
                    `init 0 ifsettledgo
                    # rolling outputs
                    0 `drive output
                    BACKANGLE `lift output
                    `std `calib output
                )
                ]
        ]
    ]
    `init initmachine
;

# this machine is a basic roller - it will set the wheels upright
# and move at 500. However, it will only produce this output
# when the hormone is low, otherwise it outputs none. This way,
# this behaviour will subsume the lurching behaviour until things
# go wrong when we will be free to lurch.
```

```
:mkhormoneroller [%
    `name `hormoneroller,
    `inputs [%],
    `outputs [% `drive none,`lift none,`calib none ] ,
    `states [%
        `nothing [%
               `entry (), `exit (),
               `update (
                   # the hormone output neuron is LOW and all the wheels close to
                   # FRONTANGLE, go into the next state
                   ?Hormone?`output 0.5 <
                   1 wheels each {i lactual FRONTANGLE withinliftepsilon and} and if
                       `rolling gostate
                   else
                       none `drive output
                       none `lift output
                       none `calib output
                   then
               )
               ],
        `rolling [%
               `entry (), `exit (),
               `update (
                   # if the hormone is HIGH, we go into the state which does nothing
                   ?Hormone?`output 0.5 >
                      `nothing 0 ifsettledgo

                   # only start driving once all the lift wheels have stopped moving

                   1 wheels each {i lactual 0 withinliftepsilon and} if -500 else 0
                       then
                      `drive output
                   0 `lift output
                   `std `calib output
               )
               ]
        ]
    ]
    `rolling initmachine
;

:build
    newmachines
    # per-wheel machines
    [] wheels each {
        [%
         `sensor     i mkwheelsensor,
         `isfwd        mkisforward,
         # mkbackward is redundant in the case of wheels 3,4
         # but it's retained here for simplicity
         `isback       mkisbackward,
         `roller       mkrollfwd,
         `output     i mkwheeloutput
         ],
    }

    !WheelList
    # create the two lurchers (we do each side separately)
    mklurchback !LurchEven
    mklurchback !LurchOdd
    # and the plain roller we use to override when the hormone
    # is high
    mkhormoneroller !HormoneRoller
    # and this is the kicker, which kicks the wheels back when they're all centred
    mkkicker !Kicker
;

:getwh |w:|
    :"(w -- )get a wheel in the canonical on-board numbering scheme"
    ?w 1- ?WheelList get
;

:rd |w,o,m:|
    :"(w o m -- v) get the output of a given machine in the given wheel"
    ?o ?m ?w getwh get readout
;

:wr |v,w,i,m:|
    :"(v w i m --) write the input of a given machine in the given wheel"
    ?v
    ?i ?m ?w getwh get writein
;


:run |:dowheeldatum|
    updateallmachines

    # update the hormone
    # sum the currents
    0 wheels each {i dcurrent + i lcurrent + }

    # Hormone cascade fed by current
    ?Hormone2 updateHormone drop
    ?Hormone2?`level # use the level of H2 rather than output
    ?Hormone updateHormone drop

    "hlevel=" ?Hormone?`level + udpwrite
    "hout=" ?Hormone?`output + udpwrite
    "hlevel2=" ?Hormone2?`level + udpwrite
    "hout2=" ?Hormone2?`output + udpwrite

    # fetch the scf UDP property, which is used in debugging,
    # and set the simulated current factor to some factor of it
    ?scf 2* !simcurrentfactor


    # here we do the routing and subsumption/inhibition

    # first, read the inputs into the isfwd and isback machines
    ?WheelList each {
        i?`sensor dup i?`isfwd directall i?`isback directall
    }

    # The front wheels start rolling forwards when they are fully backwards.
    # The other wheels wait for "trigger" from the wheel in front, which
    # happens (briefly) when they complete their rolls.

    1 `out `isback rd           1 `go `roller wr
    1 `trigger `roller rd       3 `go `roller wr
    3 `trigger `roller rd       5 `go `roller wr

    2 `out `isback rd           2 `go `roller wr
    2 `trigger `roller rd       4 `go `roller wr
    4 `trigger `roller rd       6 `go `roller wr

    # and any rolling is stopped when the wheel is fully forward
    ?WheelList each {
        `out i?`isfwd readout    `stop i?`roller writein
    }

    # when the back wheel on each side is in the forward position, start a lurch on
        that side
    5 `out `isfwd rd    `go ?LurchOdd writein
    6 `out `isfwd rd    `go ?LurchEven writein
    # we stop all the lurching when the back wheels are in place
    5 `out `isback rd   `stop ?LurchOdd writein
    6 `out `isback rd   `stop ?LurchEven writein


    # connect to the lurchers to the output via a subsumption action; actually three
        of them.
    # More than that, because it's duplicated for each wheel.

    # this is done for each type of data (represented by a symbol) in each wheel:
    # - get the value of that data in the roller for the wheel
    # - get the value of that data in the lurcher, and subsume the roller data
    #   with it (so if the lurcher has data it will override)
    # - and then subsume that with the output of the hormoneroller.
    # - store the data into the output for that wheel
    # It's written as a local function for convenience.

    ( |w,symbol,lurcher:|
        ?w ?symbol `roller rd
        ?symbol ?LurchOdd readout subsume
        # now we add another layer - when all the legs are down,
        # we need subsume the lurcher to kick them backwards.
        ?symbol ?Kicker readout subsume
        # and then we subsume *that* with the hormone-low roll behaviour
        ?symbol ?HormoneRoller readout subsume
        ?w ?symbol `output wr

    ) !dowheeldatum

    [1,3,5] each {
        [`drive, `lift, `calib] each {
            j i ?LurchOdd ?dowheeldatum@
        }
    }
    [2,4,6] each {
        [`drive, `lift, `calib] each {
            j i ?LurchEven ?dowheeldatum@
        }
    }
;



