Interfacing between a cerebellar model and a robotics simulator

Harm Aarts

Utrecht University
Department of Information and Computing Sciences

Internal supervisors:
dr. Marco Wiering
dr. Frank Dignum

External supervisor:
dr. Harri Valpola

INF/SCR-06-21

Date of submission: 24 November, 2006

Abstract

In this master's thesis a software library will be presented to be used in conjunction with a robotic simulation platform. This library enables roboticists to conduct research without the usual constraints imposed by a simulator. It enables fast-prototyping of models with Matlab and the use of distributed models.

A model of the cerebellum is used to showcase this library. First a simple demonstration will be given showing the general working of the model and its stability when faced with noise. A second demonstration will be given showing the ability of the model to cope with and use complicated inputs. These inputs are obtained through the processing of camera images gathered from the simulation.

The theory behind the cerebellar model and the visual processing techniques will be discussed in-depth as well as the design decisions leading to the software library. Doing so will facilitate future extensions to both the cerebellar model and the software library.

Acknowledgements

This thesis resulted from a nine month stay in Finland at the Helsinki University of Technology. First of all I would like to express my gratitude to Harri Valpola my supervisor at location in Finland. Not only was his guidance during the project invaluable but his relentless optimism kept me going.

Further I would like to thank dr. Marco Wiering who not only during my thesis supported me as first supervisor but also during my years at the Utrecht University. It was he who got me interested in the Artificial Intelligence research field. Another word of gratitude for dr. Frank Dignum. During the years in which I did my master it was he who always managed to find the spare time to supervise the extra curricular projects I (with fellow students) would come up with. Even now he found the patience to go over my work one more time. Many thanks to both of you.

I would also like to thank my colleagues Janne Hukkinen, Heikki Joensuu, Ville Mannari and Antti Yli-Krekola. Whether it was for the shared frustrations, the great rewrite of the model or just the great discussions and drinks I loved working with you!

What would I have done without your continuous support Hagar? Whenever I needed it you would patiently listen to me. The last stretch your support was invaluable. Your love and understanding made this thesis happen.

Last but by no means least I would like to thank my family for their support even when great distance separated us.

Contents

Chapter 1

Introduction

Our world is filled with challenges, we decide what to have for lunch and we evade onrushing cars. Beside these obvious examples we need to navigate innumerable challenges we are not even conscious of. We walk without a moments thought, we recognise a friend in a crowded bar in an instant. On closer inspection these seemingly trivial problems are tremendously difficult. Robots have barely began walking [Geng et al., 2006] and the recognition of your friend in real time is still some time away.

The brain, with which we solve these problems, is an excellent generic problem solver. It is therefore no surprise that the brain has been subject to intense study through most of the modern time. In past research a lot of questions have been answered but even more have been raised. Many different brain areas have been identified, categorised and described. But the function or the behavioural consequences of these regions are not always entirely clear. The basal ganglia [Prescott et al., 2006], for example, are speculated to play a role in action selection but how this is done remains unclear. Researchers have identified many different neurotransmitters but the role of each has not been conclusively determined. The fact that our visual system is able to abstract from simple features is clear, how exactly this is done remains largely a question.

All the questions we currently entertain might not be solvable by looking at just the one brain region involved. A broader approach is called for. In this approach a holistic view on the brain is taken. Instead of looking at the individual components, one takes a look a certain functional properties of the brain and tries to determine which system or subsystem in the brain is involved. Through modelling of these systems and by putting them to the test we learn both to understand the brain and how to construct one. The functional direction of this approach would be a leading aspect. We know how neurons respond and what they are supposed to compute, there is no reason to model the neuron in every single detail. As long as the neuron operates on a biologically functional level we can abstract from the actual biological implementation.

1.1 Motivation

This thesis will strive to facilitate research in this functional but biologically inspired brain research. There are several constraints which need to be met if

this is to succeed.

1.1.1 Coping with reality

The brain is this wonderful structure not because it is just that, wonderful. There needs to be a reason for it to be like this. The world in which we live as humans is incredibly complex. For all the tasks we perform in order to keep alive we need a brain as complex as it is was called for. Evolutionary pressure made sure of that. That means if we are to study the brain it needs to be set in a challenging enough environment. The real world would of course be ideally suited with regard to complexity was it not for the fact that even the most sophisticated systems nowadays have huge problems coping with the real world. Most systems are set in a laboratory environment to alleviate the strain of the real world. And while this laboratory environment is a simplification of the real world it, in our case, should still be rich enough to provide the challenge necessary to develop meaningful systems.

1.1.2 Sharing the load

The brain is an incredibly complex system which no computer available now or in the near future can beat in terms of computational power. Vast amounts of data from a multitude of sensory systems is processed in parallel. And while many of the systems in the brain interact there are also a good many number of systems that hardly interact at all. Think for example about the system responsible for walking and the one responsible for talking. These have nothing to do with each other and can be logically separated. This property can be exploited in a sense that it would now be possible to run parts of the final model on different computing units thus distributing the work load.

1.1.3 Type of research

The very nature of this type of research calls for a flexible environment in which to do this research. The environment should be as free from distracting tasks as possible and should support the needs of the researcher active in this field. Modelling is an essential part of brain research and as such should be as easy as possible.

1.2 Consequences

These requirements call for either a lab environment in which a physical robot can be situated, or a simulated robot in a sufficiently complex simulated world. The sensors and actuators of this robot need to be accessible by more than one process for it needs to be usable in a distributed environment.

The simulator option is preferred because this has three distinct advantages over a physical robot. A simulator could be instantiated by multiple people at the same time and as more people are working on the project of building an actual brain this is clearly desirable. A simulator would also remove the need for a dedicated 'playing' field for the robot to live in. Clearly this eliminates the required space and besides that the simulated environment is easier controlled

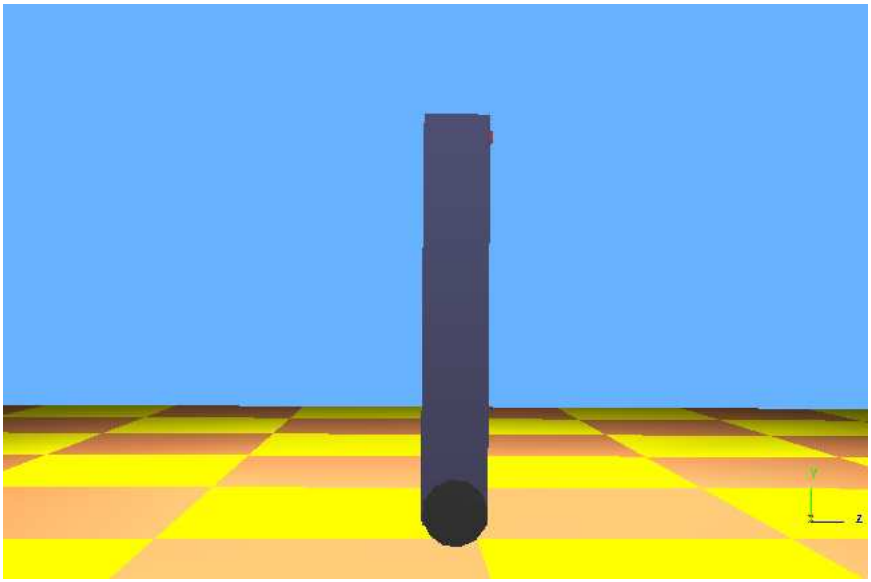


Figure 1.1: The environment and the modelled robot.

to provide a complex enough world but not too complex to overwhelm the system. Last, choosing a simulated environment reduces the need for maintenance allowing people to concentrate on the things that actually matter.

For these reasons we choose Webots [Michel, 2004] as our simulator because is supports both Java and C++ programming language and it sports a realistic physics environment. Moreover robots and the worlds in which they operate can be easily designed. An additional advantage is the wide spread use of Webots throughout the roboticists community, this ensures a broad support for the product and a potentially exchange possibilities.

This thesis will present a mean to facilitate the requirements given above. This will take form as a library written in Java which allows communication between Webots and Matlab over a TCP/IP connection. The choice for Matlab was obvious given the computational requirements, Matlab is masterfully optimised for the mathematical computations predicted to be required for this work. Moreover Matlab is well suited for the fast prototyping this kind of research calls for. This library will be put to the test in a couple of different situations. These situations are meant to demonstrate the usefulness of the library and to provide a future stepping stone to continue research in building a brain.

More specifically a robot is modelled in Webots, see Figure 1.1, which is inherently unstable. It's task is to learn to balance itself in various situations. For this a model of the cerebellum is used which is implemented in Matlab.

The remainder of the text is organised in six chapters. First a set of theoretically inclined chapters is presented, introducing the cerebellar model followed by a detailed description of several preprocessing techniques which the cerebellar model needs. The self-organising map and k-means clustering will be discussed and closing that chapter the theory of principal component analysis is explained. The fourth chapter demonstrates the cerebellar model and preprocessing techniques with a set of experiments. Chapter five shows the work

done on the communications library, this is where the design decisions and programming techniques are explained. The last two chapters contain the usual discussion and conclusion.

Chapter 2

The cerebellum

First and foremost in these experiments was the use of a model of the cerebellum. Below first an elaboration on the supposed functions of the cerebellum and next on the derived model.

2.1 Why look at the cerebellum?

When looking to build an artificial system which has the same properties as exhibited by the brain, smooth motor control is one of the most basic tasks. Without it not many other functions can practically be implemented or conceived. For a system to have smooth motor control it is easy to see that some form of prediction should be incorporated. It is generally impossible to act in a smooth fashion without some knowledge of things to come. For example if walking down a staircase would be based only on information currently available every minute movement would require intensive processing of, for example, touch sensory information. Am I already at the bottom of the steps? Is there another step or am I already on ground level? Having additional visual information providing us with the ability to anticipate enables us to just run down the stairs without a seconds thought. And even when we would lack the ability to predict from the current information and it *would* be possible to use some kind of intense sensory processing on the instantaneous data, the information required usually is available only after an inherent delay. Our neurons just are not instantaneous. A limp takes time to execute a movement due to inertia. A feedback loop of action, result, measurements, decision and action again takes time in all steps. It is therefore imperative to predict the outcome of certain actions and incorporate this knowledge into the action being taken.

The cerebellum is generally supposed to play an important role in predictive motor control [Smith, 1998], [Albus, 1971]. This predictive control exercised by the cerebellum is very important in everyday tasks. While learning to ride a unicycle or to juggle one first has to concentrate hard on the task at hand. But after a while one can perform these tasks without conscious concentration. The cerebellum learnt to perform these tasks. Given a certain input, in the case of juggling the position of the hands and the visual input concerning the ball, the cerebellum is able to 'automatically' execute the correct movements. This has the great advantage that you can now focus on other tasks, leaving the seem-

ingly trivial ones to the cerebellum. In relation to an artificial system this has the advantage that you can forfeit the difficulty of selecting relevant inputs. An artificial system with these properties would pick up the correct and relevant inputs to perform a certain task without predetermining the relevant inputs. More important the cerebellum now understands the system which it is controlling, in a way learning an internal model of the system, see [Kawato, 1999]. Using this internal model it is now able to anticipate the reaction of the system and thus exert smooth motor control.

2.2 Inner workings of the cerebellum

As mentioned in the previous section the most important function of the cerebellum is predictive control. Given a certain set of inputs it predicts a future reflexive response and based on this determines the appropriate response to avoid this. As the observant reader might have noticed in the examples the cerebellum is (amongst others) provided with processed data. It is explicitly not the task of the cerebellum to recognise a ball or to tell whether or not this sound is made by a gunshot or a jar falling. It *is* supposed to process these signals and determine their relevance to the tasks at hand. Seeing a ball is not important for balancing on your bike but it is important when juggling them. When we will be using the cerebellum we will provide it with different 'fingerprints' of textures but we will not classify them for it. It is up to the cerebellum to determine what is important.

So the cerebellum should obviously learn the model of the system (or plant as it is called in control theory) it is controlling. Various studies have shown that the cerebellum is learning through reflexes, see [Grethe and Thompson, 2003] and [Anastasio, 2003]. A reflex can be seen as an error signal, whenever a reflex occurs something clearly went wrong, you might have been falling or you might have touched something hot. Using these reflexes provides the cerebellum with a teaching signal. Depending on pure reflexes to control the system would amount to designing exact reflexes for the situation, taking into account the inherent delays and inertia in the system and its complex dynamics. This task is daunting at best and very labour intensive. Most of the time it is not even possible to design these reflexes because of the fact that reflexes are supposed to be hard-wired and in the case of our body the mechanical properties of it change radically during our life time. So the best a reflex can do is give a rough estimate of the correct action. The task of the cerebellum is now to predict when a reflex is going to happen and take action to prevent it. Thus learning to control the plant.

We can identify four different components from which we can build a model (of the cerebellum) which is able to generate smooth motor control. We have the plant or the system which needs to be controlled, the environment the plant is situated in, the adaptive controller and the teaching signal in the form of a reflex. Next we will formalise these four components thus arriving at a working model of the cerebellum.

2.3 Cerebellar model

The model of the cerebellum is a derivative of the Feedback Error Learning Model(FELM) first suggested by [Kawato and Gomi, 1992]. This suggestion has been put forward in several of their studies to biologically inspired motor control solutions. In a study by [Barto et al., 1999] the model was used to approximate the functionality of the cerebellar cortex further strengthening the biological plausibility of the model. Last in [Joensuu, 2006] the model was used for a pole balancing task and the model presented here is almost identical to that one. Figure 2.1 depicts the schematic overview of the model as we will use it.

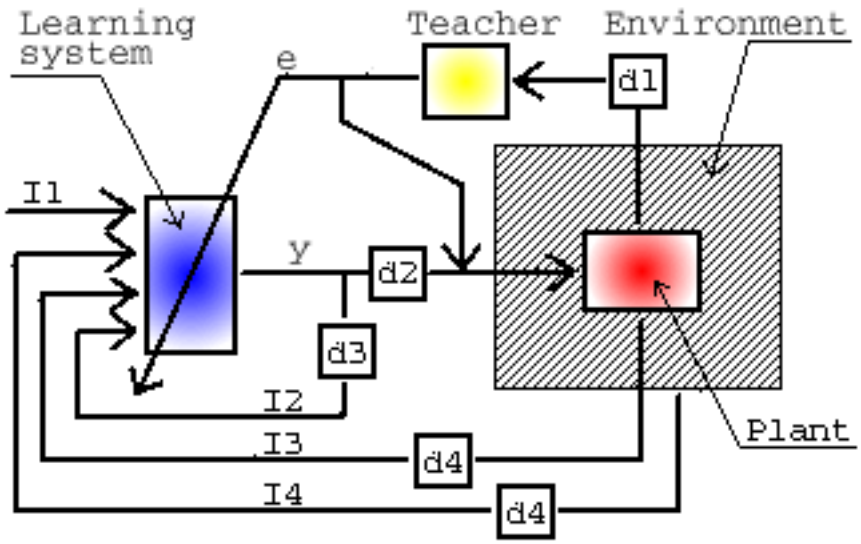


Figure 2.1: Schematic view of the cerebellar model. e denotes the error signal, dn the possible points of delay and In the inputs to the controller with $n \in \mathbb{N}^+$, y represents the output of the system. Figure adapted from [Joensuu, 2006]

Strictly speaking the plant and the environment are not part of the model of the cerebellum, however it is useful to include them in our discussion of the model in light of their explanatory properties and to provide a useful context in which the reader should place the theory. The plant, as previously explained, is the dynamic system which needs to be controlled. It accepts input from the controller and is influenced by and influences the surrounding environment. Furthermore it outputs information to the teacher and to the adaptive controller. This can be seen as proprioceptive feedback. Based on this information the teacher initiates a reflex and the controller incorporates knowledge in future actions.

