TDT4900 Computer and Information Science, Master Thesis

A behavior-based system for control of mobile robots

NTNU, June 14, 2010

Maria Johansen

Supervisor: Keith Downing

Abstract

This report will explore behavior-based robotics and relevant AI techniques. A system for autonomous control of mobile robots inspired by behavior-based robotics, in particular Rodney Brooks' subsumption architecture, have been implemented, adapted for use in a multiagent environment. The system is modular and flexible, allowing for easy addition and removal of system parts. A weight-based command fusion approach is taken to action selection, making it possible to satisfy multiple behaviors simultaneously.

Preface

This master thesis is written at the Department of Computer and Information Science, NTNU during the spring semester of 2010, and it is a further development of work done during my pre-study project of the fall of 2009.

My supervisor for the project has been professor Keith Downing and the assignment text was as follows:

> *Robots are commonplace in society; they perform all sorts of jobs. However, most are hard-wired to perform a few fixed tasks and have little or no ability to adapt to changing or unforseen circumstances. Artificial Intelligence (AI) enters the picture when robots must be capable of autonomous behavior in unpredictable environments. In this project, the student must utilize AI machine-learning techniques to produce a robot with adaptive capabilities. Possible techniques include reinforcement learning (RL), artificial neural networks (ANNs), evolutionary algorithms (EAs), or combinations of these, such as evolving neural networks. A few e-puck robots are available in our laboratory, and these should be sufficient for building adaptive robots. Students who desire more advanced equipment will have to arrange for it themselves.*

Originally, my aim for the project was to work on developing a neural network "brain" for controlling the robots. During my research for the pre-study project during the fall of 2009, I came across several interesting behavior-based approaches to robot control. Inspired by these systems I have to a considerable extent incorporated behavior-based ideas into my project, using neural networks for low-level control in some of the modules in the system.

Finally, I would like to thank professor Keith Downing for his input and advice during the past year, and the members of the Webots user group at Yahoo for their in-depth knowledge of the Webots simulator.

Trondheim, June 2010

Maria Johansen

Contents

1 Introduction

This report presents my master's thesis on behavior-based robotics. The assignment was given by professor Keith Downing, at the Department of Computer and Information Science, NTNU.

In today's society, robots and autonomous systems are everywhere. In factories, robots are used for assembly line work and manufacturing, freeing human operators from tedious manual labor. In Copenhagen, some local trains are completely autonomous, no personnel on board the train is needed. Robots are used as tour guides[1], and in homes robots take over more and more of the common household tasks such as lawn mowing and floor cleaning.

There are multiple definitions of both intelligence, and intelligence applied to robots. Arkin defines an intelligent robot as *a machine able to extract information from its environment and use knowledge about its world to move safely in a meaningful and purposive manner* (Arkin 1998). An unmanned train does not need advanced cognitive skills, but neither can it mindlessly follow some pre-programmed route. As a minimum it needs sensors detecting obstacles in its track, nobody would want a homicidal train running loose, killing stray cats and dogs and children. It should also be able to distinguish between the aforementioned and harmless objects such as plastic bags[2]. So there has always been a natural link between artificial intelligence and robotics. Robots that are to navigate in a dynamic environment will need some way to observe its surroundings and gather information from it, and it will need to act based on that information. Manual programming and hard-coding of actions can solve quite a few problems, but it is impossible to predict all kinds of unexpected situations that can occur.

The field of artificial intelligence is closely related to studies in biology and psychology. As the views on biological intelligence change, new ideas find its way into artificial intelligence research. The progress in computer science and cybernetics have also played a part, today's robots have capabilities and computational powers never dreamed about some decades ago. One of the first mobile robots with camera vision, the "Stanford Cart", needed breaks of 10-15 minutes for every meter it

[1]In the National Museum of American History, `http://www.cs.cmu.edu/~minerva/`

[2]The latter was not yet the case when I last traveled with the Copenhagen train, a few years ago when it was still in its test-phase.

moved in order to process image data (Moravec 1980).

Earlier attempts to accomplish robot intelligence have been heavily based on world models and symbolic representations, and emphasis has been on high-level planning and pure logic reasoning. The limitations of this top-down approach, together with new discoveries regarding the parallel nature of the human brain[3], led to the evolution of reactive, and later behavior-based robot control systems. Instead of the centralized and serial planning module included in deliberative systems, control in a behavior-based system is distributed, with close connections between sensing and acting.

Behavior-based robot control systems have been used for a wide range of applications, from the commercial vacuum cleaner Romba[4] to soccer playing robots (Laue & Röfer 2004). Other systems are of a more academic nature, created in order to demonstrate some functionality. Examples are the robot Toto (Mataric & Brooks 1990), implemented using a subsumption style architecture and able to navigate using landmarks, and Mataric's Nerd Herd (Arkin 1998, pages 170-172), a group of robots demonstrating emergent social capabilities such as flocking and group foraging.