:tst
    expStart
    build
    {
        run
        expUpdate ifleave
    }
    "Stopping.".
    calib
    0 setdriveall
    0 setliftall
;
```

Appendix E

Video evidence

There are a number of video files for the project, each of which is available online. These are listed in Table E.1.

1	Demonstration of the lurching gait	`https://www.youtube.com/watch?v=ufw0CnLncjY`
2	Demonstration of the alternating gait	`https://www.youtube.com/watch?v=BjRkYZ7-_gE`
3	Oscillatory behaviour in gait switching (x3 timelapse)	`https://www.youtube.com/watch?v=KS-EX95luHs`
4	Successful gait switching (x3 timelapse)	`https://www.youtube.com/watch?v=7vspI6JLdMA`
5	Bow-wave formation at roll-1000 on a loose surface	`https://www.youtube.com/watch?v=oCkwf8nakkc`
6	Roll-500 on a loose surface showing slip and stall	`https://www.youtube.com/watch?v=dajwkfjjbF8`
7	Alternating gait on the loose, showing severe slip and sinkage	`http://youtu.be/V4u1E2g6orE`

Table E.1: List of videos of experiments

Appendix F

Third party code and libraries

Very little third-party code was used in the experiments, with the exception of a few R language libraries for data analysis.

F.1 R libraries

The R libraries used were:

- `pspline` for generating polynomial smoothing splines to approximate data [31];
- `calibrate` for adding text to points in graphs (not used in final code) [18];
- `hexbin` for generating hexbinned scatter plots (not used in final code) [6];
- `xtable` for generating LaTeXtables from data [9];
- `ggplot2` for generating early versions of plots [38];
- `corrgram` for producing correlograms, useful images for visualising correlation matrices [16].

F.2 Other libraries

The monitor technically makes use of the MarbleWidget mapping library, but firstly this code was not developed as part of the project, and this feature of the monitor was not used.

All code developed for the project, even including Angort itself and the rover scripting system, uses only standard C and C++ libraries, with the exception of the Libkoki evaluation (see Section J).

F.2.1 LaTeX packages

Considerable use was made of the `listings` package with custom language definitions for Angort and monitor configuration files. In addition, the package itself was enhanced by the author to add the capability to define strings by a prefix. This was necessary for the correct formatting of Angort symbols.

Other packages used included `fancyvrb` for better verbatim sections, and `subcaption` for subfigure support. Finally, the `mmp.sty` file was modified to allow more room for subsection numbers in the table of contents.

Appendix G

Experimental checklist

This appendix shows the experimental checklist used to run each experiment.

G.1 Prep

1. Write and test experiment:
   - ensure the main word is called `tst`
   - ensure this word performs any necessary setup, then calls `expStart` followed by the main loop
   - ensure the main loop calls `expUpdate ifleave` at the end
   - test the experiment using the simulator
2. Ensure the area of the Mars yard to be used is appropriately prepared: rake the surface for loose runs, to a depth of at least 5cm. Pack down (or leave packed) for packed runs. Move simulant if necessary to maintain the slope.
3. Power up Blodwen
4. **Ensure Blodwen has sufficient charge**. Power cycle (see below) and swap batteries if required, leaving the removed battery to charge.
5. Power down auxiliary PC on Blodwen
6. Connect to Blodwen's local network
7. Transfer experiment script to Blodwen
8. Log into Blodwen
9. Start monitor on the laptop
10. Start roverscript on Blodwen with experiment script
11. **check data is arriving on monitor**

12. move Blodwen to correct starting position (60cm from black tape marker for slope runs, as far left as possible for flat runs) using remote control commands
13. Start Vicon
14. Check the rover is visible in the Capture mode and is being tracked

G.2 Experiment

1. Enter the experiment word `tst` into roverscript and press enter
2. Switch to monitor, and hit S to start monitor logging
3. Press G (or EXP GO) to start experiment and hit Start on VICOM capture **at the same time**
4. **Check that RUN goes green in the monitor** indicating the *startexp* has run
5. Observe the monitor and Vicon screen to ensure the rover is tracked and running well
6. **Wait for end of experiment** : the base of the slope for flat runs, at least 1m clear of the top of the slope for slope runs
7. Stop the experiment with the following sequence (this is important, it stops post-experiment data being recorded):
   (a) Stop the monitor log by pressing K
   (b) Stop the Vicon capture
   (c) Stop the experiment by pressing G

G.3 Post

1. Click “Load in Post” on Vicon, and wait for data to load. If prompted, save to a new Trial file and make a note of the filename
2. Select a pipeline with Export to CSV at the end (“myfav” is already made thus)
3. Run the pipeline by pressing Play
4. Insert a USB stick into the appropriate hub and copy the CSV file
5. Create a new directory for the experiment in the project folder. from the session directory (under *Desktop/PATLAB* somewhere). Copy the file into a experiment directory, and call it *vicon.csv*
6. Copy the *capture.log* produced by the monitor into the data directory.
7. In the data directory, run

   ```
   ../collateViconAndRoverCSV vicon.csv capture.log >collated.csv
   ```

   This will collate Vicon and rover data, strip the non-experimental parts of the logs, and add missing values.
8. Clean up the data with appropriate use of R.

G.4 Powering down

Power Blodwen down by logging in, issuing `sudo poweroff` and waiting for the current draw to drop to less than 1A (with the battery door open so the control systems are unpowered). Once the current is low, switch off.

Appendix H

List of Angort words for controlling the rover

Generated automatically from the rover scripting system with the `listtex` word.

Module: udp

udpwrite	(string –)	write a string to the UDP port for the monitor
addudpvar	(name –)	create a new global which mirrors the monitor
handleudp	(–)	send and receive queued UDP data

Module: control

odo	(wheel –)	get odometry for a wheel
resetodo	(wheel –)	reset odometry for a wheel
sactual	(wheel –)	get actual steer position
lactual	(wheel –)	get actual lift position
dactual	(wheel –)	get actual drive speed
update	(–)	update the rover sensor data (takes time)
scurrent	(wheel – cur)	get steer motor current
lcurrent	(wheel – cur)	get lift motor current
dcurrent	(wheel – cur)	get drive motor current
rtemp	(–)	get a given (1-9) sensor's temperature above ambient
ambtemp	(–)	get the ambient temperature
temp	(index –)	get a given sensor's temperature reading
temps	(–)	show all temperatures
calib	(–)	perform preset calibration
exceptions	(–)	list all exceptions
setlegchecks	(bool –)	enable/disable leg collision checks
reset	(–)	reset exception states, overrides standard angort word!
s	(–)	emergency stop

Module: util

listtex	(modulename –)	dump all words of a module as LaTeX
cleantex	(–)	delete the latex dump file
play	(filename –)	use aplay to play a sound file
say	(string –)	use espeak to say something
panic	(string –)	throw a ScriptException and leave the program
setupdatetick	(time –)	set length of update tick in microseconds
time	(–time)	get time since start of program in seconds
delay	(n –)	delay for n microseconds

Module: calib

caliblift	(min max wheel –)	calibrate lift motor
setliftparams	(p i d cap decay octhresh stallcheck deadzone wheel –)	set all parameters for a lift motor
lshow	(wheel –)	show lift parameters
loverth	(val wheel –)	set lift overcurrent threshold
lidecay	(val wheel –)	set lift i-decay
licap	(val wheel –)	set lift i-cap
ligain	(val wheel –)	set lift i-gain
ldgain	(val wheel –)	set lift d-gain
lpgain	(val wheel –)	set lift p-gain
calibsteer	(min max wheel –)	calibrate steer motor
setsteerparams	(p i d cap decay octhresh stallcheck deadzone wheel –)	set all parameters for a steer motor
sshow	(wheel –)	show steer parameters
soverth	(val wheel –)	set steer overcurrent threshold
sidecay	(val wheel –)	set steer i-decay
sicap	(val wheel –)	set steer i-cap
sigain	(val wheel –)	set steer i-gain
sdgain	(val wheel –)	set steer d-gain
spgain	(val wheel –)	set steer p-gain
setdriveparams	(p i d cap decay octhresh stallcheck deadzone wheel –)	set all parameters for a drive motor
dshow	(wheel –)	show drive parameters
doverth	(val wheel –)	set drive overcurrent threshold
didecay	(val wheel –)	set drive i-decay
dicap	(val wheel –)	set drive i-cap
digain	(val wheel –)	set drive i-gain
ddgain	(val wheel –)	set drive d-gain
dpgain	(val wheel –)	set drive p-gain

Appendix I

The Angort language

I.1 Introduction

Angort[1] is a stack-based concatenative programming language with some functional features. The language has grown from a simple Forth-like core over time, and has been used primarily for robot control on an ExoMars rover locomotion prototype.

This is an extremely brief introduction to the language. It may be useful for readers unfamiliar with this style of programming to look into Forth, which is an older, more primitive (but faster and smaller) language from which much of the syntax of Angort was borrowed.

It combines the power and ease of a modern dynamic language with the convenience of a Forth-like language for controlling robots in real time. For example, on our rover we have the following definitions in the startup file:

```
# define a constant "wheels" holding a range from 1-6 inclusive

range 1 7 const wheels

# define a new word "d" with a single parameter "speed"

:d [speed:]

    # set a help text for this word

    :"(speed --) set speed of all drive motors"

    # for each wheel, set the required speed to the value
    # of the parameter

    wheels each {
        ?speed i!drive
    }
;

# slightly more complex word for steering

:t [angle:]
    :"(angle --) turn front wheels one way, back wheels opposite way"

    ?angle dup 1!steer 2!steer
    0 0 3!steer 4!steer
```

[1]The name is an entirely random pair of syllables, it has no significance.

```
    ?angle neg dup 5!steer 6!steer
;

# define a word to stop the rover by setting all speeds to zero

:s 0 d;
```

Once these words are defined we can steer the robot in real time with commands like:

```
2500 d
30 t
s
```

These will set the rover speed to 2500, turn it to 30 degrees, and stop it respectively. We can also directly type things like:

```
wheels each { i dactual .}
```

which will print the actual speeds of all the wheels. It's also possible to perform some functional programming with anonymous functions:

```
:sum |list:| 0 ?list (+) reduce;
```

will allow us to sum a list and print the results:

```
[1,2,3,4,5] sum .
```

or even:

```
1 1001 range (dup*) map sum .
```

to print the sum of the squares of the first 1000 integers. As can be seen, Angort is a very terse language.

In the examples given so far, words such as `dactual`, `!drive` and `?drive` are links to native C++ code: it is very easy to interface Angort with C++.

I.1.1 Downloading and building Angort

Angort can be downloaded from `https://github.com/jimfinnis/angort`. Once downloaded it, can be built with the following commands (from inside the top-level Angort directory):

```
mkdir build
cd build
cmake ..
make
```

This is for a Linux machine with CMake and the `readline` development libraries. The standard interpreter will then be installed in `cli/angortcli`.

I.2 Getting started: immediate mode

Running the interpreter will give a prompt:

```
0|0>
```

The two numbers are the number of garbage-collectable objects in the system and the number of items on the stack, respectively. The interpreter is in “immediate mode”, as opposed to “compilation mode” — any words entered will be compiled to bytecode and run when enter is pressed, rather than being added to a word definition.

In this mode, control constructs like `if...else...then` or loops cannot span more than one line. This is because such structures can only operate within a single bytecode block, and the current block is closed and run at the end of each line in immediate mode. Inside a word definition this rule does not apply.

I.3 The stack

Angort is a stack-based language: there is a single stack containing values, and most words change the contents of the stack in some way. For example,

```
3
```

by itself will just put the value 3 on the stack. Then

```
.
```

will pop the value from the top of the stack and print it.

```
3 4 + .
```

will push 3 and 4 onto the stack, then add them together replacing them with 7, and then print the 7. More complex expressions are built out of sequences of operations on the stack. For example, the expression

$$\sin(5 + 32 + \sqrt{43 \times 12})$$

would be written as

```
43 12 * sqrt 32 + 5 + sin
```

Converting expressions into this so-called *reverse Polish notation* is a little difficult at first, but rapidly becomes second nature[2]

[2]Most old calculators work using RPN, and quite a few of the more powerful programmables still do.

I.4 Defining new words

New words are defined with code of the form

```
:wordname ... ;
```

A simple example is:

```
:square dup *;
```

This will define a word `square` which will duplicate the value on top of the stack, multiply the top two values (effectively squaring the number[3]) and exit, leaving the squared number on the stack. This can then be used thus:

```
3 square 4 square + .
```

which will print 25, i.e. $3^2 + 4^2$. While defining a word, Angort is is compilation mode — words will be converted to bytecode but added to the definition of the new word rather than executed immediately. Word definitions can therefore span more than one line:

```
:factorial |x:|
    ?x 1 = if
        1
    else
        ?x ?x 1 - factorial *
    then
```

I.4.1 Word parameters and local variables

Until now, all the code has been perfectly valid Forth[4]. However, the stack manipulation required to write Forth is a challenge (and modern computers have a little more memory), so Angort has a system of named parameters and local variables. These can be defined by putting a special block of the form

```
|param1,param2,param3... : local1,local2,local3|
```

after the word name in the definition. Locals and parameters are exactly the same internally, but the values of parameters are popped off the stack when the new word is called.

Once defined, locals (and parameters) can be read (i.e. pushed onto the stack) using a question mark followed by the variable name. Similarly, a local is written (i.e. a value popped off the stack and stored in the local) by using an exclamation mark. For example,

```
:pointless |x,y:z|
    ?x ?y + !z
;
```

will read the two arguments into the locals x and y, add them, store the result in the local z, and then exit, throwing everything away. Note that a function with parameters but no locals is defined by leaving the part after the colon empty:

[3]This will produce an error if the value is not a number.

[4]With the exception of the factorial function, which uses local variables and recursion.

```
:magnitude |x,y:|
    ?x dup *
    ?y dup * +
    sqrt
;
```

while leaving the part before the colon empty will define a function with locals but no parameters:

```
:countToTen |:count|
    0!count
    {
        ?count 1+ dup !count
        dup .
        10 = ifleave
    }
;
```

I.4.2 Word documentation strings and stack pictures

Following the locals and parameters (or just the word name if there are none) there may be a word documentation string. It has the form

```
:"(before -- after) what the word does"
```

The section in brackets is known as a *stack picture,*[5] and describes the action of the word on the stack. The part before the hyphen shows the part of the stack removed by the word, and the part after shows its replacement. For example:

```
:dist |x1,y1,x2,y2:|
    :"(x1 y1 x2 y2 -- distance) calculate the distance between two points"
    ?x1 ?x2 - abs dup *     # this is (x1-x2)^2
    ?y1 ?y2 - abs dup *     # this is (y1-y2)^2
    + sqrt                  # sum them, and find the root
;
```

Note that the "after" part of the stack picture often doesn't refer to a named variable — it's just a label for the value left behind on the stack.

[5]Or sometimes a *stackflow symbol*.

I.5 Types

Angort is a dynamically typed language, with type coercion (weak typing). The following types are available:

Name	Definition	Example of literal
None	The nil object, specifying "no value"	`none`
Integer	32 or 64-bit integers depending on architecture	`5045`
Float	32-bit floats	`54.0`
String	Strings of characters	`"Hello there"`
Symbol	Single-word strings, internally stored as integers and generally used as hash keys	`` `foo ``
Code	Blocks of Angort code, anonymous functions	`( dup * 1 + )`
Integer range	A range of integers between two values with optional step	`0 4 range`
Float range	A range of floats between two values with step	`0 4 0.1 frange`
List	A array/list of values (stored as a dynamic array)	`[1,2,"foo"]`
Hash	A map implemented as a hash table, where keys can strings, symbols, integers or floats	``[% `foo 1, "fish" "fish"]``

"Code" above actually conflates two types internally — codeblocks, which have no environment; and closures, which consist of a codeblock and some stored variables. They appear identical to the user. There are some other types used internally, such as the types for iterators and deleted objects in a hash.

I.5.1 Coercions

- Integers and floats are coerced to strings in string contexts.
- In binary operations, if one of the operands is a float, both are coerced to floats:

```
0|0> 1 2 / .
0
0|0> 1.0 2 / .
0.500000
```

- In certain binary operations (currently just "+") if one of the operands is a string, both will be coerced to strings.

Much of the type system is untidy, and a major reorganisation is being planned.

I.6 Conditions

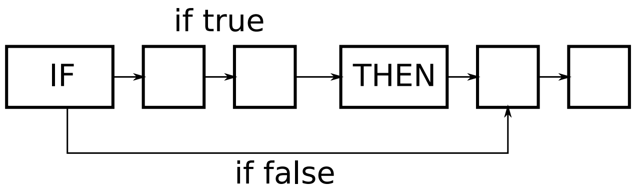
These have the form:

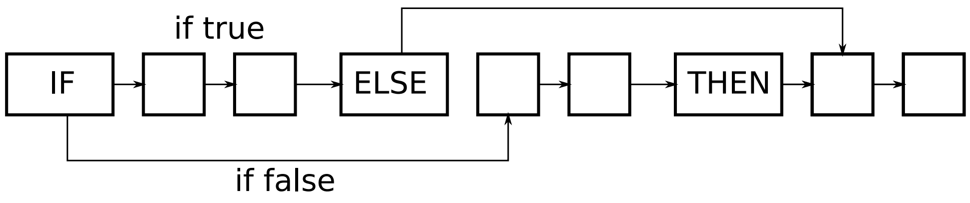
```
<condition> if <runs if true> then
```

The word `if` pops the item off the top of the stack, and jumps forward to just after the corresponding `then` if it is false (zero). There is also a two way branch:

```
<condition> if <runs if true> else <runs if false> then
```

This works the same way, except that the `if` jumps to after the `else` if the top value is false, and `else` jumps forward to just after `then`. This is shown in the figure below:





Here is a code example, printing whether a number is even or odd:

```
:evenodd
    # Get the number on the stack modulo 2, and use
    # whether it is zero or not as the condition.
    2 % if
        "number is odd"     # stack this string if there is a remainder
    else
        "number is even"    # stack this string if remainder is zero
    then                    # end the conditional
    .                       # print the top of the stack
;
```

We can run this with various values:

```
0|0> 4 evenodd
number is even
0|0> 3 evenodd
number is odd
```

Conditions can be nested.

I.7 Loops

There are two kinds of loops in Angort — infinite loops, which must be broken out of using `leave` or `ifleave`; and iterator loops, which loop over a collection (i.e. a hash or list) or a range. Both are delimited with curly brackets `{}`, but iterator loops use the `each` word before the opening brace.

I.7.1 Infinite loops

Any `leave` word will break out of an infinite loop:

```
:foo |:counter|         # a local variable "counter", no parameters
    0 !counter          # set counter to zero
    {                   # start loop
        ?counter.       # print the counter
        ?counter 1+     # increment the counter
        !counter        # and store it back
        ?counter 100=   # is it equal to 100?
        if leave then   # if so, leave the loop
    }
;
```

This will count from 0 to 99. The sequence

```
if leave then
```

is so common that it is coded to a special word of its own, and can be written as

```
ifleave
```

The word above could be written tersely as

```
:foo |:c| 0!c {?c. ?c 1+ !c ?c 100= ifleave};
```

Loops can be nested, and the `leave` words will jump out of the innermost loop in which they appear.

I.7.2 Iterator loops

Iterator loops loop over an *iterable* value. These are currently ranges, hashes and lists. They are delimited by curly braces like infinite loops, but are preceded by the `each` word. This pops the iterable off the stack and makes an iterator out of it, over which the loop runs. The iterator exists for as long as the loop does, and is destroyed when the loop completes. Again, iterator loops can be nested (and can be nested with infinite loops). With an iterator loop, the counting example can be rewritten as:

```
:foo
    0 100 range         # stack a range from 0-99 (see below)
    each {              # start the iterator loop
        i               # get the current item in the iterator
        .               # print it
    }                   # end of loop
;
```

Note that the end value of all ranges is the first value *outside* the loop. This looks somewhat odd with integer ranges, but serves to prevent float ranges relying on equality tests.

```
:foo
    0 10 0.09 frange    # float range from 0-10 in steps of 0.09
    each {i.}           # print them
```

will print about 9.990008 as its final value.

Although we've not covered lists and hashes yet, it's useful to know that we can iterate over the values in a list. For example

```
[1,2,3,4,5] each {i.}
```

will print the numbers from 1 to 5. Similarly, we can iterate over the keys of a hash:

```
[% "foo" "bar", "baz" "qux"] each {i.}
```

will print the hash's keys: "foo" and "bar".

I.7.2.1 Nested iterator loops

Each loop has its own iterator, even if they come from the same iterable:

```
:foo |:r|
    0 10 range !r           # store a range
    ?r each {               # outer loop
        i.                  # print value of outer loop iterator
        ?r each {           # inner loop
            i.              # print value of inner loop iterator
        }
    }
;
```

The range here is used twice, generating two separate iterators with their own current value.

While the word `i` retrieves the iterator value inside the current loop, inside nested iterator loops it is often useful to get the value in outer loops. This can be done with the `j` and `k` words, to get the outer and next outer iterator values:

```
:foo |:r|
    0 10 range !r           # store a range
    ?r each {               # outer loop
        ?r each {           # inner loop
            i j +           # print sum of inner and outer iterators
        }
    }
;
```

I.7.3 The state of the stack inside a loop

Note that the iterator is not kept on the normal stack — it is popped off when the loop starts and kept inside a special stack. The same applies to infinite loops: no loop state is kept on the stack. This leaves the stack free for manipulation:

```
# sum all the integers from 0 to x
:sumto |x:|
    0               # stack an accumulator
    0 ?x 1+ range   # stack a range from 0 to x inclusive
    each {          # pop the range, make an iterator,
                    # then transfer it to the iterator stack
        i           # get current iterator value
        +           # add it to the accumulator
    }               # end loop
;                   # finish and return the accumulator
```

I.7.4 Words to create ranges

There are three words to create a range on the stack:

name	stack picture	side-effects and notes
range	(x y – range)	create an integer range over the interval $[x, y)$ with a step of 1; i.e. a range from x to $y - 1$ inclusive.
srange	(x y s – range)	create an integer range over the interval $[x, y)$ with a step of s
frange	(x y s – range)	create an float range over the interval $[x, y)$ with a step of s

See Section I.4.2 for how stack pictures define the parameters and return values for a word.

I.8 Globals

Global variables are defined in two ways. The "polite" way is to use the `global` keyword, which creates a new global of the same following it, initially holding the nil value `none`:

```
0|0> global foo
0|0> ?foo.
NONE
0|0> 5!foo
0|0> ?foo.
5
```

The other way to define globals is simply to access a variable whose name begins with a capital letter. If no global or local exists with that name, a global is created with the initial `none` value:

```
0|0> 5 !Foo
0|0> ?Foo.
5
```

Globals, unlike locals, do not require a `?` or `!` sigil. If used "bare", their value will be stacked, unless they contain an anonymous function. In this case, the function is run (see Section I.10 below). It is, however, good practice to use the sigil.

I.9 Constants

Constants are similar to globals, but with the following differences:

- they are defined and set using the `const` keyword — this will pop a value off the stack and set the new constant to that value;
- they can never be redefined or written to.

Here are some examples which might be found at the start of a maths package:

```
3.1415927 const pi
2.7182818 const e

180 pi/   const radsToDegsRatio
pi 180/   const degsToRadsRatio

:degs2rads
    :"(degs -- rads) convert degrees to radians"
    degsToRadsRatio*
;

:rads2degs
    :"(rads -- degs) convert radians to degrees"
    radsToDegsRatio*
;
```

I.10 The true nature of word definitions

Words are actually global variables bound to anonymous functions. Given that anonymous functions are written as blocks of Angort in brackets (see below), then

```
:square |x:| ?x ?x *;
```

could also be written as

```
global square
(|x:| ?x ?x *) !square
```

with exactly the same end result. Referring to a global by using the `?` sigil will simply stack its value, whereas referring to it without the sigil will check if it is holds a code block or closure and run it if so, otherwise stack the value. This is useful in functional programming.

I.10.1 Sealed words

Combined with constants, this allows for "sealed" word definitions. In general all words can be redefined, but a word defined thus:

```
(|x:| ?x ?x *) const square
```

cannot be. This can be useful in package development.

I.11 Lists

Angort lists are actually array based, with the array resizing automatically once certain thresholds are passed (similar to a Java ArrayList). A list is defined by enclosing Angort expressions in square brackets and separating them by commas:

```
[]                # the empty list
[1,2,3]           # a list of three integers
["foo","bar",1]   # two strings and an integer
```

As noted above, lists can be iterated over. Lists can also contain lists, and can be stored in variables — more precisely, references to lists can be stored in variables:

```
0|0> [1,2] !A       # create a list and store it in global A
0|0> ?A!B           # copy A to B
0|0> 3 ?A push      # append an item to A
0|0> ?A each {i.}   # print A
1
2
3
0|0> ?B each {i.}   # print B - it also has the extra item!
1
2
3
```

Note that the list in B has also changed — it is the same list, just a different reference. The following are the words which act on lists, with their stack pictures:

name	stack picture	side-effects and notes
[	(– list)	creates a new list
,	(list item – list)	appends an item to the list in place, returning the list
]	(list item – list)	appends an item to the list in place, returning the list
get	(n list – item)	get the nth item from the list
set	(item n list –)	set the nth item in the list
remove	(n list – item)	remove and return the nth item
shift	(list – item)	remove and return the first item
unshift	(item list –)	prepend an item
pop	(list – item)	remove and return the last item
push	(item list –)	append an item
in	(item iter –)	return true if an item is in a list, integer range or hash keys

Note that the literal notation for lists — the square brackets and the comma — fall naturally out of the definition of the words[6]. The comma is a useful word, acting as a way to add items to a list on top of the stack without popping that list. This means we can write code to do a list copy:

```
:copylist |list:|
    []                  # stack an empty list
    ?list each {        # iterate over the list passed in
        i ,             # for each item, add it to the list on the stack
    }
;                       # finish and return the list on the stack
```

[6]There is an exception: if the tokeniser finds the sequence `[]` it discards the second bracket, allowing us to notate the empty list in a natural way.

I.12 Symbols

Symbols are single-word strings which are stored in an optimised form for easy and quick comparison (they are turned into unique integers internally). They are specified by using a back-tick (`) in front of the word. They're most useful as keys in hashes, covered in the next section.

I.13 Hashes

Hashes allow data be stored using keys of any hashable type (integers, floats, strings, symbols – anything except hashes and lists). Hashes are created using a similar syntax to the list initialiser, but with a `%` after the closing brace and data in key,value pairs[7]:

```
[%]
```

creates an empty hash, and

```
[%
    `foo "the foo string",
    `bar "the bar string"
]
```

creates a hash with two entries, both of which are keyed by symbols (although the keys in a hash can be of different types). We can add values to the hash using `set` which has the picture `(val key hash --)`:

```
[%]!H                           # create the empty hash
"the foo string" `foo ?H set    # add a string with the key `foo
"the bar string" `bar ?H set    # add a string with the key `bar
```

and read values using `get` which has the picture `(key hash -- val)`:

```
1|0> `foo ?H get
1|1> .
the foo string
```

We can also iterate over the hash's keys:

```
:dumphash |h:|
    ?h each {                   # iterate over the hash's keys
        i p                     # print key without trailing new line
        ":   " p                # print colon and spaces
        i ?H get                # get the value of the key in the hash
        .                       # and print it
    }
;
```

If we run this on the hash *H* defined above, we get

```
1|0> ?H dumphash
foo:   the foo string
bar:   the bar string
```

[7]This is a somewhat awkward syntax, but all the other bracket types were used elsewhere.

I.13.1 Shortcut symbol get and set

There is a syntactic sugar for retrieving the value of a key in a hash where the key is a literal symbol. Instead of using

```
`foo ?H get
```

we can use

```
?H?`foo
```

This has been added because this is by far the most common use-case. We also have the same ability to set a value in a hash with a literal symbol:

```
96 ?H!`temperature
```

instead of

```
96 `temperature ?H set
```

I.13.2 Words for hashes

Hashes can also use many of the same words as lists:

name	**stack picture**	**side-effects and notes**
[%	(– hash)	creates a new hash
,	(hash key value – hash)	adds a value to the hash, returning the hash
]	(hash key value – hash)	adds a value to the hash, returning the hash
get	(key hash – value)	get a value from the hash, or `none` if it is not present
set	(value key hash –)	set a value in the hash
remove	(key hash – value)	remove and return a value by key
in	(key hash –)	return true if a key is in the hash

Note that the comma and close bracket words examine the stack to determine if they are working on a list or a hash.

I.14 Garbage collection

Garbage collection is done automatically - up to a point. Specifically, the system does reference-counted garbage collection continuously, which works well in most cases. However, it is possible to create cycles:

```
[] !A               # make a list called A
[?A] !B             # make a list called B, referencing A
?B ?A push          # add a reference to B in A
```

Now there are two objects referencing each other — a cycle. This can happen in lists, hashes and closures. Reference-counted garbage collection will never delete these. Therefore it may be necessary in programs with a complex structure to call the full garbage collector occasionally. This is done manually by the word