The teacher is a relatively easy part of the model. It only receives input from the plant and according to some hardwired rules it initiates a reflex. Depending on the desired complexity this reflex can be very simple, for example move to the left, or more complex as the situation requires. The reflex signal acts both on the plant and the controller. Acting on the plant provides the adaptive

controller with useful contextual information through the sensors of the plant and environment. The reflex arriving at the controller functions as the error signal and provides the basis of learning.

The last part, the adaptive controller, would be the actual model of the cerebellum in combination with the teacher. The adaptive controller accepts a multitude of inputs and generates an output signal in an arbitrary range. The output signal would, in cerebellar endowed creatures, be analogous to motor commands. The inputs to the controller are fed to a single logistic neuron after being weighted, it is these weights which are changed by the learning rule which makes the controller adaptive.

The adaptive controller receives four distinctly different inputs. As mentioned earlier it also takes the reflex as an input, this serves as the teaching signal. This is a very rough signal which more or less does the right thing. The reflex is explicitly not involved in the generation of the output of the controller in any direct sense of the word though it does act on the plant. It *only* serves to support learning in the model.

Beside the reflex it takes the output it generated earlier, thus creating a direct feedback loop. However this is not strictly necessary when there are no transmission delays, see [Joensuu, 2006]. Next it takes information from the environment, this could be anything. And last it takes 'proprioceptional' input from the to be controlled plant, think in this case of tactile or visual information from the plant itself. The last three inputs can be delayed, artificially or not, any number of milliseconds. This is not necessary for the correct operation but proves to show that the cerebellar model is able to handle these kind of delays. The fact that the reflex is not delayed stems from the fact that in the real cerebellum the generation of the reflex happens in the cerebellum alongside the controller thus imposing a negligible delay. Hence forth we will refer to inputs when meaning all inputs except the reflex.

The output o_k of the cerebellar model is generated using a simple hyperbolic tangent

$$o_k(t) = f_k(\sum i_k(t)w_{jk}) \tag{2.1}$$

where $i_k(t)$ is the input to the cerebellum at time step t and $f_k(x)$ is defined as

$$f_k(x) = \frac{1}{1 + e^{-ax}} \tag{2.2}$$

where a defines the slope of the function. The learning rule used by the cerebellum is a variant of the common *Delta rule*.

$$\Delta w_{jk}(t) = \eta r_k(t) l_j \tag{2.3}$$

In this η stands for a small positive number representing the learning rate, l_j stands for the leaky integrator on which will be returned shortly, k stands for the neuron, $r_k(t)$ stands for the reflex acting on the model. The reflex r replaces the expression giving the difference between actual and desired output which is usually found in the Delta rule.

In order to handle transmission delays and inertia so called 'eligibility traces' are introduced, see [Raymond and Lisberger, 1998] for a more detailed description along with a biological motivation. These traces try to tackle the problem of delayed transmissions and inertia. They do so by spreading inputs to the

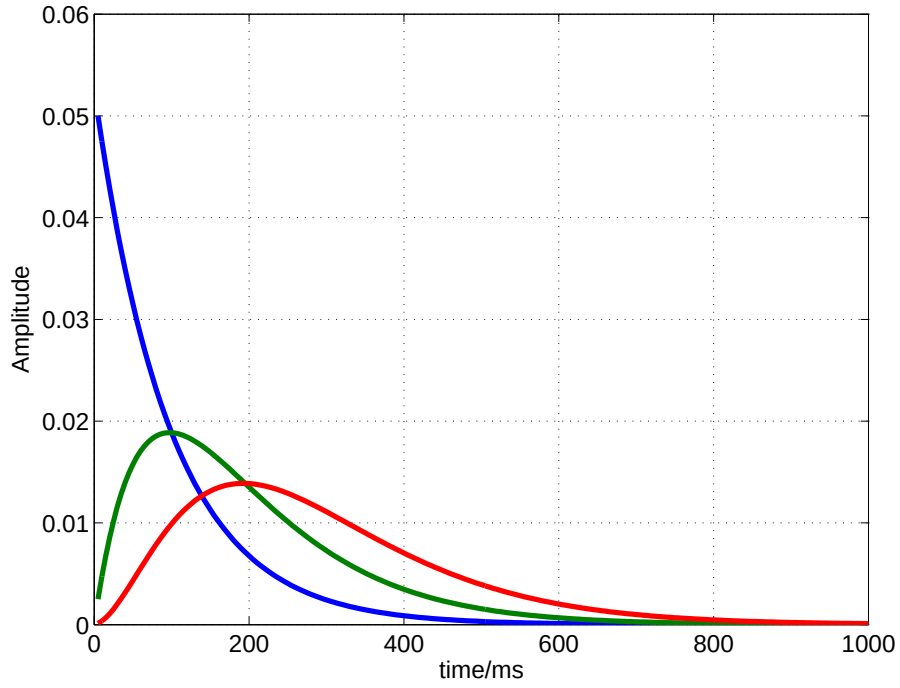


Figure 2.2: The effects of leaky integrators after a single pulsed input. The highest and steepest declining (in blue) a single integrator, second highest (in green) two integrators in series, lowest (in red) three in series. Adapted from [Joensuu, 2006].

cerebellum of time. These traces are implemented as, possibly cascaded, leaky integrators with the equation:

$$l_k(t+1) = (1-\alpha)x_k(t) + \alpha l_k(t) \tag{2.4}$$

in which $l_k(t)$ stands for the value of the leaky integrator k at time step t, $x_k(t)$ stands for the activation arriving at the integrator k at time step n and α is a constant governing the behaviour of the integrator. The higher this constant the more the integrator's activity is defined by past activations. In Figure 2.2 one can see the effect of one, two and three leaky integrators in cascade. For learning to converge there must still be some activation left in the relevant leaky integrators at the time the error signal arrives. To see this it is important to note that the learning rule is designed to enhance correlations between the error signal and the inputs relevant to it. This ensures that when there is still some activation left in the integrators the learning rule will make sure that if there is a correlation it will be found. When the signals are severely out of sync with the reflex conversion can take a long time.

Summarising the section the procedural calls involved in the model are shown. The following pseudo code shows how the model operates during runtime. For the teacher:

Data: Proprioceptive inputs
Result: Determined reflex
initialisation;
while *true* **do**
  receive inputs;
  generate reflex $r_k(t)$;
  apply reflex to plant;
  sent reflex to controller;
end

Algorithm 1: Algorithm for the teacher.

And for the controller:

Data: Proprioceptive inputs, reflex
Result: Converged controller
initialisation;
while *true* **do**
  receive inputs;
  receive reflex $r_k(t)$;
  if *reflex != null* **then**
    update weights w_k;
  end
  calculate force $o_k(x)$;
  apply force to plant;
end

Algorithm 2: Algorithm for the controller.

As can be seen the whole process is simple and easy to understand. This model has been shown in [Joensuu, 2006] to learn to balance a simple robot quite fast.

Chapter 3

Preprocessing techniques

This chapter is dedicated to several techniques which are used to process some of the input data which the cerebellar model uses.

3.1 k-Means Clustering

Often it is necessary or desirable to categorise data in discrete groups. The cerebellum is an excellent example as it needs processed inputs. k-means clustering does exactly that. It creates k non overlapping groups or clusters from the elements in an arbitrary data set. Intuitively one can say that the algorithm tries to minimise the distance of the elements inside the cluster and tries to maximise the distance to the other clusters. It does so in an iterative way.

What follows is a batch version of the k-means algorithm. First the algorithm constructs k random clusters by random division of the data samples. From these clusters the *centroids* are calculated. A centroid is the mean of a cluster's elements. Usually for cluster k a simple mean is used like

$$c_k = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{3.1}$$

where N stands for the number of elements in the cluster. For multi dimensional vectors an element-wise mean can be used. There are, however, many different methods to calculate a centroid of a data set and it depends on the data which type of mean is most useful.

The next step entails the calculation of the *distance* of each element in the data collection with each centroid. The distance is simply the difference between the centroid and the data element. If an element is not in the cluster which minimises the distance the element is reassigned to the cluster which does.

Then update the centroids of the clusters by recalculating the mean. Repeat this process until convergence or till an arbitrary stopping condition is reached, for example, after n runs or a minimal distance is obtained.

The end result is a situation in which each element in the data set is categorised in one of the k clusters, each cluster contains data elements which are in some way more similar to each other than to every other element not in that particular cluster. Categorisations for a new data sample can be obtained by

comparing the new data sample with the k different centroids and categorising the new sample to the category with which the distance to the relevant centroid is smallest. From this it is easy to imagine an online version of k-means clustering. After classifying the new sample to a certain class and thus a certain cluster, the relevant centroid of that cluster can be updated based on the old cluster elements and the newly added sample.

One of the great disadvantages of the k-means algorithm is its sensitivity to outliers in the data. The resulting centroids with an simple averaging calculation can differ vastly if only one outlier is present. To better explain this consider the following two dimensional data set, Table 3.1. This data is shown in Figure

n	x	y
1	1	1
2	5	5
3	5	7
4	6	7
5	7	6
6	2	9
7	2	11
8	3	10

Table 3.1: Example data

3.1. As one can see element 1 is obviously a statistical anomaly but the resulting clusters shown as convex shapes in the figure do radically alter shape. The best categorisation, or description if you will, of the data is not achieved. We will address this problem in the following section.

3.2 Self-Organising Map

A common problem with learning or the representation of data is its dimensionality. More often than not this is too high to properly deal with. Take for example a power plant. At any given time there are innumerable sensor readings. You can capture this data at a given moment ending up with a vector the size of all sensors and parameters combined. If you take measurements at regular intervals(and why would you not do so if you are to use it?) the amount of data collected is potentially huge. Huge data collections is something we, in principle, can handle, think of large databases. But this particular data has a problem. It is unstructured to the naked eye and thus makes it very hard to perform basic operations on this data. For example picking the most relevant(important) element, discerning relations between values or determine whether or not an entry is a statistical anomaly or outlier, is difficult.

In unstructured data there is no easy way for either human or algorithm to determine which entry is relevant or related, to either the problem at hand or other entries. *Self-organising maps(SOMs)* [Kohonen, 1998] help to solve the problem of high dimensionality by assigning a lower dimensional mapping to a high dimensional data sample and revealing relevant data. The self-organising map is also known as a *self-organising feature map(SOFM)*.

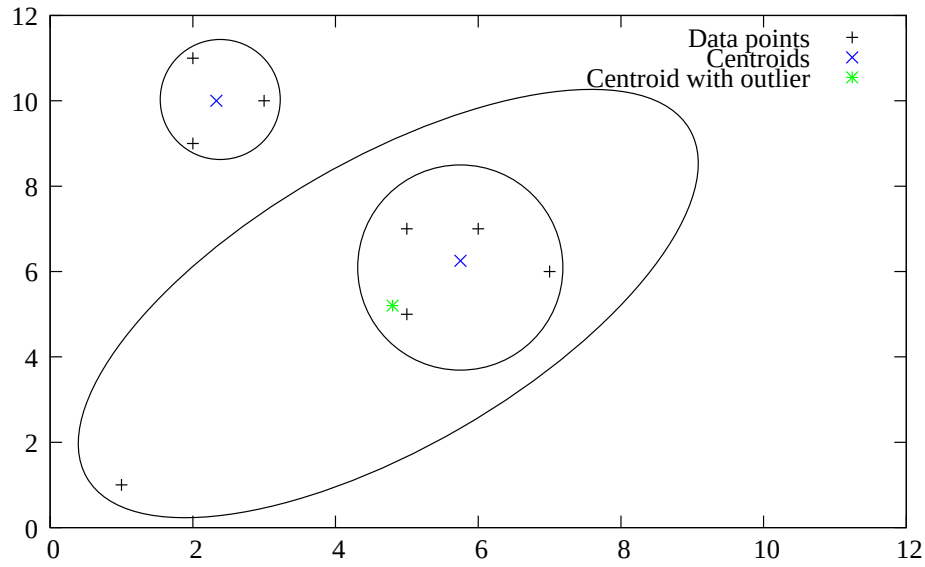
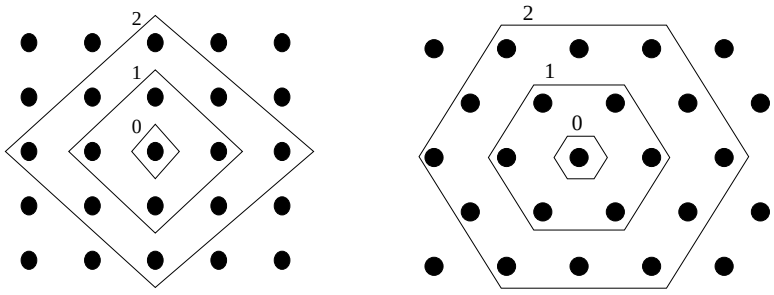


Figure 3.1: Data plotted from Table 3.1 plus centroids. The cluster which results from the inclusion of the outlier is obviously a bad representation of the data.

A SOM is a collection of neurons organised in an ordered topological fashion. The number of neurons can range from a few dozen to thousands. Because of the topological ordering each neuron has a fixed 'distance' to each other neuron. This becomes important in the learning and visualisation process which will be described in a moment. Every neuron has a weight vector (also known as *prototype* vector, *model* or codebook vector) equal in length as the input(training) vectors denoted $m_i = [m_{i1}, \ldots, m_{id}]$ where d is the length of the input vector. These are initialised randomly, in the more sophisticated SOM implementations the weight vectors are initialised as a regular array of vectorial values that lie in the subspace spanned by the two largest principal components of the training data, see [Kohonen, 2001]. This makes the computations far faster since an ordering is already imposed on the map and a smaller neighbourhood function and learning rate can be chosen. In essence each neuron performs a mapping from a high dimensional input to a low dimensional output without losing any relevant information.

The most common topological ordering of the output vectors is in a grid-like manner, if this is the case this specific variant is called a Kohonen Map after its Finnish inventor Teuvo Kohonen. Figure 3.2 shows a square and hexagonal example. Three dimensional cubes or other shapes can also be used but are less common. Because of this topological ordering it is possible to define a neighbourhood for each neuron. The neighbourhood influences which weight vectors are changed during learning.

During training a winner-takes-all approach is followed. Each iteration a random training sample is selected from all available data. The selected sample is compared to each weight vector of each neuron on the SOM. The comparison can be, and is often, as simple as calculating an Eucledian distance. Other



(a) Rectangular map (b) Hexoganal map

Figure 3.2: A square and hexoganal map together with a discrete neighbourhood. The numbering denotes the distance to the centre node.

distance measures can also be used depending on the data. From these comparisons the closest match is selected, called the *Best Matching Unit(BMU)*. Determining the BMU using Euclidean distance is expressed in the following equation in which the BMU is denoted as weight vector m with the subscript c.

$$\|x - m_c\| = \min_i \{\|x - m_i\|\}$$

Here i stands for the number of neurons and x is the selected sample. In the remainder of the equations of this section the following notation will be used to lighten the reading

$$c(x) = \arg\min_i \{\|x - m_i\|\} \tag{3.2}$$

The weights of the BMU and its neighbourhood are then adjusted to better match the input vector according to

$$m_i(t+1) = m_i(t) + \alpha(t) h_{c(x)i}(t)(x(t) - m_i(t)) \tag{3.3}$$

Here an important aspect of the SOM is introduced, note the function $h_{c(x)i}(t)$ which stands for the *neighbourhood* function of neuron i with respect to the BMU c of sample x and which takes time t as argument to implement a possible decrease in the width of the function h. This decrease in width has been shown to obtain better convergence results [Kohonen, 2001]. This function defines the topological neighbourhood of the BMU. It ensures that similar vectors are grouped together thus making visualisations and discovering of relationships in data easier by structuring the output data. The neighbourhood function can come in many shapes, although the most common function is a Gaussian.

$$h_{c(x)i}(t) = e^{\left(-\frac{\|r_i - r_c\|^2}{2\sigma(t)^2}\right)} \tag{3.4}$$

In which $\sigma(t)$ decreases monotonically in time and $r_i - r_c$ is the distance between the current neuron and the BMU on the topological map. Figure 3.3 shows several other neighbourhood functions commonly used.