The motivation behind the project has been to explore various existing robot control approaches, and based on these implement a system for autonomous control of an e-puck robot. The system presented in this report is inspired by behavior-based robotics, it is modular and functionality can easily be added and removed. The system is adapted for use in a multiagent system, and have built-in support for e-puck to e-puck communication. The intention is to use this system as a basis for further research and development.

1.1 Report structure

Chapter 2 contains background information on robotics, and also various relevant subjects in the field of artificial intelligence, especially neural networks. Chapter 3 introduces the systems created for this project, while some experiences from running the behavior control system are given in chapter 4. Chapters 5 and 6 evaluates and concludes the report. The appendices gives a short tutorial to using and modifying the different systems, together with the necessary setup files for recreating the neural networks described in the report.

[3]My guess is that the advances in parallel computing that took place in the 80's also played a role in moving away from serial implementations.

[4]`http://store.irobot.com/corp`

2 Background

This chapter will explore various topics related to robotics and artificial intelligence. It starts out exploring various ways in which to obtain autonomous control of a robot, and then looks at the artificial intelligence techniques relevant to robotics. A special emphasis is on artificial neural networks.

2.1 Robot control systems

Just as humans have a central nervous system controlling the body, robots need a controller system to coordinate sensory input and actuator commands.

Traditionally, the controller architectures were designed by combining the three primary processes *SENSE*, *PLAN* and *ACT*. Sensor input is gathered by the *SENSE* module, and then forwarded further into the system. *PLAN* combines sensor input and world knowledge into the next task for the robot to perform. The *ACT* process interprets this task into output actions (such as moving the wheels of the robot).

There are basically four main approaches to using AI to design a robot control architecture; deliberative, reactive, hybrid deliberative/reactive and behavior-based (Murphy 2000). The next sections will give a short overview of each of them.

2.1.1 Deliberative systems

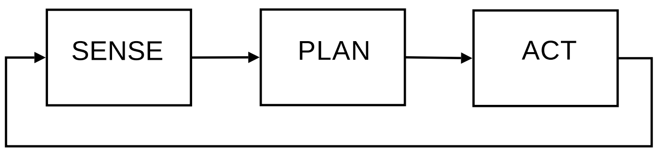


Figure 2.1: Model of the hierarchical paradigm in AI robotics.

The deliberative, or *hierarchical*, approach to AI robotics was dominant in the early days of AI. As the name implies, the hierarchical robot controller functions as a serial unit. In order to produce output, the robot has to sense its surroundings (gather sensor input), plan its next step by analyzing the input, and then act according to the calculated plan. In most deliberate system, the planner does its work on a model of the robot's environment. Sensor input is added to this model in each time step, thus making the model more and more accurate (that is, given that world remains stationary).

One problem with this approach is that it proves difficult in a dynamic environment. Compared to reacting, planning is slow. And while the robot uses processing time while calculating its next move, the environment it operates in may have changed so that its world model no longer is an accurate representation. In addition to this, the more complex the environment, the more processing power and memory is needed to accurately model the world and calculate the next step.

Shakey[1], probably the first robot with artificial intelligence, used a hierarchical controlling system. It used camera vision and bump sensors for navigation, and was able to perform tasks such as moving blocks from one location to another.

2.1.2 Reactive systems

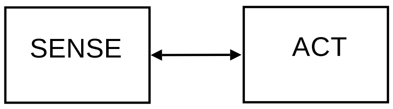


Figure 2.2: Model of the reactive paradigm in AI robotics.

The above-mentioned problems with the hierarchical approach, combined with new knowledge in the field of biology and cognitive science, led to the development of reactive systems. The time-consuming planning stage of the hierarchical controller was cut out, and sensor input became directly coupled to acting. Thus systems created in this fashion are better suited for dynamic environments, they act upon close-to-real-time sensor values, instead of preprocessed sensor input and world models.

The controller may process several sense-act pairs in parallel resulting in outputting a combination of several actions, unlike a hierarchical controller that is only able to process one operation at a time.

This is analogous to how reflexes work in humans, when encountering for instance extreme heat, "sensor input" is immediately translated into action by the spinal cord, instead of forwarding the signals to the brain for processing.

[1]Created at Stanford Research Institute in the late 1960's, `http://www.ai.sri.com/shakey`

When encountering problems that cannot be translated to simple sense-act operations, for instance problems that require memory, pure reactive systems reach a dead-end.

But even without memory, researchers have been able to create pure reactive systems that are fairly complex. Many relatively "simple" animals have a mainly reactive nervous system, and researchers have been able to simulate insect-like behavior using a reactive system (see for instance Arkin (1998)).