```
gc
```

This will scan the entire system for cycles, and delete cycles of objects which are unreferenced from elsewhere. Incidentally, this is the same style of garbage collection used by Python, but Angort does not run the full collector automatically.

I.15 Functional programming

Anonymous functions are defined with brackets, which will push an object representing that function (and any closure created) onto the stack. This can then be called with `call` (which can be abbreviated to "`@`") Such functions may have parameters and local variables.

For example, this is a function to run a function over a range of numbers, printing the result:

```
:over1to10 |func:|
    1 10 range each { i ?func@ . } ;
```

With this defined, we can now use it to show the squares of those numbers:

```
(|x:| ?x ?x *) over1to10
```

or more simply

```
(dup *) over1to10
```

I.15.1 Words for dealing with functions

name	stack picture	side-effects and notes
map	(iter func – list)	apply a function to an iterable, giving a list
reduce	(start iter func – result)	set an internal value (the accumulator) to "start", then iterate, applying the function (which must take two arguments) to the accumulator and the iterator's value, setting the accumulator to this new value before moving on.
filter	(iter func – list)	filter an iterable with a boolean function

We can now list all the squares of a range of numbers:

```
0 100 range (dup *) map each {i.}
```

We can also write a function to sum the values in an iterable using `reduce`:

```
:sum |iter:|
    0           # the accumulator value starts at zero
    ?iter       # this is the iterable
    (+)         # and this is the function
    reduce      # repeatedly add each item to the accumulator,
                # setting the accumulator to the result. When
                # finished, return the accumulator.
;
```

I.15.2 Closures

Anonymous functions can refer to variables in their enclosing function or word, in which case a closure is created to store the value when the enclosing function exits. This closure is mutable - its value can be changed by the anonymous function. For example, consider the following function:

```
:mkcounter |:x|     # declare a local variable x
    0!x             # set it to zero
    (               # create a function
        ?x dup .    # which prints the local
        1+ !x       # and increments it
    )
;
```

This returns an anonymous function which refers to the local variable inside the function which created it. We can run this function and store the returned function in a global:

```
mkcounter !F    # run it and store the returned function+closure
```

If we now run

```
    ?F call
```

a few times, we will see an incrementing count - the value in the closure persists and is being incremented. We can call mkcounter several times and each time we will get a new closure.

It's important to note that closures are copy closures - the anonymous function makes a copy of the local, and all changes inside the anonymous function happen to that copy, not the local in the parent:

```
:foo |:x| 4!x       # store 4 in the local x
    (10 !x)         # store 10 in the closure's version of x
    @               # run the anonymous function
    ?x .            # this will print 4, because the local hasn't changed.
;
```

This is a consequence of Angort's extremely simple syntax.

I.16 Built-in words

I.16.1 Stack manipulation

There exist a number of words whose sole purpose is to manipulate the stack. These words, and their stack pictures, are given in the table below.

name	**stack picture**
dup	(a – a a)
drop	(a –)
swap	(a b – b a)
over	(a b – a b a)

Note that this is a much smaller set than the set of Forth stack words, which includes `rot`, `roll`, `nip` and `tuck`. The local variable system makes such complex operations unnecessary.

I.16.2 Binary operators

In the following operations, these conversions take place:

- If one of the operands is a string, the other will be converted to a string. Only ”+” and comparison operators will be valid.
- If one of the operands is a float and the other is an integer, the integer will be converted to a float and the result will be a float.
- The comparison operators will do identity checks on objects (lists, ranges etc.), not deep comparisons.
- The comparison functions actually return integers, with non-zero indicating falsehood.

name	**stack picture**	**side-effects and notes**
+	(a b – a+b)	
-	(a b – a-b)	
$*$	(a b – a*b)	
/	(a b – a/b)	
%	(a b – a%b)	remainder (”mod”) operator
$>$	(a b – a$>$b)	string comparison works as expected
$<$	(a b – a$<$b)	string comparison works as expected
=	(a b – a=b)	equality test: string comparison works as expected
!=	(a b – a!=b)	inequality test: string comparison works as expected
and	(a b – a$\wedge$b)	binary and – inputs are coerced to integers, nonzero is true
or	(a b – a$\vee$b)	binary or – inputs are coerced to integers, nonzero is true

I.16.3 Unary operators

name	stack picture	side-effects and notes
not	(a – !a)	logical negation of a boolean
neg	(a – -a)	arithmetic negation of float or integer
abs	(a – |a|)	absolute value
isnone	(a – bool)	true if value is `none`

I.17 Getting help

There are many other functions and operations available in Angort. These can be listed with the `list` word, and help can be obtained on all words with `help` (except the very low-level words compiled directly to bytecode, which are all covered above):