After training the resulting map can be used to visually inspect the data to infer relations between the data. The map can also be used to cast an input to a lower dimension and be used to classify new inputs by presenting them to the map and see where they are placed on in the output space.

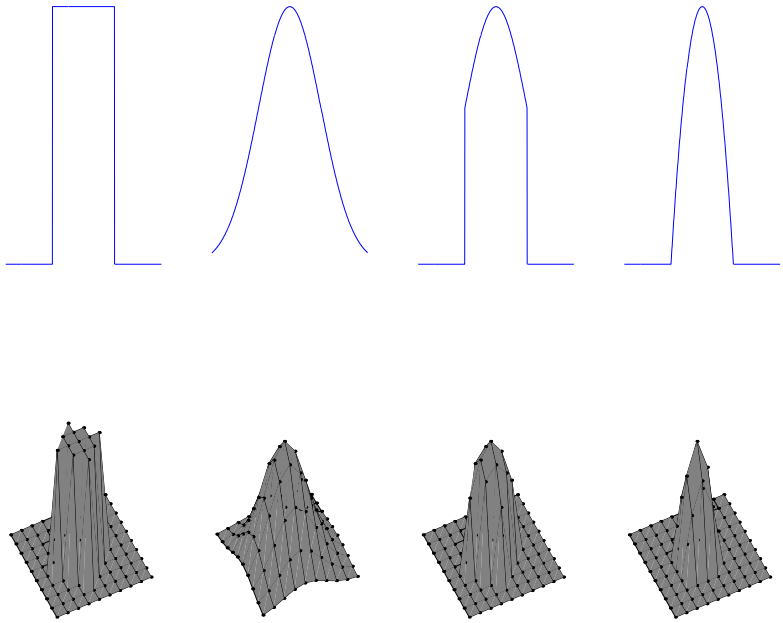


Figure 3.3: Four different neighbourhood functions.

It is important to note that the SOM algorithm presented above is online. Each time a new element is presented to the SOM which results in a slight change in the weights of the BMU and its neighbours. A SOM can also be implemented as a batch algorithm [Kohonen, 1992]. In this case the prototype vectors are initialised in the usual way, either randomly or based on the eigenvectors. Then for each and every sample the neighbourhood function and thus the BMU is determined as before in 3.2. This yields for each neuron i on the map a distance $h_{c(j)}$ to the BMU of each sample. After that instead of using 3.3 the following is used for each sample x_j where j is the number of samples available

$$m_i = \frac{\sum_j h_{c(j)i}(t)x_j}{\sum_j h_{c(j)i}(t)} \tag{3.5}$$

In words, each prototype or weight vector has associated with it a collection distances. In fact it has j of these values, one for each data sample as defined in 3.4. Multiplying these values with the associated sample x_j and summing this yields the total 'strength' of the weight vector. This now only needs to be normalised by dividing it by the total value of the neighbourhood function. Normalising each vector this way makes sure that the relative strengths between weight vectors are equal. This way of looking at it departs *from* each weight vector instead of working from a presented data sample *to* each weight vector. This process can be repeated until convergence or until a preset condition. This definition will prove useful in the next section.

In summary the SOM is most commonly used for visual inspection of data sets and to provide the necessary dimensionality reduction. An additional advantage is the property that the SOM constructs prototype vectors of the input data. This averaging mechanism is useful to filter out noise in the original data and to again reduce the number of elements to describe the data set.

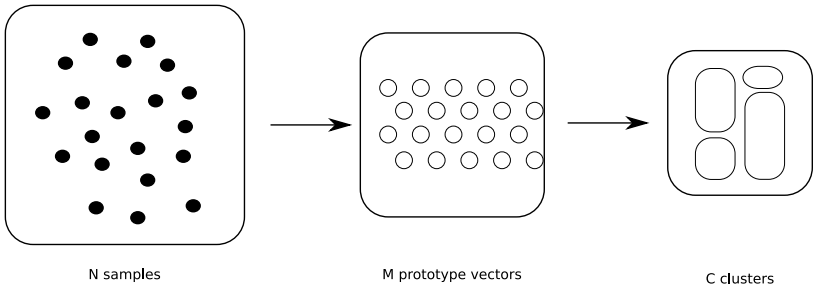


Figure 3.4: A schematic overview of the merger between a SOM and k-means

3.2.1 Merging SOM and k-means

One can imagine using a k-means algorithm on a raw data set. And in fact a SOM is a k-means clustering with k equal to the number of neurons on the map and a neighbourhood function which is one on the BMU and zero elsewhere. This can easily be seen when showing the centroid update rule of the batch version of k-means 3.1 next to the weight update rule of the batch version of the SOM 3.5.

$$\begin{aligned} m_i &= \frac{\sum_j h_{c(j)i}(t) x_j}{\sum_j h_{c(j)i}(t)} \\ c_k &= \frac{1}{N} \sum_{i=1}^{N} x_i \end{aligned}$$

When obeying the previously stated restrictions on the neighbourhood function and number of neurons, these two functions amount to the same thing.

Clustering on an already converged SOM has several advantages. Most important is noise reduction. The SOM's neurons act like local averages. Using these averages makes the clustering less sensitive to random variances in the original data. In Section 3.1 we saw the great adverse effect of outliers on the performance of the clustering. Moreover the computational load is shown to be significantly reduced by the use of a two-layered approach. A clustering algorithm needs only to cluster the prototype vectors instead of the raw data points, see[Vesanto and Alhoniemi, 2000] and [Lampinen, 1992].

In Figure 3.4 the two-layered system is schematically shown which utilises the converged SOM for clustering.

3.3 Principal Component Analysis

This section describes principal component analysis(PCA) [Smith, 2002], [Hyvärinen et al., 2001], [Jackson, 1991]. Also known as Karhunen-Loève transform or Hotelling transform [Hotelling, 1933]. PCA is a statistical technique often used for analysing high dimensional data, pattern analysis or data compression. It identifies patterns in data in a way which highlights the differences and similarities. Strictly speaking PCA is a linear transformation to a new coordinate system where the axis are perpendicular to each other and ordered in such a way that the greatest variance in the data set is projected on the

first axis, the second greatest variance on the second axis and so on. In other words it decomposes the original random variables in a set variables which are uncorrelated, this maximises variance per variable.

The procedure is divided in several steps and will be describe here. The last couple of steps are not required but serve to complete the picture as PCA describes it. Accompanying the following steps is an example which will demonstrate the procedures in order to make the explanation more concrete and insightful.

Data collection This is a straightforward step and serves more to bootstrap our example. In principle data can be n dimensional. For the example, however, is chosen for two dimensional data to facilitate its easy display, see Table 3.2. In this table the generated data is shown along side the same data with the means removed. Why this is important is shown in the next step.

x	y	x	y	x	y
0.2853	0.2089	-0.7728	-0.3002	0.3018	0.1718
0.4314	0.1756	-0.6267	-0.3335	0.4113	0.2207
0.5271	0.2479	-0.531	-0.2612	0.518	0.2683
0.7727	0.2661	-0.2854	-0.243	0.7296	0.3626
0.5234	0.4493	-0.5346	-0.0598	0.5899	0.3003
0.9154	0.4674	-0.1427	-0.0418	0.9235	0.4491
1.0253	0.4966	-0.0328	-0.0125	1.0261	0.4948
0.9714	0.4996	-0.0866	-0.0096	0.9822	0.4753
1.0989	0.4517	0.0408	-0.0574	1.0708	0.5148
1.385	0.515	0.3269	0.0059	1.3329	0.6317
1.3844	0.7252	0.3264	0.2161	1.4107	0.6664
1.5299	0.7454	0.4719	0.2363	1.5395	0.7238
1.6059	0.6575	0.5479	0.1484	1.5702	0.7375
1.5432	0.7984	0.4852	0.2893	1.5704	0.7376
1.8715	0.9319	0.8135	0.4228	1.8939	0.8819

Table 3.2: Left random generated data, in the middle the same data with the mean subtracted and on the right the reconstructed data leaving out the least significant variable.

Calculate the covariance matrix *Variance* defines the amount of spread in the data. It is a measure telling something about the average deviation from the mean of the data and is closely related to the standard deviation. Variance is defined as

$$var(X) = \frac{\sum_{i=1}^{n}(X_i - \widehat{X})^2}{(n-1)} \tag{3.6}$$

In which n is the number of data samples, X the random variable in question and $\widehat{X}$ the mean of that variable. And although this measure proves to be very useful in PCA we are more interested in how variables vary (linearly) with respect to each other. So the natural extension of variance is the *covariance*

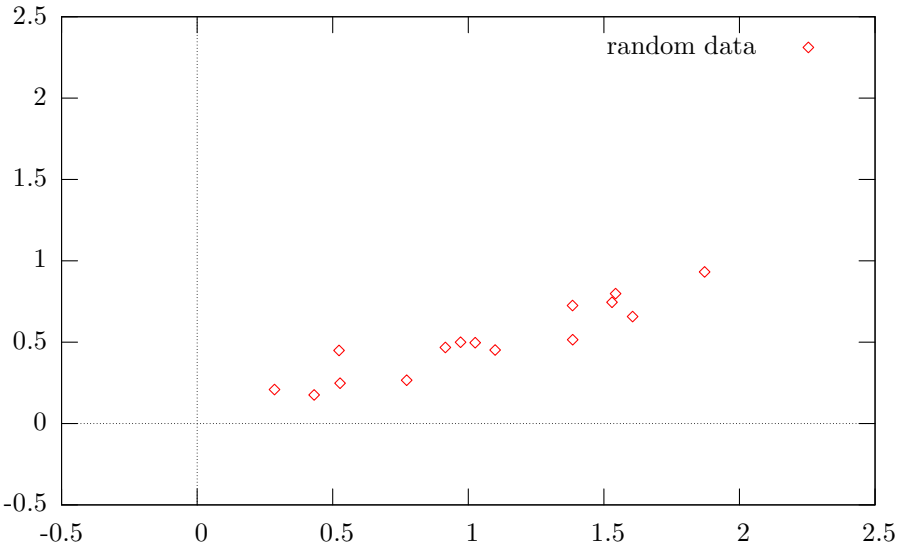


Figure 3.5: The generated data plotted

which looks how two different variables behave together.

$$cov(X,Y) = \frac{\sum_{i=1}^{n}(X_i - \widehat{X})(Y_i - \widehat{Y})}{(n-1)} \tag{3.7}$$

As can easily be seen $cov(X,X)$ equals $var(X)$. The sign of the covariance tells us something about how the variables behave. If the covariance is positive then if one variable increases the other does so too. And the other way around is true as well. If the covariance is negative then when one variable decreases the other increases. In other words whenever there is covariance between two variables there is a linear relationship between the two. The size of the covariance tells how strong this relation is. When the covariance is zero between two variables they exert no influence on each other and thus there exists no linear relation between the two. This is exactly the situation for which PCA works. Linear dependant variables contain information about each other, when one moves the other does. There is an inherent redundancy between those variables which could be removed in order to discern 'true' patterns in the data and not the ones induced by a linear relationship. Moreover, whenever this is the case variables can be omitted without influencing the remaining variables.

It is often the case that while inspecting data it is not immediately clear what to look for. Inspecting the *covariance matrix* gives in a glance all covariances(and thus variances) of all variables in the data set. A covariance matrix of n variables is constructed as follows:

$$\mathbf{\Sigma} = \begin{pmatrix} cov(X_1,X_1) & cov(X_1,X_2) & \dots & cov(X_1,X_n) \\ cov(X_2,X_1) & cov(X_2,X_2) & \dots & cov(X_2,X_n) \\ \vdots & \vdots & \ddots & \vdots \\ cov(X_n,X_1) & cov(X_n,X_2) & \dots & cov(X_n,X_n) \end{pmatrix}$$

The matrix Σ is symmetric in the diagonal and on this diagonal are the variances of individual variables.

Going back to the example the covariance matrix looks like:

$$\mathbf{\Sigma} = \begin{pmatrix} 0.2359 & 0.1025 \\ 0.1025 & 0.0518 \end{pmatrix} \tag{3.8}$$

A note of caution, while reading PCA literature [Smith, 2002] one often runs in to another step preceding the computation of the covariance matrix. In this additional step the empirical mean is removed from the data and only then the covariance matrix is computed. However in 3.7 the means are already subtracted from the data set. The confusion is caused by an alternative definition of covariance matrix.

$$\Sigma = \frac{\mathbf{X}\mathbf{X}^T}{n-1} \tag{3.9}$$

This does yield the same result obtained by 3.7 but only if the empirical means is subtracted.

Calculate the Eigenvectors and Eigenvalues The next step in the determination of the principal components is the calculation of the *eigenvector* and *eigenvalue* pairs of the covariance matrix. The covariance matrix is then used as a transformation matrix. Eigenvector and eigenvalue pairs tell something about the dynamics of a given matrix transformation. Almost all vectors change direction when that given transformation is applied on them. Eigenvectors do not change direction. The corresponding eigenvalue gives the change in scale the eigenvector undergoes given the transformation.

In short the eigenvectors define the principal directions of variation and the corresponding eigenvalues tell something on how strong the variance is in each new dimension. The higher the eigenvalue the more variance of the data is explained with the corresponding eigenvector. These eigenvectors are then taken to be the new axes. The remainder of this paragraph will explain why these results hold and how to obtain them.

Finding eigenvalues and eigenvectors is done by solving

$$\mathbf{A}\boldsymbol{v}_\lambda = \lambda \boldsymbol{v}_\lambda \tag{3.10}$$

In which $\mathbf{A}$ is the transformation matrix, $\boldsymbol{v}_\lambda$ the eigenvector and λ the corresponding eigenvalue. Solving this equation analytically is relatively easy for a small, and most notably square singular matrix. In this case dimensionality should not be larger than three. Solving this involves finding the determinant of the transformation matrix $\mathbf{A}$ minus the eigenvalues λ times the identity matrix and equating this to zero.

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0 \tag{3.11}$$

These eigenvalues λ can then be used to find the eigenvectors by solving

$$(\mathbf{A} - \lambda\mathbf{I})\boldsymbol{v}_\lambda = 0 \tag{3.12}$$

which is equal to 3.10, see [Strang, 1998]. If the dimensionality of the matrix is larger or nonsingular, then an iterative approach can be taken which will not be described here. Mathematical environments like Matlab[1], Octave[2]

[1] `http://www.mathworks.com`

[2] `http://www.octave.org`

and Mathematica[3] all include functions to iteratively calculate eigenvectors and eigenvalues.