2.1.3 Hybrid deliberative/reactive systems

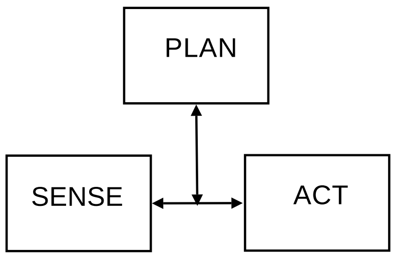


Figure 2.3: Model of the hybrid paradigm in AI robotics.

As a result of the limitations of the reactive approach, a hybrid between the hierarchical and reactive paradigms was born in the early 1990's, and is frequently used in industrial robots today.

The planning unit in a hybrid approach is responsible for decomposing complex tasks into subtasks, that can be translated into actions. The sensing input is handled in both hierarchical and reactive style. Input that can easily be translated into an action is executed in the style of the reactive paradigm. Input that needs further processing is sent to the planning unit.

Input that is treated as reflexes may also sent to the planner, to be incorporated in the robot's world model, or its "memory".

2.1.4 Behavior-based systems

Behavior-based robotics has much in common with the reactive approach from section 2.1.2. Both are based on having a set of tasks (actions/behaviors) running in parallel, with no centralized control. There appears to be a fairly diffuse boundary between reactive and behavior-based systems, in some literature behavior-based robotics is treated as a subgroup of reactive systems(Murphy 2000), while others describe it more as a further development of the reactive paradigm (Mataric & Michaud 2008, page 896).

However, what *is* agreed upon is that pure reactive systems are strictly limited to a sense-act type organization, there is no memory and hence the system has no ability to learn and develop itself. Behaviors in behavior-based systems *can* be as simple as a sense-act pair, but they may also be more complex and incorporate such things as learning and reasoning, making behavior-based robotics more powerful.

Examples of behaviors are "avoid obstacles" or "locate food". In a behavior-based system, all behaviors ought be able to function independently of each other. A consequence of this is the possibility of developing and adding behaviors incrementally, making it easier to test and debug an otherwise complex system. Behaviors are usually manually created, but some algorithms for automatic design also exist. Jenkins & Mataric (2002) describes a method for creating basis behaviors for control of a humanoid robot.

A behavior-based system should be *situated* and *embodied.* That is, it should deal with the environment in a direct way, and not through abstractions or models. All behaviors should have direct access to sensor output and actuators.

One of the main researchers in this area, Rodney Brooks, has this to say on the use of models in behavior-based architectures (Brooks 1990):

> *Our experience with this approach is that once this commitment [to the physical world] is made, the need for traditional symbolic representations soon fades entirely. The key observation is that the world is its own best model. It is always exactly up to date. It always contains every detail there is to be known. The trick is to sense it appropriately and often enough.*

However, absolute compliance with this statement involves a risk of reducing the flexibility and usefulness of a system. An example is DAMN, a behavior-based control system that is studied in greater detail in section 2.1.5.

With multiple behaviors running simultaneously, some form of control system is necessary. Usually, the controller employs some form of behavior arbitration or command fusion, or a combination of both. Systems with an arbitration-style action selection will select the most appropriate behavior in each time step, allowing this behavior full control of the robot. On the other hand, command fusion controllers take into consideration the result of all behaviors, and attempts to merge them into one set of actuator commands.

In the next sections a couple of behavior-based architectures will be presented.

2.1.5 Behavior-based robot architectures

Some of the architectures described below are described as reactive, others as hybrid-architectures. The different methodologies are merely a way of grouping systems, there are no definite boundaries between them.

What all the systems described over the next pages have in common is that they all in some way or another incorporates the ideas of behavior-based robotics.

The subsumption architecture

The subsumption architecture was conceived by Rodney Brooks in the 1980's (Brooks 1985), and is one of the most well-known behavior-based robot architectures.

Behaviors in a subsumption system are organized in a layered structure, or *levels of competence* as they were named. The lower levels represent basic survival functions, whereas the higher levels consist of more "intellectual" tasks, such as route planning. When running, the robot should execute all competence levels in parallel, with no level being in control of the others. However, levels are able to stimulate or inhibit the levels below them (see figure 2.4).