```
0|0 > "filter" help
filter: (iter func -- list) filter an iterable with a boolean function
0|0 >
```

I.18 Topics missing from this document

There are several topics which will be discussed in later versions of this document:

- other built-in words;
- writing words in C++ and adding them to Angort;
- properties (which look like global variables but invoke Angort code);
- embedding an Angort interpreter in other systems.

Appendix J

Libkoki

Libkoki is a "computer vision library for detecting and estimating the position of fiducial markers" [23]. It takes images from a camera and locates markers within them. These markers are images made up of black and white squares, printed onto paper or card. The library, once calibrated correctly for the camera, reports the locations and orientations of all the markers found, relative to the camera. It was evaluated as a possible localisation system for the rover, to provide a distance along the track, when it was thought that the Vicon system may have been unavailable.

A set of five large markers were printed onto A4 paper, so they would be visible by a consumer webcam over the required range of 3-4 meters. All five markers were then placed on a flat surface (the PATLab door) so that they were coplanar. The camera was positioned facing the surface, its optical axis roughly perpendicular to the surface. The output of the library consists of a set of locations and orientations, so the final output value was obtained by finding the mean z coordinate of all the visible markers.

Calibration of the library requires providing the focal length and image size of the camera Both were given for the camera used (a Logitech c270), but the reported focal length gave incorrect results.

Therefore, the camera was positioned at a known distance from the markers, and a program written to perform a binary search on the focal length until the reported distance matched the true distance. This was done at a range of distances, every 50cm from 1 to 3.5m, and the final focal length was the mean of the values obtained.

Once calibration was complete, the true distance against the average distance from all visible markers was plotted, resulting in the the plot in Figure J.1. Five runs were made. The figure shows an error of within 5cm until we reach 3.5m, where the accuracy (and the repeatability) begins to fall off markedly until we reach an error of 10cm. Since the run of the gait experiments was estimated to be around 4m, of which the final metre or so contained the incline so important to the results, this was unfortunate.

Another issue with this system which would have caused problems is that the rover is constantly moving, destabilising the image. No experiments were done on marker detection for a moving rover, because a decision was made to persevere with getting the Vicon system working (with the considerable assistance of Mark Neal and Dave Price). However, it is possible that the accuracy could have been sufficient for the experiments to have still gone ahead.

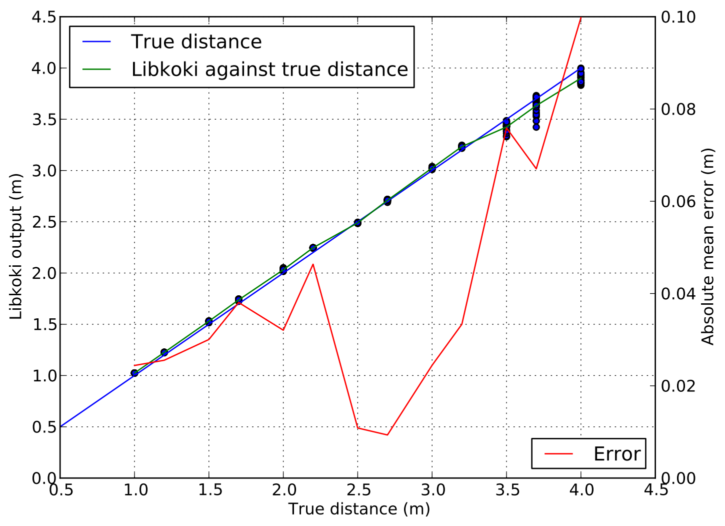


Figure J.1: Libkoki output distance against true distance. The blue line shows correct calibration (output distance = true distance) while the green line is constructed by drawing a line between the means of the results. The red line was obtained by subtracting the output value from the correct value.

Annotated Bibliography

[1] R. Altendorfer, N. Moore, H. Komsuoglu, M. Buehler, J. Brown, H.B., D. McMordie, U. Saranli, R. Full, and D. Koditschek, "Rhex: A biologically inspired hexapod runner," *Autonomous Robots*, vol. 11, no. 3, pp. 207–213, 2001. [Online]. Available: http://dx.doi.org/10.1023/A%3A1012426720699

Describes a robot implementing an alternating tripod gait.

[2] R. C. Arkin, "Homeostatic control for a mobile robot: Dynamic replanning in hazardous environments," in *1988 Robotics Conferences*. International Society for Optics and Photonics, 1989, pp. 407–413.

A important early work in artificial endocrine systems by one of the founders of reactive robotics, describing how an AES can assist in homeostasis — the maintenance of a safe internal environment for the system.

[3] R. A. Brooks, "A robot that walks; emergent behaviors from a carefully evolved network," *Neural computation*, vol. 1, no. 2, pp. 253–262, 1989.

A famous paper describing a very successful legged robot, Genghis, based on the subsumption architecture, built up of several layers, and capable of quite complex behaviour.

[4] R. A. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14–23, 1986.

A seminal paper in reactive robotics, this describes the subsumption architecture and its application in a simple mobile robot which performs complex tasks without central planning.

[5] R. G. Brown, "Exponential smoothing for predicting demand," in *OPERATIONS RESEARCH*, vol. 5, no. 1. INST OPERATIONS RESEARCH MANAGEMENT SCIENCES 901 ELKRIDGE LANDING RD, STE 400, LINTHICUM HTS, MD 21090-2909, 1957, pp. 145–145.

The original source of the exponential smoothing algorithm.

[6] D. Carr, ported by Nicholas Lewin-Koh, and M. Maechler, *hexbin: Hexagonal Binning Routines*, 2013, r package version 1.26.3. [Online]. Available: http://CRAN.R-project.org/package=hexbin

Description of the hexbin R package, used to generate hexagonal binned plots, and used for exploring the data, but not used in the final plots.

[7] A. D. Craig, "How do you feel? interoception: the sense of the physiological condition of the body," *Nature Reviews Neuroscience*, vol. 3, no. 8, pp. 655–666, 2002.

Source of the definition of the term "interoception". From the abstract: "Recent functional anatomical work has detailed an afferent neural system in primates and in humans that represents all aspects of the physiological condition of the physical body."

[8] J. A. Crisp, M. Adler, J. R. Matijevic, S. W. Squyres, R. E. Arvidson, and D. M. Kass, "Mars exploration rover mission," *Journal of Geophysical Research: Planets*, vol. 108, no. E12, pp. n/a–n/a, 2003. [Online]. Available: http://dx.doi.org/10.1029/2002JE002038

Fairly detailed overview of the Mars Exploration Rover mission (Spirit and Curiosity), written before the rovers landed. This provides an overview of the rover technology, and some useful data (such as rover speeds).

[9] D. B. Dahl, *xtable: Export tables to LaTeX or HTML*, 2014, r package version 1.7-3. [Online]. Available: http://CRAN.R-project.org/package=xtable

Description of the xtable R package, which can generate LaTeXtables from text. Use to produce statistical tables.

[10] L. Ding, H.-b. Gao, Z.-q. Deng, and J.-g. Tao, "Wheel slip-sinkage and its prediction model of lunar rover," *Journal of Central South University of Technology*, vol. 17, no. 1, pp. 129–135, 2010. [Online]. Available: http://dx.doi.org/10.1007/s11771-010-0021-7

Describes slip-sinkage of wheeled rovers on loose lunar regolith, which is similar in many ways to Martian regolith. Although not immediately applicable to wheel-walking, the conclusions — that slip-sinkage is a problem and that the height of wheel lugs can affect this — are interesting.

[11] J. Duysens and H. W. Van de Crommert, "Neural control of locomotion; part 1: The central pattern generator from cats to humans," *Gait & posture*, vol. 7, no. 2, pp. 131–141, 1998. [Online]. Available: http://www.cs.cmu.edu/~cga/legs/nclpt1.pdf

An interesting, but perhaps not directly relevant paper describing experimental evidence for the presence and nature of central pattern generators in vertebrates — neural structures which produce rhythmic patterns, including stepping motions, even while disconnected from other inputs.

[12] J. Finnis, "Monitor configuration file syntax," https://github.com/jimfinnis/angort, Feb. 2013, accessed April 2014.

A document describing the syntax of configuration files used by the monitor, which reads data from the rover, shows it onscreen and captures it to a file.

[13] J. Finnis, "Angort programming language," https://github.com/jimfinnis/angort, Feb. 2014, accessed February 2014.

Angort is a garbage-collected, Forth-based language with functional elements, specifically designed for robot control and prototyping. I'm using it in the initial stages of the project and to provide command control in later stages.

[14] J. Finnis, "Blodwen rover documentation," http://pale.org/rover.pdf, Feb. 2014, accessed February 2014.

Comprehensive documentation of the rover's control hardware and software. No doubt I'll be consulting this a great deal.