This process of finding the eigenvalues and eigenvectors will be demonstrated through the example presented earlier. Using the covariance matrix 3.8 as calculated in the previous section as transformation matrix and equation 3.11, first subtract the eigenvalues times the identity matrix

$$\mathbf{A} - \lambda\mathbf{I} = \begin{pmatrix} 0.2359 - \lambda & 0.1025 \\ 0.1025 & 0.0518 - \lambda \end{pmatrix} \tag{3.13}$$

then take the determinant of this matrix with $ad - bc$.

$$\begin{aligned} \begin{pmatrix} 0.2359 - \lambda & 0.1025 \\ 0.1025 & 0.0518 - \lambda \end{pmatrix} &= (0.2359 - \lambda)(0.0518 - \lambda) - (0.1025)(0.1025) \\ &= 0.0017 - 0.2877\lambda + \lambda^2 \end{aligned}$$

solving this equation yields two eigenvalues $\lambda_1 = 0.2817$ and $\lambda_2 = 0.006$. Then, using Equation 3.12, we obtain the eigenvectors $\boldsymbol{v}_{\lambda_n}$ for each of the n eigenvalues. These eigenvectors are congregated in matrix $\mathbf{V}$ containing all eigenvectors as columns

$$\mathbf{V} = \begin{pmatrix} -0.9133 & 0.4073 \\ -0.4073 & -0.9133 \end{pmatrix} \tag{3.14}$$

here λ_1 is associated with the first column and λ_2 with the second. This can be demonstrated as follows. A matrix can be composed from a multiplication of its eigenvectors and eigenvalues

$$\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{-1}$$

Because V is an orthonormal matrix this equation can be rewritten as

$$\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^{T} \tag{3.15}$$

in which Λ stands for the diagonal matrix containing the eigenvalues. This notation is convention when considering eigenvalues and eigenvectors and this thesis will follow that convention. Equation 3.15 comes from the following. From 3.10

$$\mathbf{AV} = \mathbf{A}\begin{pmatrix} \boldsymbol{v}_{\lambda_1} & \cdots & \boldsymbol{v}_{\lambda_n} \end{pmatrix} = \begin{pmatrix} \lambda_1\boldsymbol{v}_{\lambda_1} & \cdots & \lambda_n\boldsymbol{v}_{\lambda_n} \end{pmatrix} \tag{3.16}$$

The trick is to split the matrix $\mathbf{AV}$ into $\mathbf{V}$ times $\mathbf{\Lambda}$

$$\begin{pmatrix} \lambda_1\boldsymbol{v}_{\lambda_1} & \cdots & \lambda_n\boldsymbol{v}_{\lambda_n} \end{pmatrix} = \begin{pmatrix} \boldsymbol{v}_{\lambda_1} & \cdots & \boldsymbol{v}_{\lambda_n} \end{pmatrix} \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix} = \mathbf{V\Lambda} \tag{3.17}$$

Now $\mathbf{AV} = \mathbf{V\Lambda}$ holds which can be rewritten as shown in 3.15. This shows the relation between the eigenvalues and eigenvectors.

To see what the calculation of the eigenvectors actually means Figure 3.6 shows the sample data with its means subtracted with the two eigenvectors overlain on top. What can be clearly seen in this figure is that the two eigenvectors

[3] `http://www.wolfram.com/products/mathematica`

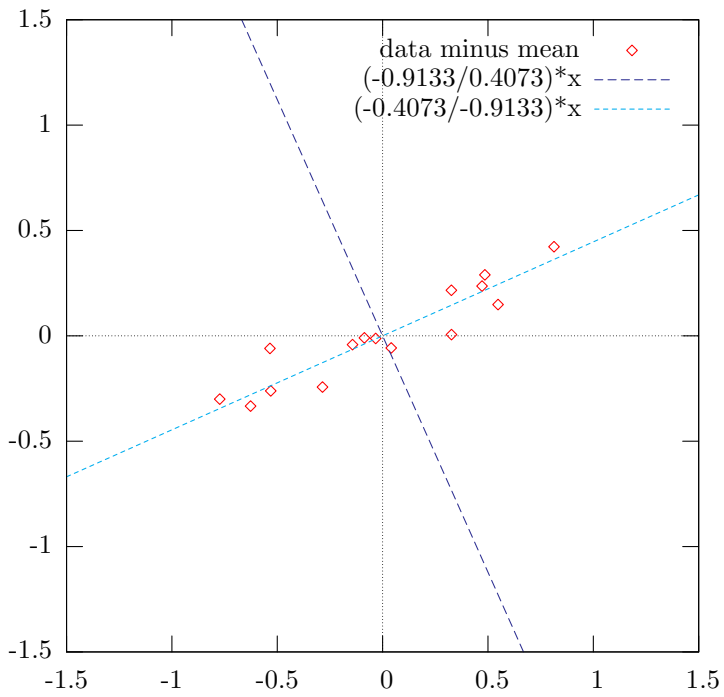


Figure 3.6: The mean generated data plotted together with the two eigenvectors.

define a new system of axis in which the greatest variance is along the principal axis. The principal axis is the one with the greatest eigenvalue, which is in this case the first one and thus corresponds to column one in 3.14.

To understand why the first axis is the one with the greatest variance consider some n dimensional data collection $\mathbf{D}$ with zero means and the covariance matrix $\mathbf{\Sigma}_D$. The data can be projected on an arbitrary (column) vector $\boldsymbol{z}$ yielding the transformed collection $\mathbf{D'}$ via $\mathbf{D'} = \boldsymbol{z}^T\mathbf{D}$. The covariance matrix of that new collection is then given by

$$\mathbf{\Sigma}_{D'} = \boldsymbol{z}^T\mathbf{\Sigma}_D\boldsymbol{z} \tag{3.18}$$

This result follows from the definition of the covariance 3.9 without the means

$$cov(\boldsymbol{z}^T\mathbf{D}_i, \boldsymbol{z}^T\mathbf{D}_j) = \frac{\boldsymbol{z}^T\mathbf{D}_i(\boldsymbol{z}^T\mathbf{D}_j)^T}{n-1} = \boldsymbol{z}^T\frac{\mathbf{D}_i\mathbf{D}_j^T}{n-1}\boldsymbol{z}$$

Note that the right hand side of the equation includes the original covariance matrix. Now take $\boldsymbol{z}$ to be the eigenvector matrix and 3.15 it follows that $\mathbf{V}^T\mathbf{\Sigma}\mathbf{V} = \mathbf{\Lambda}$. This shows that when rotating the data with the eigenvectors as rotation matrix the new covariance matrix is equal to the eigenvalue matrix of the original data.

The remainder of the steps all involve the use of these results in terms of data compression and reconstruction and are not required but serve to complete the full view of PCA.

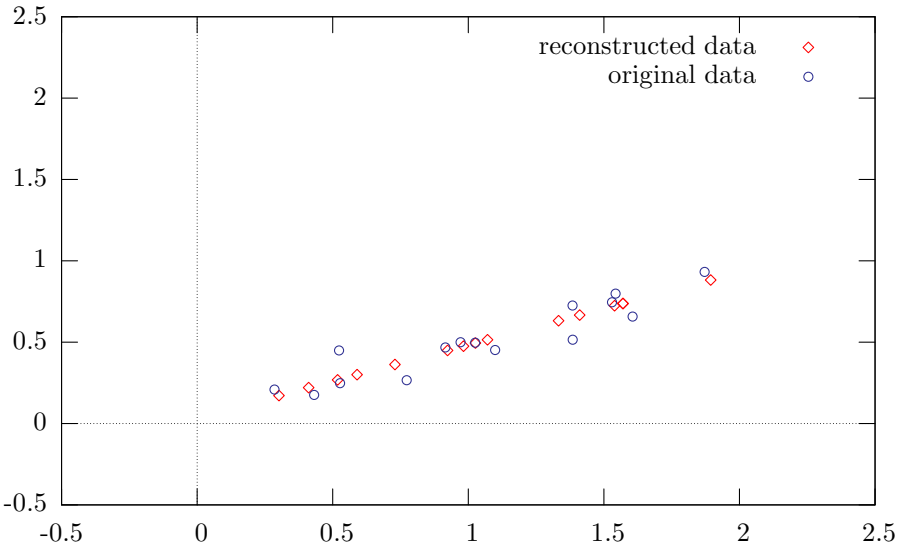


Figure 3.7: The reconstruction of the original data without the least significant component together with the original data.

Reducing the dimensionality On inspection of the eigenvalues in the previous section it can be observed that the values differ quite a lot. In fact by far the most variance is explained by the eigenvectors with the largest eigenvalue (that is the first). This is the most important relation in the example data. So if the other component is dropped the original data can still be explained in a sufficiently high degree(this of course depends on what 'significant' exactly means). This results in a data set which has a lower dimensionality than the original data. To demonstrate consider again the example. From 3.14 only the first column will be used to construct an approximation of the original data.

$$\mathbf{Y} = \mathbf{X}\boldsymbol{v}_{\lambda_1} \tag{3.19}$$

In which $\mathbf{X}$ stands for the initial data set and $\mathbf{Y}$ for the lower dimensional form. In other words the initial data set is now expressed with only one new variable but which still contains much of the information. The result is shown on the right in Table 3.2. Checking the reconstruction is easy by calculating the covariance matrix of $\mathbf{Y}$ computed with the full eigenvector matrix, this should yield a matrix with all zeros off the diagonal. Each new variable is uncorrelated with the others.

When reconstructing the data to the original coordinate system another transform is required (note this is for the example, for the full reconstruction one needs to use the full eigenvector matrix)

$$\mathbf{X}_{\text{reconstructed}} = \boldsymbol{v}_{\lambda_1}^T \mathbf{Y} \tag{3.20}$$

The result is plotted in Figure 3.7.

Chapter 4

Proofs of principle

This chapter will be dedicated to experiments showing the usefulness of GATE, Chapter 5, and showcasing the versatility of the cerebellar model as it is described in Chapter 2. The following experiments are all performed with a system combining GATE and the cerebellar model. Some additional preprocessing steps will be added using the techniques described earlier. Every experiment is based on the work of [Joensuu, 2006] and thus entails the balancing of an unstable robot, specifically the experiments without transmission delays.

4.1 General inputs, reflex and robotic shape

This section will set the stage for the experiments to come. First the basic shape of the used robot is shown followed by a description of the common inputs used in both experiments. Finally the construction of the teacher reflex will be explained.

4.1.1 Robotic shape

The general body plan of the robot is consistent throughout the experiments and is shown schematically in Figure 4.1. The task of the robot is always the same, it has to keep itself from falling over. As can be seen from the picture the robot has an elongated form making it inherently unstable. The robot has to move back and forth in order to keep its balance much in the same way a unicycler would do. The experiments in this chapter all extend on this theme.

In the first experiment the general shape of the balancing robot is preserved as it was presented in [Joensuu, 2006]. In the second experiment there are some significant changes. Foremost is the addition of a camera to the body plan. This camera added to the robot enables the robot to receive images from the front of the robot. This has also direct consequences for the environment the robot is situated in. For the system to develop in reasonable fashion a sufficiently challenging environment is required. This explains the presence of the plane with the forest texture as seen in Figure 4.8.

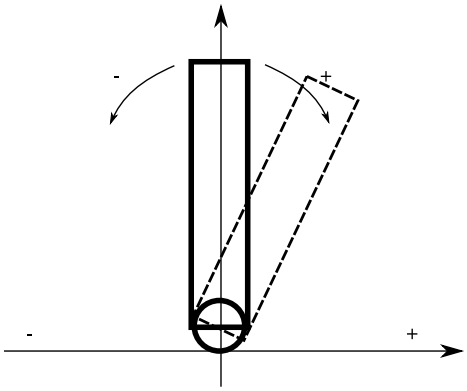


Figure 4.1: A schematic representation of the robot. On the horizontal axis the negative and positive direction is depicted. The same is done for the angle. The dashed silhouette indicates a robot out of balance.

4.1.2 Inputs

As in the earlier incarnation of the pole balancing experiment the cerebellum was provided with information on the location, the speed and the angle of the robot. Additional to these there are now also inputs coding for angular velocity. All four are, after scaling between minus one and one, encoded in a series of Gaussian functions. Angle and angular speed are encoded in 31 functions, location and speed both in 11. This decision was made out of conviction that the latter two variables were less important.

A set of these Gaussian functions are shown in Figure 4.2 in which, as an example, the angle is displayed on the horizontal axis and each response on the vertical axis. As one can see different input units respond different to different angles. This form of encoding is chosen to give the model an adequate range of inputs. For example, a small deviation in angle can be catastrophic when trying to maintain one's balance. When coding that information in one unit with a range between zero and one the value for which balance can be maintained might be in the range 0.49 and 0.51 making it unnecessary hard for the model. This encoding departs slightly from the encoding used in the original experiments in which only the angle was encoded in a series of Gaussian functions. In these experiments the other inputs might hold relevant information as well. Each function has a standard deviation of a constant 0.005 whether it is coding for angle, speed or any other type of input.

4.1.3 Reflex

The teacher or reflex part of the controller is a hierarchical system, see Figure 4.3. This hierarchical setup incorporates different aspects in the reflex allowing for an inherent guidance toward, for example, a certain position or speed of the robot. The particular reflex hierarchy used here did not incorporate the position as part of the system. When desired, however, adding location would come on top of the hierarchy.

Bottom up the hierarchy is explained as follows. Take A_m to be equal to zero. This means that the reflex is a scaled result from the desired angle A_d

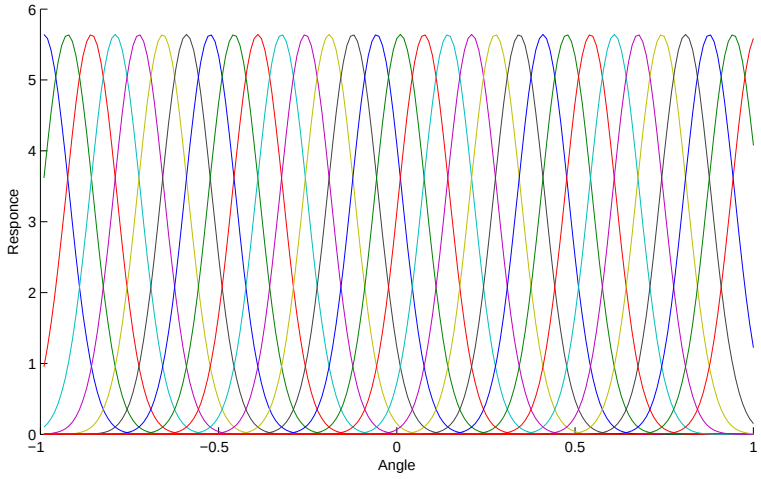


Figure 4.2: Each curve's maximum is associated with a different input value.

$$S_d - S_a = S$$
$$\downarrow \beta S = A_m$$
$$(A_d - A_m) - A_a = A$$
$$\downarrow \alpha A$$
$$R$$

Figure 4.3: The reflex hierarchy. S_d denotes the desired speed, S_a the actual speed. The same naming convention holds for A_d and A_a denoting desired and actual angle. A_m is the modifier applied to the desired angle.

minus the actual angle A_a. The desired angle, given the task, should be zero. Whenever the robot is leaning forward, shown in Figure 4.1 as the dotted silhouette, the reflex should be directed in the positive direction. This is accomplished whenever coefficient α is smaller than zero. For the next layer take A_d and A_a to be zero, the robot is in balance. Further take the actual speed S_a to be smaller than the desired speed S_d, so the result of $S_d - S_a$ is positive. The question one should ask is how this situation translates into a modification of the desired angle. If, as is the case here, the robot is moving too slow the desired angle should be more positive. In other words the term A_m should be negative. This can only be achieved if coefficient β is negative. The range in which α and β still yield a feasible reflex is very wide. Experiments have shown that α can be as small as -0.1 and as large as -55. At very low values convergence was not guaranteed in reasonable time and at very large values oscillations started to occur. The best results were obtained when using a value of -1. This makes sense as the value for α has a direct connection with the learning rate as this value scales the reflex and the reflex is responsible for the weight adjustments, as shown in 2.3. For β the range of tested values are smaller, -1 is the lowest tested value in which case convergence was not a problem albeit slow. Values as large as -15 still led to convergence however the resulting system was slightly unstable as the reflex is quite aggressive in obtaining the desired speed.