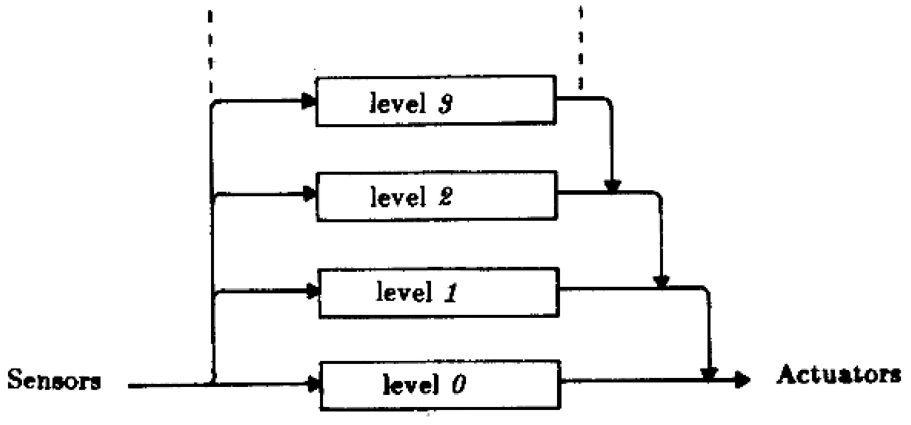


Figure 2.4: Illustration showing levels of competence in the subsumption architecture. Courtesy of Brooks (1985).

One of the important aspects with this structure is that one layer is only dependent on the functionality of its lower levels, and not the other way around. This makes it possible to develop the robot in incremental stages, first implementing basic functionality (moving around, obstacle avoidance and so on). Higher level behaviors are then added, without any need for modifying the existing lower levels.

Coordination of behaviors in the subsumption architecture is priority based, by using the layers' abilities to suppress and inhibit lower layers. In practice, this approach may prove difficult in many-layered systems. The complexity of the action selection increases drastically with the number of layers, and the layers also tend to interfere with each others' goals.

In Brooks' original version, all competence levels were implemented as augmented finite state machines (AFSM's), with each level running on a separate processing unit. As a demonstration of his architecture, Brooks created the robot Allen, implemented as a mobile robot connected by cable to a Lisp-machine doing all calculations.

Another system implemented using the subsumption architecture is Cog (Brooks et al. 1999)[2], a humanoid robot developed at MIT. It has a sensory system built to resemble that of a human, and a body consisting of two functioning arms, a head and a torso (but no legs). In the beginning, Cog resembles something like a human baby. It has basic abilities like being able to move body parts and gather sensory input, but not with the level of control seen in an adult. By interacting with humans, it is able to learn and develop new skills. The intention is to use the robot to study and test theories of human and artificial intelligence.

DAMN

DAMN, or Distributed Architecture for Mobile Navigation, is a behavior-based architecture primarily made for robot navigation (Rosenblatt 1995). The DAMN architecture use a command fusion system for its behavior coordination, allowing for simultaneously satisfying more than one behavior.

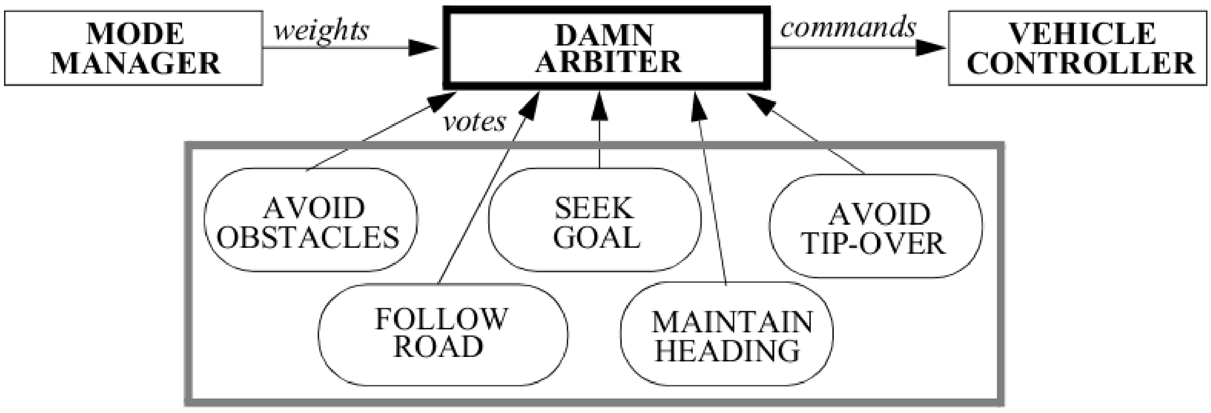


Figure 2.5: Overview of the DAMN architecture. Courtesy of Rosenblatt (1995).

[2]`http://www.ai.mit.edu/projects/humanoid-robotics-group/cog/cog.html`

During execution, all behaviors will generate a vote for all possible navigation commands. The controller will take the weighted sum of all votes, and from this deduce which command is the best.

Where appropriate, DAMN behaviors will make use of internal world representations. Even though opposing Brooks' philosophies from section 2.1.4, this relieves the robot's sensors from some of their responsibility, they need not at all times have a full overview of the environment.