[15] J. C. Finnis and M. Neal, "A Simple Drive Load-Balancing Technique for Multi-wheeled Planetary Rovers," in *Proceedings of TAROS (Towards Autonomous Robotic Systems), Oxford, UK, August 2013*, 2013. [Online]. Available: http://pale.org/taros2013.pdf

A paper written by the author on using an model of inflammation to balance power management across the same six-wheeled rover used in these experiments.

[16] M. Friendly, "Corrgrams: Exploratory displays for correlation matrices," *The American Statistician*, vol. 56, no. 4, pp. 316–324, 2002.

Describes the corrgram, a nice way of displaying correlation matrices.

[17] M. P. Golombek, J. A. Grant, T. J. Parker, D. M. Kass, J. A. Crisp, S. W. Squyres, A. F. C. Haldemann, M. Adler, W. J. Lee, N. T. Bridges, R. E. Arvidson, M. H. Carr, R. L. Kirk, P. C. Knocke, R. B. Roncoli, C. M. Weitz, J. T. Schofield, R. W. Zurek, P. R. Christensen, R. L. Fergason, F. S. Anderson, and J. W. Rice, "Selection of the mars exploration rover landing sites," *Journal of Geophysical Research: Planets*, vol. 108, no. E12, pp. n/a–n/a, 2003. [Online]. Available: http://dx.doi.org/10.1029/2003JE002074

Paper describing the selection process for the Mars Exploration Rover landing sites, including engineering constraints such as solar power availability.

[18] J. Graffelman, *calibrate: Calibration of Scatterplot and Biplot Axes*, 2013, r package version 1.7.2. [Online]. Available: http://CRAN.R-project.org/package=calibrate

Description of the calibrate R package, used for adding text to points in plots, used for exploring data but not used in the final code.

[19] B. D. Harrington and C. Voorhees, "The challenges of designing the rocker-bogie suspension for the mars exploration rover," in *37th Aerospace Mechanisms Symposium, May 19-21, 2004, Johnson Space Center, Houston, Texas*. Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2004, 2004.

Description of the rocker-bogie suspension of the NASA MER rovers, discussing the various design decisions, which is useful to compare and contrast with the ExoMars Concept-E system.

[20] G. C. Haynes and A. A. Rizzi, "Gaits and gait transitions for legged robots," in *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*. IEEE, 2006, pp. 1117–1122.

Useful data on how gaits are constructed and notated, which I looked at and then moved beyond — they could not easily be applied to hybrid gaits.

[21] J. Hidalgo, F. Cordes, and D.-R. I. Center, "Kinematics modeling of a hybrid wheeled-leg planetary rover."

Discusses a model of a hybrid rover, which is then tested in simulation, showing the immature nature of research into walking hybrid wheeled-leg rovers.

[22] G. Hughes, "The co-ordination of insect movements i the walking movements of insects," *Journal of Experimental Biology*, vol. 29, no. 2, pp. 267–285, 1952.

Interesting discussion of the true complexity of insect gaits, showing that it's a little more involved than the common ideas used in robotics.

[23] C. Kirkham, "Libkoki, a computer vision library for detecting and estimating the position of fiducial markers," https://github.com/chrisjameskirkham/libkoki, Mar. 2014, accessed March 2014.

Libkoki is a library for visually detecting the positions of markers placed in the world. It was used as a backup location technique, in case the Vicon was unavailable or unsuitable.

[24] V. Kucherenko, A. Bogatchev, and M. Van Winnendael, "Chassis concepts for the exomars rover," in *Proceedings of the 8th ESA Workshop on Advanced Space Technologies for Robotics and Automation, Noordwijk, The Netherlands*, 2004.

A description of the various chassis concepts for the ExoMars rover. Our rover uses Concept E, and the gaits tested were designed for this concept.

[25] M. Maimone, Y. Cheng, and L. Matthies, "Two years of visual odometry on the mars exploration rovers," *Journal of Field Robotics*, vol. 24, no. 3, pp. 169–186, 2007.

A description of the visual odometry system for the MER rovers, describing how a ground truth for position can be obtained to moderate levels of accuracy by camera images alone.

[26] S. Moreland, K. Skonieczny, D. Wettergreen, V. Asnani, C. Creager, and H. Oravec, "Inching locomotion for planetary rover mobility," in *Aerospace Conference, 2011 IEEE.* IEEE, 2011, pp. 1–6.

Describes and analyses the inching gait on a four-wheeled rover, showing a gain in tractive force caused by a "unified deep soil mass" created by each static wheel.

[27] M. Neal, "Once more unto the breach: Towards artificial homeostasis," *Recent Developments in Biologically Inspired Computing*, pp. 340–365, 2005.

This paper describes the basic concepts behind artificial endocrine systems and how they can be coupled with artificial neural networks to provide important new capabilities, particularly with regard to homeostasis.

[28] M. Neal, J. Finnis, T. Owen, C. Chetwood, and G. Dodds, "Endocrine inspired, power aware scheduling for a Mars rover," *IEEE Transactions on Robotics*, 2014, manuscript submitted for publication (copy on file with author).

A paper the author was involved in last year, using an artificial endocrine system to better manage mission scheduling in a simulation of the ExoMars mission

[29] N. Patel, R. Slade, and J. Clemmet, “The ExoMars rover locomotion subsystem,” *Journal of Terramechanics*, Mar. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.jterra.2010.02.004

Describes the wheel configuration of the ExoMars rover chassis - our rover is an RCL-E prototype.

[30] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2008, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

The manual for the R programming language for statistical computing, used extensively for plots and analysis.

[31] B. Ripley, “Package ‘pspline’,” http://cran.r-project.org/web/packages/pspline/pspline.pdf, accessed April 2014.

Description of the polynomial spline smoothing package in R. Used widely for generating the splines which approximate quantities for calculating metrics, primarily because the splines it produces are differentiable.

[32] F. Ritz and C. Peterson, “Multi-mission radioisotope thermoelectric generator (mmrtg) program overview,” in *Aerospace Conference, 2004. Proceedings. 2004 IEEE*, vol. 5, March 2004, pp. –2957 Vol.5.

Description of the MSL Curiosity rover’s radioisotope power generator.

[33] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, Dec. 2009, 9780132071482.

A standard textbook in the field, this provides a good introduction to perceptrons and neural networks.

[34] C. Sauze and M. Neal, “Long term power management in sailing robots,” in *OCEANS, 2011 IEEE - Spain*, 2011, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/Oceans-Spain.2011.6003406

A paper on an application of AES to power management in robotic boats — work which led into the ENDOVER project which in turn inspired the current study.

[35] M. F. Silva and J. T. Machado, “A historical perspective of legged robots,” *Journal of Vibration and Control*, vol. 13, no. 9-10, pp. 1447–1486, 2007.

Very useful summary of the state of the art (as of 2007) and evolution of legged robots.

[36] J. Timmis and M. Neal, “Timidity: A useful emotional mechanism for robot control?” *Informatica*, 2003.

A discussion of the concept of the artificial endocrine system, accompanied by a simple experiment demonstrating its utility — and also showing that such a system produces interesting behaviour humans might attribute to emotions.

[37] D. Wettergreen and C. Thorpe, "Gait generation for legged robots," in *IEEE International Conference on Intelligent Robots and Systems*, 1992.

Gives a useful overview of some approaches to gait generation, including the subsumption architecture eventually selected.

[38] H. Wickham, *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. [Online]. Available: http://had.co.nz/ggplot2/book 978-0-387-98140-6.

A library for producing elegant plots, but not used because of difficulties in producing data showing several values plotted on multiple Y axes (something Wickham believes is confusing, so he doesn't permit it).

[39] K. Wright, *corrgram: Plot a correlogram*, 2013, r package version 1.5. [Online]. Available: http://CRAN.R-project.org/package=corrgram

Documentation for the R corrgram package

[40] K. Yoshida and H. Hamano, "Motion dynamics of a rover with slip-based traction model," in *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, vol. 3. IEEE, 2002, pp. 3155–3160.

Discusses modelling a wheeled rover using Bekker and Reece-Wong models, providing a useful slip ratio metric.