4.2 Irrelevant inputs

In this experiment it is shown that the cerebellar model is tolerant to noise and able to ignore disruptive inputs. Besides the normal inputs to the cerebellum a large number of inputs is added which will relay only noise. The cerebellum should learn to ignore these inputs and as a result the weights associated with these inputs should be (close to) zero. The experiment will take place in a simplified environment as already shown in Figure 1.1 for there is no need for the additional features in Figure 4.8. The learning rate used in this experiment is held at a constant 0.001, the length of the eligibility trace cascade was set at 3 with a slope parameter $a = 0.1$, one iteration is a 5 millisecond large step in the simulator.

4.2.1 The disruptive inputs

Besides the regular inputs, twenty additional inputs are added. Where twenty is an arbitrary number chosen because it is sufficiently large in comparison to the other inputs. These inputs will receive the noise. The noise fed to the cerebellum comes in two variants. In the first variant the noise is uniformly distributed between minus one and one and changed every iteration. In the second variant the input to the noise receiving units is one every iteration. This is done in order to investigate how the cerebellum holds in face of a large increase in the input energy. The simulation is run till convergence. Convergence is defined as being achieved when the weights have changed only very little (smaller than 5×10^{-5}) the last 1000 iterations and the robot has not fallen down during these iterations. The value of α in the reflex was set to -1 and the value of β was set to zero in effect turning this layer of the reflex off. In this particular experiment only the act of balancing is important not how this is achieved.

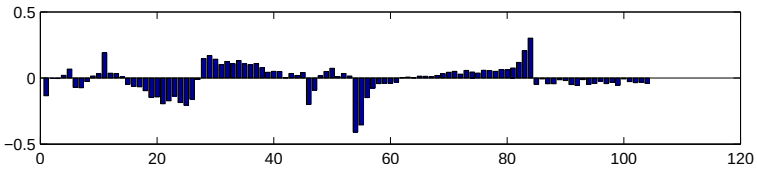


Figure 4.4: The resulting weights after convergence(5000 iterations) in a typical experiment run. Weight numbers 85 to 104 represent the weights associated with the noise inputs. In this variant of the experiment the weights were initialised randomly between 0 and 0.1 and the noise applied to the noise receiving units was constantly one.

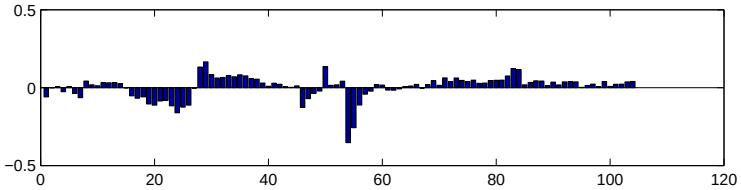


Figure 4.5: As Figure 4.4 but input is continuously random for the last input units.

4.2.2 Results

As can be seen from Figure 4.4 and 4.5 the weights associated with the disruptive inputs (units 85 to 104) indeed remain close to zero. Would the learning rate approach zero in the limit the weights would converge to zero. All the while the robot does learn to balance. Table 4.1 shows ten runs for each type of experiment together with the number of iterations needed for convergence and the number of times the robot fell over. For reference purposes ten runs without the twenty noise inputs are presented along side.

Figure 4.6 shows a typical example of how the changes in the weights progress during training, from a similar figure the convergence criteria were inferred. Along side this figure the output and reflex are shown. As can be seen from this figure the reflex is relatively small compared to the cerebellar output making the weight changes relatively small as well (see Equation 2.3). When choosing a higher value for α the reflex becomes stronger which might be desirable in cases where the speed of convergence is more important, however in these experiments fast convergence was not required and this approach leads to more stable results.

Initial results shown in Table 4.1 prompted further investigation as the results were not statistically significant but hinted at a performance *increase* for the experiment in which random noise was applied. In order to do so 30 additional runs were performed for each experiment and the benchmark without the noise. In doing so increasing the sample size fourfold. These results are shown in Table 4.2.

The extended results from the two variants of the experiment show that the model has no difficulty with the additional noise, but that random noise does not improve performance. Only in the experiment in which the cerebellum is pounded with a constant noise of one performance is significantly deteriorated (with a p-value of 0.0064 for the iterations and 0.0002 for the number of resets).

Constant noise		Random noise		No noise	
Iterations	Resets	Iterations	Resets	Iterations	Resets
4000	7	2400	5	3400	6
3600	5	2200	4	4400	7
2600	5	3000	5	2400	5
3400	7	2000	4	3800	6
2600	5	3200	6	3000	5
2400	5	2200	4	3000	5
3200	5	2400	5	2400	5
3200	6	4000	7	2800	5
3200	6	2400	5	2000	4
2400	5	2400	5	2800	5
Averages		Averages		Averages	
3060	5.6	2620	5.0	3000	5.3

Table 4.1: Convergence results for, left, constant input of one to the noise receiving inputs and, middle, for random input to these units. To the right the results for ten noiseless runs.

Constant noise		Random noise		No noise	
Averages		Averages		Averages	
Iterations	Resets	Iterations	Resets	Iterations	Resets
2990	5.45	2630	4.6	2570	4.3

Table 4.2: Average convergence results for the original results with the results 30 additional runs.

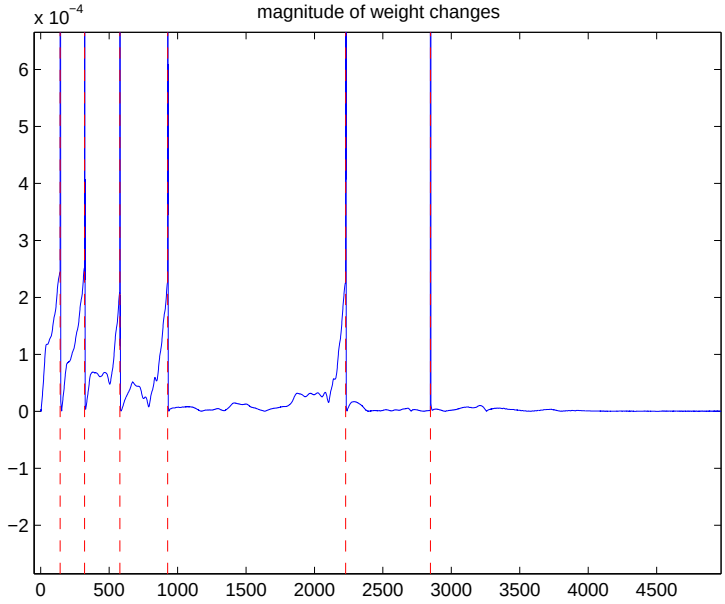


Figure 4.6: In this graph typical weight changes are plotted against time (the number of iterations). A vertical dashed bar(red) shows where the robot fell triggering a revert. Twenty additional inputs received random input, α was set to -1.

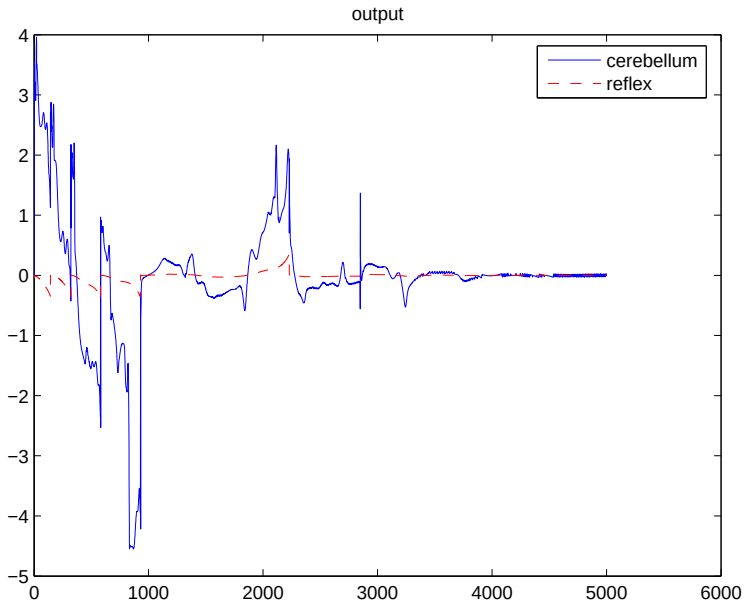


Figure 4.7: From the same experiment as in Figure 4.6. The dashed line(red) shows the output of the reflex, the other the output of the cerebellar model.

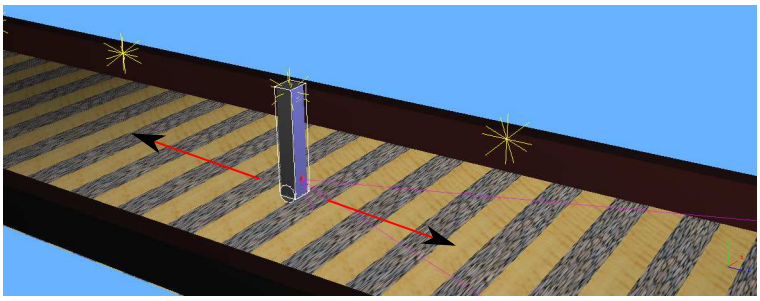


Figure 4.8: The robot in the environment as it is used in the texture experiment. The arrows on the ground (in red) depict the possible movements of the robot. The lines protruding from the small cylinder on the front of the robot is the viewing field of the camera.

In this case the model receives, on average, a higher amount of input. Especially when the weights have not yet converged to zero the output is thus significantly stronger and leads to more resets as to robot falls over more often. Naturally, from the definition of convergence, this leads to a higher required number of iterations to converge. As with random noise the average additional input is zero and thus does not lead to significant degradation of performance if any at all.

From the structure of the weight matrix the importance of certain inputs can be inferred. The units 12 to 42 from the inputs encode the angle from most negative(12) to most positive(42), for example, from Figure 4.4 one can see clearly where the angle turns positive (unit 27) and thus the cerebellar output should inverse its direction. In units 54 to 84 angular speed is encoded. Here it is clearly visible that when the angular velocity is high the cerebellum reacts strongly to that. When angular velocity is high something is amiss and strong intervention is deemed necessary.

In conclusion the cerebellar model handles noise very well. Even when a large proportion of the inputs are irrelevant the model quickly figures this out and reduces the associate weights to near zero. The energy added by the noise signal does have influence on the performance but this comes as no surprise.

4.3 An uneven world

The goal of this experiment is to demonstrate the capabilities of the cerebellum beyond the near trivial and perfect data supplied in the experiments done in [Joensuu, 2006]. In those experiments the inputs to the cerebellum were absolute and correct measures of, for example, the speed and angle of the robot.

4.3.1 Adaptations to the world and robot

In this experiment the robot is placed on an uneven terrain and is equipped with a camera. This robot is depicted in Figure 4.8 in which the direction of motion as well as the camera and camera frustum is shown. The task of the robot remains the same, the maintenance of balance. In this case however the robot

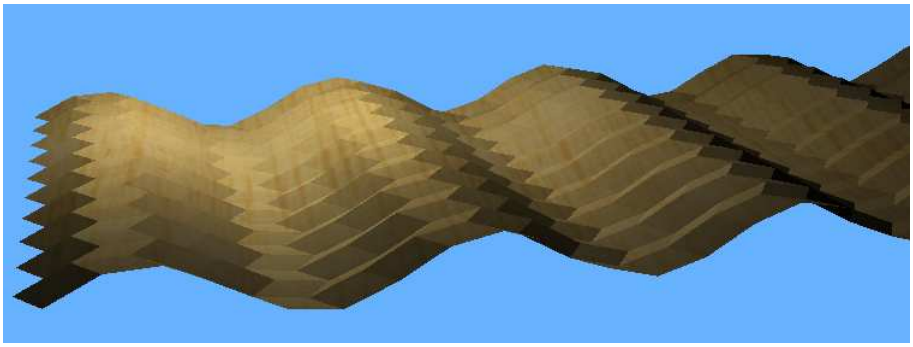


Figure 4.9: In this uneven world the robot needs to anticipate the bumps in order to remain balanced.

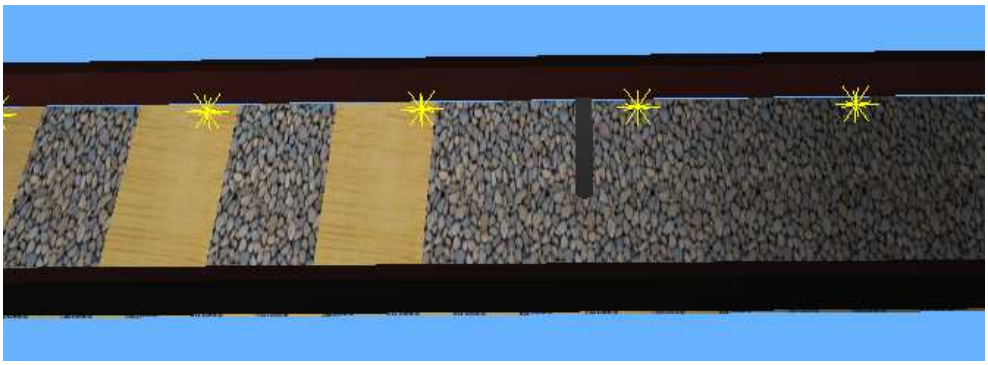


Figure 4.10: In this simulation the robot is induced to advance forward(to the left in the figure) thus making the robot traverse the different textures. One of the textures imposes an artificial drag on the robot.

is forced to move forward in order to have it traverse the uneven terrain. The camera is mounted on the front of the robot tilted slightly down so it can observe the terrain just in front of the robot. The robot will need the information from the camera in order to keep itself balanced.

After building the uneven terrain as depicted in Figure 4.9 the physics engine of the simulator proved unable to cope with the round wheels of the robot and a terrain build from triangular tiles. Appendix B elaborates on this problem. This forced the rethinking of the experiment. Eventually it was decided that the world would be rebuilt to simulate an uneven terrain with different textures. This world is shown in Figure 4.10. The light coloured texture induces an amount of drag on the robot. This drag is completely artificial and is calculated based on the robot's position. The drag is then subtracted from the motor actuation. Note that the robot starts on the right of the track where no drag inducing textures are present. This is done in order to give the robot a chance to reach the desired velocity before hitting the zones where drag is induced.

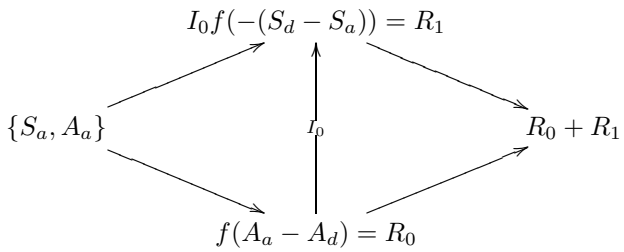


Figure 4.11: The subsumption reflex. I_0 is the inhibition exerted by layer 0. The function $f(x)$ is a hyper tangent used to squash the results.

4.3.2 Changes in the reflex

During preliminary tests the reflex as used in the previous experiments seemed to yield unstable results in this particular Webots world. It was therefore decided that the original reflex would be replaced by an entirely different variant. This variant would be based on the so called *subsumption architecture* first put forward in [Brooks, 1985]. The reflex is still layered and each layer is responsible for a part of the robot's behaviour, much in the same way the original reflex was. The layers are ordered from high priority to low priority. The low priority layers inhibit the high priority layers when they are active. In Figure 4.11 the layers of the new reflex are shown. The inhibition ranges from 0 (total inhibition) to 1 (no inhibition) and is dependant on the output of the layer.