Schemas and AuRA

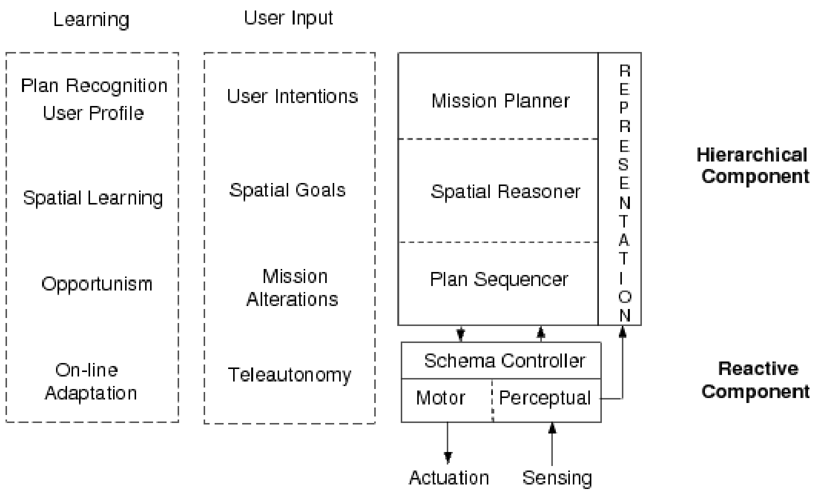


Figure 2.6: High level schematic of AuRA. Courtesy of Arkin & Balch (1997).

Schema theory first originated in psychology and neurology at the beginning of the 20th century, as a method of decomposing human behavior into functional modules, describing the interaction of perception and action.

Arbib was one of the first to combine schema theory and robotics, and the idea was further developed by, amongst others, Arkin in his Autonomous Robot Architecture (AuRA) (Arkin 1998). In a robotics context, Arkin (1989) describes schemas as *basic units of behavior specification.* Complex behaviors can be created by the combination of multiple schemas.

Two general groups of schemas are defined; *perceptual* and *motor* schemas. Motor schemas produce output actions, while perceptual schemas are connected to the robot's sensors, gathering input that can be used by the motor schemas.

Instead of organizing behaviors in a static structure, AuRA use a dynamic network of schemas based on the current goals of the system. As new perceptual data becomes available, the structure of the network changes.

Rather than a “winner takes it all” type behavior arbitration, a fusion method is used. Each behavior contributes to the output action, the magnitude of the contribution is determined by the current schema network. In AuRA, behavior output is represented as vectors, generating potential fields as described in section 2.1.5.

Activation networks

Based on the *Society of the Mind* theory of Minsky (Minsky 1986) and Brooks' subsumption, Maes describes activation networks as *a society of interacting, mindless agents, each having their own competence* (Maes 1989).

The idea is that in a system with multiple goals, the competence modules are to cooperate locally so that the system as a whole should select “good enough” actions. That is, actions that are not necessarily at all times optimal, but are still acceptable. This is done by having the modules inhibit and activate each other, much like in the subsumption architecture, only that the links in an activation network are dynamic.

In order to be activated, a competence module must fulfill a set of conditions:

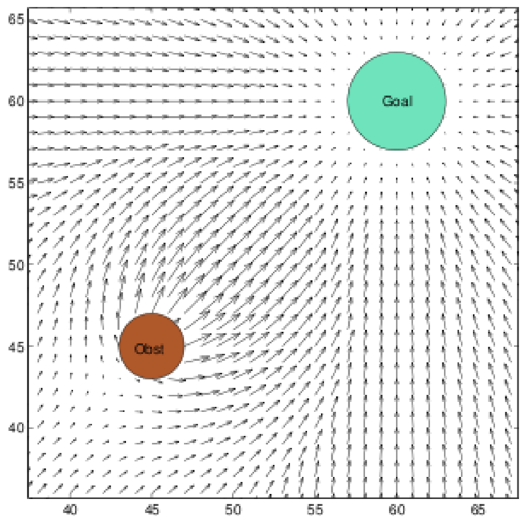
1. A set of preconditions must be fulfilled, allowing the module to execute. A `put-down-box` module could for instance have the precondition `is-carrying-box`.
2. Its activation level must be greater than a threshold.
3. Its activation level must be greater than any other module fulfilling 1) and 2).

Activation levels are calculated based on the weighted inhibition and activation from other modules. Action selection in an activation network is of the “winner takes it all” type, but the activation requirements will often favor actions that contribute to fulfilling multiple goals.

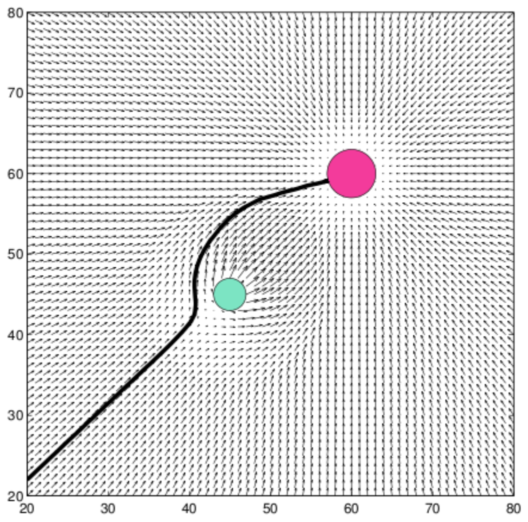
Potential fields

A potential field can be seen as an array of vectors. Hence, behavior output in a potential field type architecture is represented as vector fields. Action selection is done by combining all available vector fields, summing each vector.

Originally conceived as a method for achieving real-time obstacle avoidance in mobile robots (Khatib 1985), potential fields have later been used in combination with behavior-based schema systems (Arkin 1998, pages 141-165), and also for soccer playing robots (Laue & Röfer 2004).



(a) Combined potential fields



(b) Robot trajectory from start, via obstacle (blue) to goal (pink).

Figure 2.7: The potential field generated by an environment with an obstacle (repulsive force) and a goal (attractive force). Courtesy of Goodrich (2000).

Obstacles will generate a repulsive field, while the desired locations are attractive. The combination of two such fields is seen in figure 2.7. A robot controlled by a potential field architecture would at all times move in the direction of the vector at its current position. If it was to have the knowledge of the environment shown in figure 2.7(a), its trajectory could be as in 2.7(b).

2.2 Multi-agent robotic systems

A multi-agent system consists of more than one and up to thousands of robots operating in one environment. The individual agents are usually autonomous, and can perform a number of tasks such as soccer playing (Andre & Teller 1999) and garbage collecting (Arkin & Balch 1998). The systems differ in other areas as well, such as the composition of the group and the communication model used (and some systems have no communication at all). To easier classify the various systems, a taxonomy have been developed in amongst others Dudek et al. (1996).

Depending on the task to be performed, using more than one robot has a number of advantages. One is fault tolerance, if one in a 100 robots fail, there are still many left who can keep working. Some tasks will be performed faster by a group of agents than one, such as searching for objects (the more “eyes” the higher the chances of locating the object). Other tasks may be decomposed to be solved part by part by a team of robots. The main disadvantages are the risk of robots interfering with each other, and also the need of coordination, the complexity of which increases with the size of the group.

2.3 Artificial Neural Networks

Artificial neural networks (ANNs) are mathematical models based on biological neural networks. An ANN can be seen as a directed graph, cyclic or acyclic. The global behavior and abilities of the network are determined by the topology of the network, the strength of each individual connection (synaptic weights) and the node activation functions.

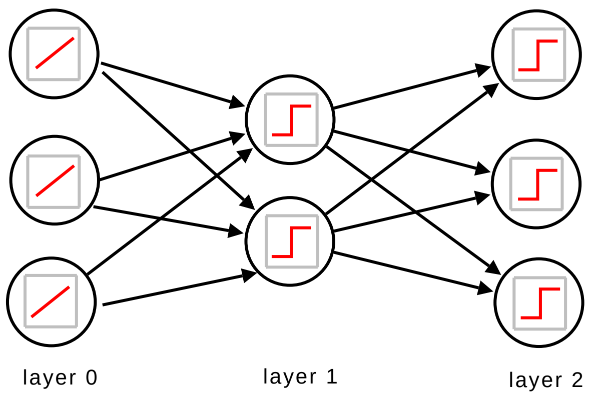


Figure 2.8: Illustration showing a feedforward ANN with one hidden layer. The nodes in the input layer have a linear activation function, while the hidden and output layers have step activation functions. All layers are fully connected with its next upstream layer.

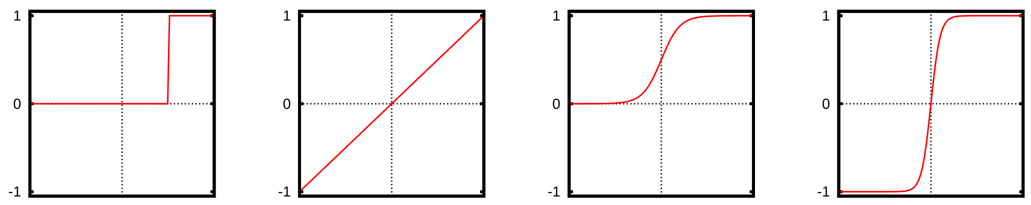
The most general neural network topology is the case where every node is connected to every node (fully connected neural network). All other network topologies can be represented as special cases of this topology, by setting some of the weights to 0. Due to the number of connections required, this type of network is difficult and time consuming to train, and is rarely used.