4.3.3 Controller elements

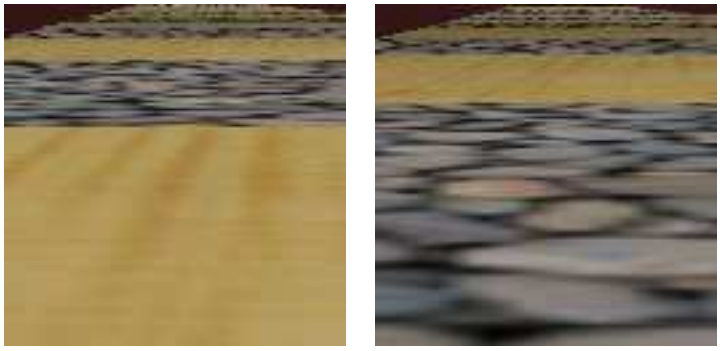
The controller of the robot has two distinct parts which each require further elaboration. First the additional input will be discussed, second the construction of the detectors will be explained together with the actual real-time use of the detectors.

Inputs Still assuming a scale between minus one and one in the latest version of the balancing experiment visual information is provided to the cerebellum in the form of 2 additional inputs. Each output of a detector for a specific texture is rescaled to lay between minus one and one. The drag inducing texture detector is the second of the two. The construction of the actual detectors is not the task of the cerebellum. As pointed out in the introduction the human cerebellum often does use pre-processed information, the task (amongst many others) of the cerebellum is to determine what is relevant for the present issue. However it is relevant to explain the nature of these texture detectors.

Texture detectors Texture detectors can be build in many ways. Histogram approaches are among the frequently used methods, for example using co-occurrence matrices [Ohanian and Dubes, 1992]. Here, however, a different approach is taken. In order to construct the detectors, two pictures on which the two different textures are clearly visible are manually selected, see Figure 4.12 and 4.13 for the to be detected textures and the images taken during runtime

(a) lightwood (b) stone

Figure 4.12: The textures which need to be detected. Both are 128x128 pixels large.



(a) Lightwood prevailing texture (b) Stone prevailing texture

Figure 4.13: The images taken during runtime with either one of the textures prominently present.

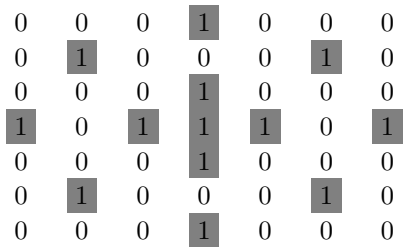


Figure 4.14: The 'retina' as it is used when taking samples. Only the gray cells are selected.

with which the detectors are constructed. The construction is then done as follows.

Collect samples: From two real time images in which one of the two textures was prominently present one thousand samples were taken. The samples are taken at random locations on the texture and are 7x7 pixels large. In addition not every pixel in that patch is processed but a mask as shown in Figure 4.14 is applied to the patch. Only a 'retina' like area is eventually selected from the patch. This is done in order to further reduce the number of pixels which need processing.

Whitening: The next step entails the so called *whitening* of the data collected. This means that the empirical mean is removed from the data (the data is *centred*) and transformed in such a way that the variables (pixels in this case) are independent from each other. This is of course done with PCA as discussed in Section 3.3. This yields a collection of whitened samples. The whitening and unwhitening matrix are saved as they are to be used in the detection process.

Making a SOM: Constructing a SOM is essentially a preprocessing step for k-means to work from. This step makes use of the SOM Toolbox[1] from the Helsinki University of Technology(HUT). This toolbox is written for and in Matlab and uses some advanced features like the initialisation along the eigenvectors and a fine tuning phase.

Clustering using k-means: The SOM Toolbox also includes a k-means clustering algorithm which can be conveniently used with the resulting map from the previous step. Clustering of the map with $k = 2$ yields the two centroids displayed in Figure 4.15.

The resulting two centroids or codebook vectors are then used in the detection process.

4.3.4 Runtime detection

The detection process is very similar to the process used to construct the detectors, see Algorithm 3. The robot will transmit images at request back to Matlab where the image is sampled in much the same way when constructing the detectors only this time 500 samples are taken. These samples are then whitened

[1] `http://www.cis.hut.fi/projects/somtoolbox/`

Figure 4.15: The two cluster centroids describing the two textures best. The images are constructed by inverting the process when making a sample. The used mask is used and the blanks are filled in with a weighted value from the surrounding pixels.

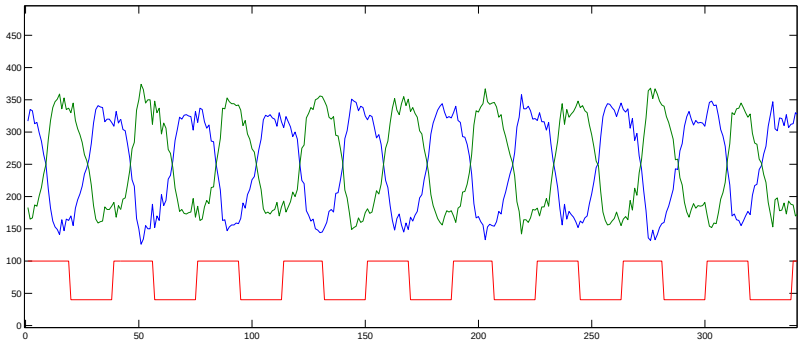


Figure 4.16: The robot is forced to move forward while its centre of gravity is placed between the wheels to keep it from falling over. The jagged lines show the activation of the two detectors, the lowest line shows the transitions between textures over time.

but this time with the stored whitening matrix. This is done because otherwise the rotation done on the sample would be different than for the rotation leading to the prototype or codebook vector. The result would be meaningless. These runtime whitened samples are then binned based on the distance between the sample and the two codebook vectors. The sample is binned in that bin with which the distance is smallest. This results in two bins which, when normalised, are the two inputs to the cerebellum. The distance measure is a relatively simple one

$$d(\boldsymbol{a}, \boldsymbol{b}) = \sqrt{\sum (\|\boldsymbol{a} - \boldsymbol{b}\|)^2)} \tag{4.1}$$

in which the raising to the power of two occurs pairwise for each element in the equal length vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. To show the effectiveness of these detectors see Figure 4.16 in which the robot is manually progressed over the textured terrain all the while processing images. In the figure it is clear that the information from the detectors can be used to reliably detect a texture transition and thus a potential induced drag.

```
Data: Runtime image
Result: Two normalised bins
bins ← ⟨0, 0⟩;
for 0 to maxNrOfSamples do
    sample = getNextSample;
    whiten(sample);
    if distance(sample,detector1) < distance(sample,detector2) then
        bins(0) ← bins(0) + 1
    else
        bins(1) ← bins(1) + 1
    end
    normalisedBins ← bins/sum(bins);
end
```

Algorithm 3: Gathering and processing runtime samples.

4.3.5 Experimental parameters and setup

The experiment was conducted with a reflex in which the desired speed was set to 10 radians per second which is one fifth of the highest theoretical speed of the robot. This was done in order to ensure that the robot traversed the textures at sufficiently high enough speed. The reason for forcing the robot to drive forward is to keep the problem linear. If the robot is driving backward the information of the texture detectors are useless. On the other hand when moving forward this information is relevant. A relation like this is nonlinear and can not be learned by the cerebellum in its current form. When making sure the robot moves forward the information is (almost) always relevant and the problem remains linear. As in the previous experiment the learning rate, length of the eligibility trace cascade and slope parameter were respectively 0.001, 3 and 0.1.

Again two variants of the experiment are conducted. In both cases the detectors are operative but in one case the result of the computations are overwritten by a random value between the usual minus one and one. This is done in order

Detectors enabled		Detectors disabled	
Iterations	Resets	Iterations	Resets
9400	10	11020	11
11900	11	14900	11
10240	11	11200	11
11940	11	9100	10
10380	11	14180	10
10700	11	11440	11
10820	11	12940	12
11160	11	11160	11
10940	11	10140	11
10780	11	11880	12
Averages		Averages	
10826	10.9	11796	11.0

Table 4.3: Convergence results for the drag inducing texture experiment. On the left the results with the texture detectors enabled, on the right without the texture detectors.

to assess the influence exerted by the texture detectors. Both experiments are ran several times until convergence. Convergence in this experiment was defined as achieving a location of 6.5 on the z-axis which is on the other end of the textured track.

4.3.6 Results

Stable performance proved much harder than in the previous experiments. This is due to the added difficulty of not only the different drag factors on the different textures but also due to the addition of a desired speed. This makes it much harder for the robot to balance. Whereas the desired speed had no influence in the previous experiment, where the output of that part of the reflex was set to zero, in this case the desired speed has significant influence on the reflex pushing the robot constantly out of balance. Taken these additional difficulties in account the cerebellar model still assigns a role for the texture detector inputs however small. As seen in Figure 4.17 the last two input units play not an overly large role but are not zero and thus are deemed significant by the cerebellum. What also can be seen from the figure is that in effect the two detectors are complementary. If one takes 500 samples and classifies n samples as one class then the remaining $500 - n$ samples must be classified as the other class. While this was not made explicit this fact is nicely picked up by the cerebellum. If one is to interpret the weights of the last input units one can see that when the drag inducing texture is detected, corresponding to a high value on the very last unit, a small positive contribution is made to the output of the cerebellum. Because the detectors are complementary a similar positive contribution is made by the first detector.

Table 4.3 shows how long it took for convergence to be achieved when the textured detectors were turned on and turned off. As suggested by the weights shown in 4.17 when the detectors are on it converges slightly faster as the

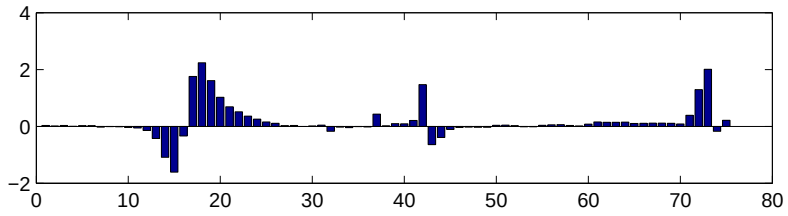


Figure 4.17: The last two of units were used to represent the input from the texture detectors.

cerebellum incorporates this knowledge in its output. However no statistical significance can be adhered to these results as the p-value of both the number of required iteration and resets are respectively 0.2031 and 0.4331.

Chapter 5

GATE

In this chapter explains how GATE came to be. The structure is as follows, first the choices regarding the programs and programming languages will be explained and how they fulfil the requirements of this project. Then a sketch of the actual design of GATE will be presented together with the techniques used.

5.1 Requirements and fulfilments

This section explains why certain languages were chosen for the implementation of this project and why their choice helped meeting the requirements.

Webots and Java Webots is a robotics simulation platform by Cyberbotics Ltd [1]. The program features realistic physics using the Open Dynamics Engine (ODE) [2]. This was an important consideration when choosing Webots because this contributes to allowing us to perform realistic simulations. The importance of this was already shown, to provide a sufficiently challenging environment and it aides in transferring the results to the real world. Note that while a non-realistic physics engine could be very challenging, the absence of a physics engine would prove too simple. The 3D modelling tool as Webots supports it is sufficient, though not great. More important is the fact that the code produced complies for the better part with the Virtual Reality Modelling Language (VRML)[3], this allows one to model environments in third party software as ArtOfIllusion[4] or AC3D[5]. The next attractive feature of Webots is the abundance of working or workable demonstrations bundled with the software. This provides for enough examples for novice users to understand the possibilities of Webots and the ability to forgo trivial though time consuming modelling of robots and environments. Furthermore a reasonably large user base and a very active and helpful developer instilled confidence that possible future problems could be quickly resolved. Last the fact that programming can be done in both C++ and Java. This increases the chances of the future use of Webots by providing this amount of flexibility. Moreover the expertise of working with Java

[1] http://www.cyberbotics.com
[2] http://www.ode.org
[3] http://www.web3d.org
[4] http://www.artofillusion.org
[5] http://www.ac3d.org

was already present with the author expediting the project significantly. Added to these facts which made Java our prime choice was the platform independence of both Java and Webots enabling us to use GATE in any setting. Last the network abstraction is an integral component of Java and while there do exist numerous C++ libraries which provide similar functionality using such a library would introduce yet another dependency.

Matlab When choosing a language to implement our models the choice for using Matlab was an easy one. First of all the very nature of the research done by engineers is highly experimental. This means that fast prototyping of software and models is the order of the day. Scripting languages like Matlab provide the means necessary for quick testing of ideas. Then there is the fact that Matlab suites the computational needs of working with brain models very well. Matlab is specifically designed for efficiency in large arithmetical operations. Last Matlab enjoys wide acceptance within the research community and by using it the potential user base is greatly improved.

5.2 General idea

Taken Webots and Matlab there is no native interface between them other than importing Matlab into Java or C++ which can be done but only with custom-made classes unsupported by Mathworks. Importing Matlab into Java or C++ would still require a fair amount of knowledge of the mentioned programming languages which is exactly what is being tried to avoid. This would compromise the requirement of easy accessibility. More important this would also compromise the idea to use a distributed computing environment. Calling Matlab directly from the Java interface with Webots would limit the users to computing facilities provided by the platform Webots is running on. No additional computers could be hooked into the system. These reasons prompted the development of a library called *GATE* which is an acronym from *Great Access To Everything* enabling the interaction of Webots and Matlab over a TCP/IP connection. When writing this library usability was considered to be of the utmost importance. A (very) low threshold for usage is required to enable researchers to focus on the actual modelling of brain areas. This resulted in a library which tries to stay as close to the original usage of Webots as possible and as far away as possible from any technical details like complicated library imports and lengthy initialisations. The choice to implement the integration of Matlab and Webots with a TCP/IP approach facilitates the requirement of a distributed environment. Using TCP/IP enables multiple Matlab instances to connect to Webots. The library is written in Java, but note that this does not automatically follow from the assumptions in Section 5.1. In that section it was merely pointed out that programming in Webots could be done in both C++ and Java and the latter one was the preferred option. However for much of the same reasons pointed out in that section this library is written in Java. Another and equally significant reason to choose Java over C++ is the native support for *reflection* in Java. For the reader to understand the importance of reflection in this library it is necessary to first delve deeper in the architecture of GATE and the interaction between Webots and Matlab through GATE.

When using Webots there are two tasks a user must typically perform. One

is modelling the environment and the robot in the simulator. Although this is an important and necessary step this is not in the scope of this work to describe it and is left for the reader to explore would he desire to do so[6]. The other task is building a behavioural routine for the robot or, as Webots calls it, a controller. This will also be the naming convention used hence forward. The controller is exactly that, a program written in Java of C++ controlling the actions of the robot. Both languages are more or less interchangeable in functionality within Webots and what follows is equally true for each. Any relevant deviations will be made apparent if instructive.

In this thesis only the surface of building Webots controllers will be touched but a couple of details need mentioning in order to fully comprehend and appreciate the design of GATE. First of all it is always necessary, when writing a controller, to subclass the Webots provided `Controller` class. Note the capital letter used for the `Controller` class, it is customary and even required in Java to write class names with capital letters. This tradition is followed whenever it helps to clarify the text. Subclassing another class means that functionality of the other class is extended through the new class without touching or altering the other class, subclassing is also known as inheritance. Usually the relation between a class and an inheriting class is referred to as a parent child relation. It is important to note what happens whenever the subclass is instantiated. Only *one* object is created which is both an instantiation of the sub- and superclass. The superclass object never sees the light of day. This is important to realise when reflection is explained. Subclassing the `Controller` class provides the user defined class with the necessary methods to be instantiated and run by Webots. Inheritance bears the additional advantage that future changes in the parent class are by definition available in the child classes without the need for any changes, it is only necessary to recompile the subclass.