Instead, the nodes are often organized in layers, where nodes in a layer share certain connection properties.

The most common type of layered network used today is the feedforward network, where only connections from layer i to j are allowed, if $i < j$. By definition inter-layer connections are not allowed, and feedforward networks are always acyclic. Some neural network training methods, such as the backpropagation algorithm, work only on feedforward networks.

A neural network node has an input value (membrane potential in its biological counterpart), defined as the weighted sum of the output of the nodes connected to it, and an output value (activation level). The activation level is found by feeding the membrane potential into the activation function of the node. Figure 2.9 describes a few common activation functions.

A run through the network consists of feeding data into the first layer (the input layer), and then propagating it through the network through node to node connections.



(a) Unit step function, with a threshold $\theta = 0.5$

(b) Linear, $f(x) = x$

(c) Sigmoid, using the logistic function $\frac{1}{1+e^{-x}}$

(d) Sigmoid, using the function $\tanh(x)$

Figure 2.9: Some common neuron activation functions

First, it is necessary to calculate the weighted sum of all of inputs x to the node:

$$x_j = \sum_{k=1}^{N} w_{kj} o_k \tag{2.1}$$

where N is the total number of input links to j, w_{kj} is the weight from node k to j and o_k is the output of node k.

The output value is then:

$$o_j = f(x_j) \tag{2.2}$$

where $f()$ is the activation function of node j (see figure 2.9).

When summing over all j, equations 2.1 and 2.2 can be summarized by:

$$\mathbf{O} = f(\mathbf{W}^T \mathbf{X}) \tag{2.3}$$

In order to have the ANN learn a certain behavior, it is necessary to train it. This usually implies modifying weights in the network with a factor Δw:

$$w_{kj} = w_{kj} + \Delta w_{kj} \tag{2.4}$$

where w_{kj} is the weight from node k to j. There are several ways of calculating Δw, depending on the type of network and learning method used.

The differences between the different combinations of neural network building blocks are described in the next sections, by looking at a few common neural network architectures and their applications.

2.3.1 Perceptron network

The perceptron network is one of the simplest feedforward neural networks, it has low computational demands and it is able to learn quickly. It consists of two fully connected layers. The input layer passes values through to the output layer, where calculations are done (Rosenblatt 1958). The nodes in the output layer usually has a step activation function (see figure 2.9), resulting in a binary output pattern.

During training, pairs of input and desired output patterns are used. The input pattern is fed into the network, and its output is compared with the desired result. If they match, then nothing is done. If this is not the case, the weights of the connections between input and output layers are adjusted according to the delta rule:

$$\Delta w_{kj} = \eta(d_j - o_j)o_k \tag{2.5}$$

where η is the learning rate, d_j is the desired output of node j, o_j is the actual output, and o_k is the output of input node k.

If the patterns to be learned are linearly separable, the delta rule will eventually train the network so that it recognizes all of them. If, however, they are *not* linearly separable, the perceptron network will fail.

For instance, it is impossible having a perceptron network function as an XOR. Say one has a network of binary nodes, two in the input layer and one output node. The four possible input patterns that can be fed into the network are [0 1], [1 0], [1 1] and [0 0]. It is possible to train the network so that it responds with a 1 to [0 1] and [1 0]. But since there are no third two-dimensional combination that is linearly separable with both [0 1] and [1 0], the network will then respond with a 1 to *all* patterns presented to it, thus becoming rather useless. There is no way we can train it so that it also outputs a 0 to patterns [1 1] and [0 0].

2.3.2 Multilayer perceptron network

The multilayer perceptron is a modification to the perceptron network, with support for multiple layers. This is done by altering the way the network learns. The delta rule described above is not applicable to networks with multiple layers, since the desired output pattern is only known for the output layer, and not for the middle layers. Instead, multilayer perceptron networks make use of the backpropagation algorithm for learning (Rumelhart et al. 1986).

During training, the network is presented with pairs of input and desired output patterns, as described in section 2.3.1. For each pair, feed the input pattern through the network, calculating the output of each node according to equation 2.2, in the direction input → output layer. Then, calculate the error coefficient for each node in each layer, starting at the output layer:

$$\delta_j^{output} = f'(\sum_{k=1}^{N} w_{kj} o_k)(d_j - o_j) \tag{2.6}$$

$$\delta_j^{hidden\_n} = f'(\sum_{k=1}^{N} w_{kj} o_k) \sum_{k'=1}^{M} \delta_{k'} w_{jk'} \tag{2.7}$$

Where N is the number of input links and M is the number of output links in node j. $\delta_{k'}$ is the error coefficient of node k' (in the upstream layer), and $w_{jk'}$ is the weight from node j to node k'.