Next it is imperative to declare and initialise the sensors in the user class as they appear in the simulated robot. This ties the controller to the robot in the simulation. These two steps, subclassing and initialisation of sensors, are still required when using GATE albeit the subclassing changes slightly and one additional method call is required.

Summarising, GATE provides the capacity to let Matlab and a Java controller interact with each other over a TCP/IP connection without departing too much from the usual way in which Webots is used. Further knowledge of building a controller in Webots and Java is assumed, for more information on that subject the user manual of Webots is recommended.

5.3 In depth GATE

The interface is constructed from several classes, see Figure 5.1, together implementing a simple client/server architecture which is packaged in several packages following Java guidelines, see Figure 5.2. The class `TCPConListener` plays the role of server which spawns an instantiation of the `TCPConHandler` class as soon as it receives a request from the client part `MatMain`. The fourth class `TCPController` takes the place of the superclass `Controller` in normal use of Webots. This means that users now extend the `TCPController` instead of the usual `Controller` class, Appendix A elaborates on how exactly GATE should

[6] see for example `http://www.cyberbotics.com/cdrom/common/doc/webots/guide/guide.html`

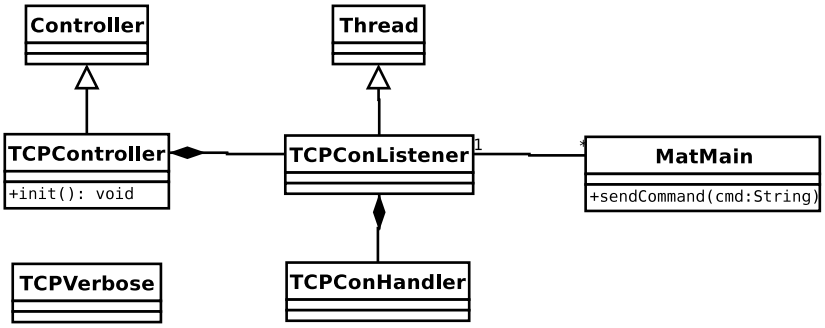


Figure 5.1: The class diagram of GATE

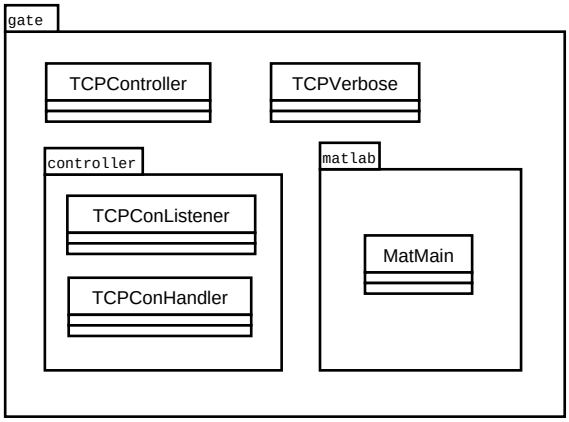


Figure 5.2: The package diagram of GATE

be used. As one can see from the class diagram 5.1 the `Controller` class is not replaced but merely subclassed. In effect the `TCPController` class sits between the ordinary parent `Controller` class and the user created subclass. The overall flow of execution as described above is depicted in Figure 5.3.

Requests Earlier in this section it was stated that the server received a *request* which it would delegate to another class. A request is simply a string of undetermined length sent by the client side (the Matlab side) to the server side. The client sends its request and waits for a response to arrive. This means that Matlab will halt execution until the call returns. While programming the Matlab side this is something to consider and a crash in Webots requires a manual abortion of the operation on the Matlab side. The format of the request string is simple. It must be an existing method call from the controller on the server or Webots side including the parameters. For example when requesting a camera image from a camera with DeviceTag `cam0` the request string should be '`camera_get_image(cam0)`', just as it would be when programming a regular controller. If the string is not a valid method call in the controllers API an exception is thrown.

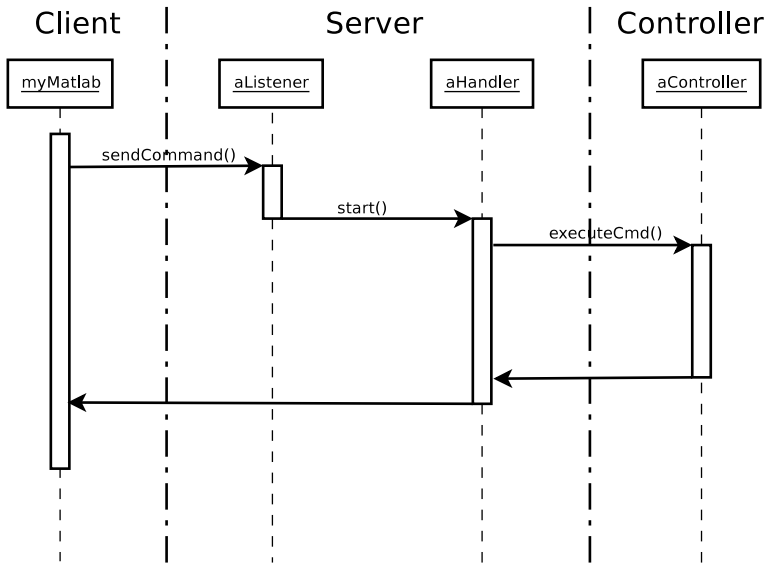


Figure 5.3: The sequence diagram of GATE. Invocation starts at the left side at myMatlab.

Reflection At this point the term *reflection* can be elucidated. As one can see from Figure 5.3 the flow of execution occurs solely between the classes defined in the GATE library. This poses a problem. On one hand the goal of GATE is not to limit the user in any way. The user should be able to do everything with GATE which he could normally do from within an ordinary controller. However at compile time of GATE it is not known what the user might implement in a user defined controller. As such it is not possible to anticipate what request strings are going to be received. If GATE is to preserve the user his full control over the Webots API and the possible user implemented methods it is required to figure out during runtime what the possible method calls are. This is exactly what reflection does. Reflection allows an application or object to discover during runtime information about itself. It can, for example, find out which methods it has. And although reflection is more than just finding out which methods one possesses this is the feature GATE will be using. Remember the procedure for creating a controller for Webots. The user is required to subclass the `Controller` class which takes care of the correct integration of the user class with Webots. Also remember that the GATE defined `TCPController` class subclasses the `Controller` class so nothing of the original functionality is lost. It is in the `TCPController` class where the reflection functionality is implemented. In this way the user defined controller, which inherits from `TCPController`, always has the ability to inspect itself and thus can find out which methods it has. Whatever methods the user adds it is always in his controller and is thus always available for inspection with reflection. The same goes for the Webots provided superclass `Controller`. Whatever methods the Webots developers will add in the future to the controller API these new methods will be available for inspection and invocation in the user defined controller through GATE. All this without needing to change a single line of code in GATE.

The only requirement for reflection to work is that there must *be* a runtime object to be inspected. It is traditional in Webots to place everything in static methods, but this is by no means required. When using GATE, a definition of a constructor should always be included in conjunction with a call to that constructor from the `main()` method. More on the use of GATE can be found in Appendix A.

Chapter 6

Discussion

6.1 Improvements on GATE

Implementations are rarely finished as is the case with GATE. In this section several aspects of the GATE library are discussed which could be subject to improvement.

6.1.1 Missing byte parsing

In the implementation of GATE the request string received from the client is parsed in order to find the correct method call. The parse algorithm is a complex entity which at the heart relies on regular expressions to cast the extracted arguments to the correct corresponding data type. This parsing algorithm does not, at the moment, accommodate the casting of arguments to bytes. This is mainly caused by the fact that implementing such an enhancement would complicate the algorithm significantly. Bytes do appear to be integers except that they are restricted to the range of ones and zeros. Incorporating a mechanism to recognise bytes from integers would be possible when using, for example, a hypothesis on the underlying methods. Assuming the extracted argument is a byte, does Webots have a method which does indeed have byte argument? Based on this result it would be possible to distinguish between integers and byte, but the procedure would complicate matters enormously. Moreover it would add little to the functionality of GATE. There are at the moment of writing only two functions which have a byte argument or a byte return value, and these functions are rarely used. These two facts combined caused the neglect of this feature in favour of more pressing matters.

6.1.2 Reversion with threads

The reversion of the simulation by a supervisor robot in the threaded version of GATE leads to an apparent lock-up. The revert is only executed when one final connection is made to the same supervisor robot. This is because of how the client/server architecture is implemented. Whenever a revert command is sent from the client side the server spawns a thread which handles this incoming request. This newly created thread returns a value to the client and terminates.

However meanwhile the main server thread continues to listen to incoming connections. Webots only reverts whenever that thread progresses in execution. And it only progresses in execution when a connection is made. A possible solution could be the termination of the main thread by the spawned handler thread as soon as this handler thread recognises the revert request. Another solution is the parsing of the request string previous to spawning the actual handling thread and stop listening to incoming requests. Something similar is done with the exclamation mark syntax, see Section A.2. The latter solution is preferred being in line with already implemented functionality, a generalisation of this implementation would not be too cumbersome.

6.1.3 Automatic reconnection attempts

Another improvement to the scheme used in GATE would be the inclusion of automatic reconnection attempts from the client to the server. This is especially useful in case of the threadless version where a previous request has not been completely processed yet and thus the server is not able to accept new requests. Another case where this feature would be desirable is when reverts are automatically issued for the client side. The revert may take arbitrary time to complete and taking care of the reconnection attempts in GATE would improve user experience.

6.1.4 Far future

Much further in the future may even lay different implementations of the client side of GATE. It might be desirable to implement a certain model or controller in a programming or scripting language other then Matlab[1]. As long as this implementation complies with the Webots API and implements the construction of a TCP/IP connection it can communicate with the server side of GATE.

During the time spent on implementing GATE and several experiments it has become clear that the principal language of Webots is C++. Future experiments might benefit from a reimplementation of GATE in C++. This might circumvent many problems as described in Appendix B.

6.2 Cerebellar model

This section holds some suggestions to improve the cerebellar model as it is presented in this thesis. It also highlights some areas in which further research might be viable.

6.2.1 Reflex construction

As described in the first experiment the reflex is a conceptual complex structure in which the coefficient signs have to be thought over rigorously in order to arrive at a correct reflex. It is better to construct the reflex in a subsumption based architecture as is done in the second experiment, as this is more intuitive and gives greater control of the contribution of each layer to the eventual reflex.

[1] for example 3APL

Moreover incorporating every aspect of desired behaviour in a reflex is not necessarily the best approach. For example when conducting the experiment involving textures it was required that the robot would move forward all the time. In this experiment that requirement was coded in the reflex. It might be better to use a different system to convey that kind of information to the model.

6.2.2 Nonlinearity

Another major improvement would be the inclusion of a system capable of learning nonlinear associations instead of the direct input to output coupling. Take for example the texture experiment. In this experiment only a forward looking camera was mounted on the robot and the robot was given time to speed up. This was done in order to prevent ambiguous information being fed to the model. Information would be ambiguous when the robot the relevance of the information depends on the presence or absence of other inputs. When the robot is moving forward the information from the camera, which is pointed forward, is relevant. It enables the cerebellum to predict possible drag. However when the robot is moving backward, when it is regaining its balance for example, the information of the camera is irrelevant. To learn the relation moving forward $\rightarrow$ camera information relevant and moving backward $\rightarrow$ camera information is irrelevant a nonlinear system is required. The inclusion of a, for example liquid state [Vreeken, 2004], would make it possible to encode this information.

6.2.3 Multi-joints

Another aspect which merits further research are multi-joint problems. This would require the model to have multiple outputs as opposed to the singular output used in the experiments described here. In such experiments efferent copies of the output to the inputs would be imperative in order to give the model a sense of context in which it is operating. At the moment of writing such experiments are indeed being performed.

6.2.4 Convergence proof and future experiments

A most natural extension to the textured world experiment is the inclusion of different and harder to separate textures. The method put forward for detection textures is ideally suited for this purpose and would require little if any alteration. This would strain the model to distinguish even small differences. Another extension would be the calculation of not just two bins but a multitude of bins. This would eliminate the exact inverse relation which now existed with the detectors. This would give the model more freedom to choose which inputs are relevant and performance would probably be better.

Unfortunately outside the scope of this thesis would be the mathematical proof pertaining to the convergence of the model when using eligibility traces. While this has been empirically established and is intuitively correct a stringent proof is at the moment missing.

Chapter 7

Conclusion

In this thesis two major advancements are put forward. One is the creation of the GATE library. The other the further extension and application of the cerebellar model.

7.1 GATE

The library GATE fulfils all requirements posed in this thesis. One of the major goals which needed to be fulfilled by GATE was the ease of usability. This in order to make it easy for researchers to start using a viable robotics simulation. Implementing the library in Java made it portable across as many platforms as Webots supports, thus placing no additional restrictions upon its usage. The current implementation is very future-proof as it uses reflexion to translate the incoming requests to actual method calls. In doing so any changes which are introduced in Webots can immediately be exploited without the need for changes in GATE. This further increases its usability because it requires no expertise on the part of the end user. The architecture of GATE makes it even more resilient against future changes because it extends upon the existing `Controller` class. Any changes in this class are, through inheritance, incorporated in the controller the user writes. The ability of GATE to use regular TCP/IP connections further enhances the range in which it can be deployed. It enables researchers to distribute models or use multiple models for robotic control, or use different computing resources to control a robot simulation. Moreover TCP/IP introduces an abstraction from the actual implementation. This can easily be used to have GATE send results to a receiving end which is not necessarily a Matlab session, as long as they both comply to the Webots API.

7.2 Cerebellar model

In the experiments pertaining to the cerebellar model several aspects are explored. It has been shown that the model is very tolerant toward noise in the input, even when a large part of its inputs consists of noise. Performance only deteriorates when a large portion of the noise relaying inputs are highly active resulting in a disproportioned activation of the cerebellar model.

On part of the second experiment much more can be concluded. In this experiment several weaknesses are exposed which merit further research. The inclusion of nonlinearity in the model should be considered as this greatly improves the range of problems for which the model can be deployed. Also the reflex as used in the first experiment has been enhanced which in turn improved performance dramatically. What the second experiment did not show was that the model was able to efficiently pick up the information regarding the textures.

Bibliography

[Albus, 1971] Albus, J. (1971). A theory of cerebellar function.

[Anastasio, 2003] Anastasio, T. J. (2003). Vestibulo-ocular reflex. In Arbib, M. A., editor, *The Handbook of Brain Theory and Neural Networks*, pages 1192–1196. The MIT Press, 2nd edition.

[Barto et al., 1999] Barto, A. G., Fagg, A. H., Sitkoff, N., and Houk, J. C. (1999). A cerebellar model of timing and prediction in the control of reaching. *Neural Computation*, 11:565–594.

[Brooks, 1985] Brooks, R. A. (1985). A robust layered control system for a mobile robot. Technical report, Cambridge, MA, USA.

[Geng et al., 2006] Geng, T., Porr, B., and Wörgötter, F. (2006). Fast biped walking with a sensor-driven neuronal controller and real-time online learning. *International Journal of Robotics Research*, 25(3):243–259.

[Grethe and Thompson, 2003] Grethe, J. S. and Thompson, R. F. (2003). Cerebellum and conditioning. In Arbib, M. A., editor, *The Handbook of Brain Theory and Neural Networks*, pages 187–190. The MIT Press, 2nd edition.

[Hotelling, 1933] Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24:417–441 and 498–520.

[Hyvärinen et al., 2001] Hyvärinen, A., Karhunen, J., and Oja, E. (2001). *Independent Component Analysis*. Wiley-Interscience.

[Jackson, 1991] Jackson, J. E. (1991). *A user's guide to principal components*. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. John Wiley & Sons, New York.

[Joensuu, 2006] Joensuu, H. O. (2006). Adaptive control inspired by the cerebellar system. Master's thesis, Helsinki University of Technology.

[Kawato, 1999] Kawato, M. (1999). Internal models for motor control and trajectory planning. *Current Opinion in Neurobiology*, 9(6):718–727.

[Kawato and Gomi, 1992] Kawato, M. and Gomi, H. (1992). The cerebellum and vor/okr learning models. *Trends in Neuroscience*, 15(11):445–453.

[Kohonen, 1992] Kohonen, T. (1992). New developments of learning vector quantization and the self-organizing map. *Symposium on Neural Networks: Alliances and Perspectives in Senri*.

[Kohonen, 1998] Kohonen, T. (1998). The self-organizing map. *Neurocomputing*, 21(1-3):1–6.

[Kohonen, 2001] Kohonen, T. (2001). *Self-Organizing Maps*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Lampinen, 1992] Lampinen, J. (1992). On clustering properties of hierarchical self-organizing maps. In Aleksander, I. and Taylor, J., editors, *Artificial Neural Networks, 2*, volume II, pages 1219–1222, Amsterdam, Netherlands. North-Holland.

[Michel, 2004] Michel, O. (2004). Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42.

[Ohanian and Dubes, 1992] Ohanian, P. P. and Dubes, R. C. (1992). Performance evaluation for four classes of textural features. *Pattern Recognition*, 25(8):819–833.

[Prescott et al., 2006] Prescott, T. J., Gonzlez, F. M. M., Gurney, K., Humphries, M. D., and Redgrave, P. (2006). A robot model of the basal ganglia: Bahavior and intrinsic processing. *Neural Networks*, 19:31–61.

[Raymond and Lisberger, 1998] Raymond, J. L. and Lisberger, S. G. (1998). Neural learning rules for the vestibulo-ocular reflex. *The Journal of Neuroscience*, 18(21):9112–9129.

[Smith, 2002] Smith, L. I. (2002). A tutorial on principal components analysis.

[Smith, 1998] Smith, R. (1998). *Intelligent Motion Control with an Artificial Cerebellum*. PhD thesis, University of Auckland, New Zealand.

[Strang, 1998] Strang, G. (1998). *Introduction to Linear Algebra*. Wellesly-Cambridge Press, 2rd edition.

[Vesanto and Alhoniemi, 2000] Vesanto, J. and Alhoniemi, E. (2000). Clustering of the self-organizing map. *IEEE-NN*, 11(3):586.

[Vreeken, 2004] Vreeken, J. (2004). On real-world temporal pattern recognition using liquid state machines. Master's thesis, Utrecht University.

Appendix A

Using GATE

In this appendix a detailed description will be given on how to use GATE. The library can be downloaded from `http://www.lce.hut.fi/~harm` or it can be found on the original Webots installation CD. When using GATE there are two things a user needs to do. One is the creation of a GATE 'enabled' controller. The other is the creation of a Matlab script sending commands or requests to Webots. First the creation of a GATE enabled controller will be explained.

A.1 Creating a GATE enabled Java controller

In the following description basic knowledge of creating a Java controller for Webots is assumed. There are really few things which need adjusting when compared to a regular Java controller. Starting at the top the imports need to change. Remember that GATE resides between the Webots provided packages and classes and the user controller. All required imports for Webots are done in the GATE package and can thus be omitted. The user only needs to import the GATE packages:

```
import gate.*;
```

No further imports are required although the user may import other packages as he sees fit. Be sure that the JAR files containing the GATE packages can be found by the Java compiler and the Java Virtual Machine (JVM), in other words it needs to be in the classpath.

Next the usual class to extend (or subclass) is not the `Controller` class but the `TCPController` class. A typical class definition looks as follows

```
public class MyTCPController extends TCPController {
```

The two changes required alterations on the original procedure. Next a couple of lines need to be added to the controller in order to get to a working implementation.

When writing a normal Java controller for Webots an endless control loop is used to keep the robot running. Usually, that is in a Webots Java controller, this loop is located in the `main()` method of the class. This is quite unusual for normally a Java class instantiation is started from the `main(String[] args)` method. It is possible to place the loop in the regular main method but staying

as close to the original intend of Webots is deemed best. However with GATE the goal is to determine the actions of the robot from outside Webots. That is in a Matlab script. There is therefore no need for an endless loop because this will (should) be handled in the Matlab script. Instead of this control loop a call from the `main(String[ args])` method to the constructor of the class is required. In Section 5.2 it was shown that for reflection to work an instance of the object is required. This is exactly what a call to the constructor does. It creates an object which in turn can be inspected. In the constructor a single call to the method `init()` needs to be placed. This is necessary for constructing a mapping from strings declared in the user defined controller containing the DeviceTags to integers representing the DeviceTags. It is in the constructor where the user can determine the server port on which to listen. This is done by overwriting the `_port` variable. It would be wise to select a sufficiently high port (over 5000) in order to avoid collisions with other programs. The default port is set to 7890. In that same constructor the default verbosity can be set with the boolean `_verbose` variable. Again this variable can simply be overwritten, default it is false. This increased verbosity will show the handling of the incoming connections and spawning of handling threads. This functionality is very much oriented at developers of the GATE library and regular users would probably want to leave this boolean alone. A typical constructor would look like this:

```
public MyTCPController(){
    _port = 9999;
    _verbose = true;
    init();
    robot_console_print("New MyTCPController controller created,
        listening on port: "+ _port);
}
```

Security policy There is one last detail which needs to be taken care of. In principle this has nothing to do with the creation of a GATE enabled controller. It has to do with the fact that Webots shields Java controllers from incoming TCP/IP connections, this is done to prevent cheating during a robotic contest (the Roboka contest) organised by Cyberbotics. This can easily be changed in the `java.policy` file found in the Webots install directory. The line:

```
permission java.net.SocketPermission "localhost:1024-","listen";
```

should be changed to accept incoming connections. This is done by applying the following change:

```
permission java.net.SocketPermission  "*:1024-","listen,accept";
```

Now incoming connections are accepted from any IP address.

A.2 Creating a GATE enabled Matlab script

The creation of Matlab scripts is fairly straightforward as well. First Matlab needs to know where it can find the JAR archive, from the Matlab commandline:

```
javaaddpath('path_to/gate.jar');
```

This command only works with Matlab 7 or higher but in principle lower versions are able to handle Java classes and thus should be able to include the JAR file. Next is the most crucial and last part. The initiation of the Java class

```
m = gate.matlab.MatMain(port_to_connect_to,
        'name_or_ip_of_remote_server');
```

In the variable `m` the reference to the newly created Java object is stored. The rest really speaks for itself. The first argument should be an integer and the second a string. The string argument can be an IP address or a hostname.

The sending of requests is done with the only public method the `MatMain` class has. Namely `sendCommand(String cmd)`. This method takes a string as argument in the form explained in Section 5.3. For example if the user wants to find out what the name of his robot is (useful when controlling multiple robots) he does

```
name = m.sendCommand('robot_get_name()');
```

Note the return value being stored in the variable `name` which has immediately the correct type. Several other examples can be found in the readme file which is included with the GATE library.

One other remark is necessary at this point. When in a Webots simulation multiple robots are present and the simulation is running in synchronous mode (which is default), Webots does not progress time for the simulation if a `robot_step(n)` command is given to only one of the robots and as a consequence that call made from Matlab will not return. Because Matlab does not support proper threads GATE implements a special syntax to tell the server side to return immediately. Whenever a command is prefixed with an exclamation mark '!' the command will return immediately after being received on the server side. For example

```
m.sendCommand('!robot_step(32)');
```

The return value is not captured here because it will be 'nil'.

A.3 Special commands

GATE sports two different special commands which can be sent in the usual fashion. The most important of the two controls the verbosity of the parsing algorithm. These are a simple 'true' and 'false', the first turning on the verbosity and the second turning it off. When verbosity is turned on a command from the client side like `m.sendCommand('differential_wheels_set_speed(10,12)')` will show up in the terminal in which Webots is started as:

```
differential_wheels_set_speed(10,12)
Element to be tested:'10'
Element to be tested:'12'
Extracted types:
Element 0 = int
Element 1 = int
Extracted values:
Element 0 = 10
Element 1 = 12
```

This shows the several steps in the parsing algorithm. First is the extracted command. Then the arguments are displayed, at that point still strings. Then the types of the arguments are determined and last the actual value is associated with the initially extracted type.

The second special command is more a courtesy to the Java Virtual Machine. Before a revert command is given to a supervisor robot the command 'die' can be sent to the other robots. This terminates the server on these robots leading to a clearer termination of these controllers. A note of caution, this command has not been fully tested and its use is probably not even necessary.

Appendix B

Problems and solutions when building GATE

In this appendix an elaboration on the many problems encountered when implementing, testing and using GATE will be given. This in the hope that possible future users and developers find it useful when doing the same.

First an overview of the used software will be presented with which GATE plays nice in order to frame the following discussion in the right context.

- JDK 1.5.0_03 and 1.5.0_06
- Webots 5.0.6, 5.1.4, 5.1.7 and 5.1.9
- Matlab 7.1.0.183(R14) SP 3 and Matlab 7.0.0.19901(R14)
- Redhat 6.2 and 7.2
- Mandriva 10.0
- Ubuntu 6.06 and 6.10

B.1 Limited connections

After using GATE for a while new connections where mysteriously refused. Everything seemed to be in working order but the new connection initiated from Matlab did not connect. A generic `connection refused` error was returned. After increasing verbosity on the server side it appeared that the Java Virtual Machine ran out of so called 'file descriptors'. These descriptors are low level pointers toward files. And, as the Unix system does, file descriptors can be anything from real files to virtual devices but also network connections. The virtual machine used at that time did not seem to release these descriptors properly and ran out of them leading to connection errors. Increasing the number of available descriptors postponed the problem but did not solve it. Closer inspection of the running 'java' instance revealed that the command was in fact aliased to the 'kaffe' binary. Kaffe is an implementation of the JVM specifications intended for education. It is written as clean as possible but only implements a subset of features. Perhaps more problematic was the version implemented in this specific

binary which was version 1.1 dating back to 1997. Installing the most recent version of Sun's JDK and using that java binary solved the problem, it now properly released the file descriptors allowing more than the initial thousand connections.

B.2 Transmitting images with threads

There were actually two obstacles to overcome when trying to transmit camera images from Webots to Matlab. First a request to send an image with `m.sendCommand('camera_get_image(cam0)')` crashed Webots and hung Matlab due to non responsiveness from the simulator. An error message

```
GLX: cannot make current.
```

was observed on the server, thus the Webots, side. This problem only occurred when using a JVM from Sun Microsystems®. The problem was eventually traced to the use of the subsystem called X on Linux which is responsible for all the graphics. Both types (X.org[1] and XFree86[2]) were tested and affected by the problem. This subsystem in combination with Webots, the JVM from Sun and the use of threads within the Java controller caused this problem. If and only if this combination of programs and techniques was used it yielded this error. From Section B.1 it is clear that another JVM could not be used. The Linux operating system was at that moment the only OS available and the general functioning was far preferred over the other alternative. Both alternatives of X caused problems closing that avenue as a solution. The only alternative left was the removal of the additional threads in GATE. This let to the incarnation of the *non-threaded* version of GATE. This does work but the threaded version is preferred for its ability to process concurrent requests which fits the requirement of a distributed environment better.

There was one additional problem when transmitting the images to Matlab. The call `camera_get_image` returned an array of positive and negative integers and by no means a usable image. After adjusting for the number of colours possible by adding to every element in this array the number $2^{24} \approx 16.7 \times 10^6$ it turned out to be bits describing a colour image. Each eight bits defined a colour in the order red, green, blue. After shifting the colour adjusted array by the appropriate number of bits for each colour and merging the resulting three mono colour images into one image. The final step entailed translating the obtained image 90 degrees clockwise in order to correct to the original orientation as the intermediate image was rotated 90 degrees counter-clockwise.

B.3 Timely connections

Creating reliable connections seemed to be a persistent problem. With the problem of file descriptors solved several other problems emerged. One of which was that sometimes (not deterministic) Webots did not seem to respond to `robot_step(n)` request. This problem could be solved by sending the same request again or, more puzzling, with $n = 0$. This could be worked around

[1] `http://x.org/`
[2] `http://xfree86.org/`

from within Matlab by checking whether or not the request was processed or by just sending a couple of requests with $n = 0$ but this was clearly undesirable. The solution lay equalising the size of the timesteps taken in both Matlab and Webots. A Webots world has a node called 'WorldInfo'. Typically the name of the world is designated here as well as the strength of gravity among other things. Also the 'basicTimeStep' is defined here. The trick is to take timesteps, with `robot_step(n)`, that are multiples of the 'basicTimeStep' as defined in that particular world file. Doing so eliminates the problem of ignored requests.

Another problem occurred when there were multiple robots (either regular or supervisors) in the world. Sometimes (again non-deterministic) one of the robots failed to initialise after a `simulation_reset` request from a supervisor. The failing robot just never returned from the constructor method call. This was and is very odd and made it impossible to run long experiments. It appears that Webots does not always wait for all robots to finish initialising and just starts the simulation. It does, however, not start when not a single robot is done initialising. This fact was exploited to circumvent the problem. What can be done if this problem occurs is to hold further execution of the controllers by one second (arbitrary) after they are done initialising. In code terms place a

```
try {
    Thread.currentThread().sleep(1000);
}
catch(Exception e){}
```

block after the `init()` call. There exists one more timing problem which will be discussed in the last paragraph.

B.4 Java support

The creators of Webots tout their product as being compatible with both Java and C++. Partly based on this statement the choice was made to acquire Webots as it appeared to support Java as well. Unfortunately this was not exactly the case. When implementing a Java controller for a supervisor some methods were absent which were available in the C++ API. It was impossible to fix this without the help of the developers. Turning toward them, however, did help in a relatively short period of time and most of the missing methods are now added to the new releases of Webots.

B.5 Physics

Another reason for choosing Webots was the support for intricate and realistic physics. This worked well up till the point where a world was designed in which the robot needed to balance in an uneven terrain, see Figure 4.9. During preliminary tests the robot displayed very strange behaviour. It seemed that it was bouncing off the *IndexedFaceSets (IFS)* which made up the uneven terrain. This problem seemed only to occur when an IndexedFaceSet was used in conjunction with a cylinder or sphere. It was of these objects our robot's wheels were made. Eventually the problem was traced to the physics engine used in

Webots namely the Open Dynamics Engine or ODE[3] for short. The collision detection between cylinders or spheres with IFS seemed faulty. The problem seems to persevere and usage of IFS in combination with cylinders or spheres is not recommended.

B.6 Loose ends

This paragraph will be dedicated to the several problems which remain and are not properly addressed or investigated.

At the moment version 1.4.1 of GATE does not support the transference of bytes. Arguments concerning bytes are not properly parsed and are cast to the default float array. This is not a big problem because only two methods require bytes as either an argument or return value.

Another more vital problem is the fact that after a large number(5000+) of reverts initiated by a supervisor none of the controllers get re-initialised which result in failures to connect from the client side. This problem seems unrelated to the problem described in the paragraph on timely connections and as it stands there is no solution other than manually reverting or restarting. This is a nuisance and could possibly be dealt with from within Matlab. Matlab could detect failures to connect and restart Webots. And although Matlab possesses the ability to call lower level system calls there is no guarantee this will work. Nor is it trivial to write such a procedure.

[3] `http://www.ode.org`