After calculating all error coefficients, the Δw's are found by:

$$\Delta w_{kj} = \eta \delta_j o_k \tag{2.8}$$

The above steps should be repeated until the correct output pattern is derived (or until the error is sufficiently small).

The multilayer perceptron is able to solve problems that are non-linearly separable. It has also been proven that a three layer perceptron network can approximate any mathematical function, given that the middle layer is sufficiently large (Cybenko 1989). The disadvantage is the number of calculations needed to learn a set of patterns. Each run through the network is more calculation-heavy than a classical perceptron network, and in addition to this, the multilayer perceptron needs more iterations due to slower learning.

2.3.3 Hopfield Network

Hopfield networks are a type of neural networks with *autoassociative* properties. The network is able to store patterns, and is able to retrieve the correct pattern from its memory when presented with noisy or corrupted input patterns.

Hopfield networks are single layered, and nodes are fully connected (however, the weight on the connection from a node to itself is set to 0). The network was first introduced by John Hopfield (Hopfield 1982).

When training a Hopfield network to memorize a set of N patterns, the weights are determined by:

$$w_{ij} = \begin{cases} \alpha \sum_{n=1}^{N} d_{ni} d_{nj}, & \text{if } i \neq j \\ 0, & \text{if } i = j \end{cases} \tag{2.9}$$

where d_{ni} are the ith part of training pattern n, and α is a scaling factor. The diagonal in the weight matrix (where $i = j$) is 0, since nodes in the Hopfield network are not connected to themselves.

During pattern retrieval, noisy patterns x are fed into the network, and output is calculated according to equations 2.1 and 2.2. This process should be repeated until there is no significant change to o.

In order to guarantee convergence, nodes must be updated asynchronously.

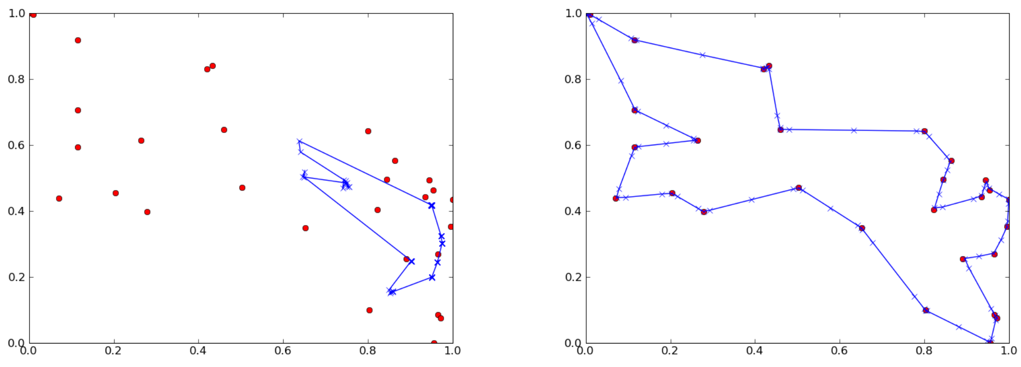
In (Hopfield 1982), it is shown that the storage capacity of the Hopfield network is $0.15n$, where n is the number of nodes. But for the patterns also to be *retrieved* without error, the storage capacity must be reduced to $\frac{n}{4 \ln n}$ (Amit 1992).

2.3.4 Maxnet and Self-organizing maps

The maxnet is a fully connected single layer network. A maxnet node has negative synaptic weights on all outgoing connections, except from the connection to itself, which is positive. The result is that the node with the highest initial value will remain high, while all other nodes will converge to 0.

Self-organizing maps, also known as Kohonen Maps, share the competitive behavior from maxnet networks. In each round, a winner node is selected. This node will excite itself and the nodes within a given radius in its neighborhood, the rest are inhibited. Usually, the radius starts out rather large, and gradually decreases for each round.

If the network is correctly tuned, the nodes will gradually order themselves to match the input pattern. Figure 2.10 shows a Kohonen network solving the traveling salesman problem.



(a) Initial setup (b) After 999 iterations

Figure 2.10: Kohonen network working on the travelling salesman problem of selected cities in Western Sahara. The full data set can be found on `http://www.tsp.gatech.edu/world/countries.html`

2.4 Machine learning

The ability to reason and improve itself over time are important aspects in an artificial intelligence system. This section will describe some important categories of learning techniques used in machine learning.

2.4.1 Supervised learning

Supervised learning methods make use of a detailed knowledge of the correct way to solve a problem. The system is presented with an input data set, together with the desired output. After the processing of each input, the system is evaluated and compared to the desired result. This feedback is used to tune the system, so that it will (preferably) perform the task better the next time.

Supervised learning methods are only applicable in situations where detailed knowledge of both the input and the desired output of the system is known, and is therefore not applicable to all situations. In a robotics context, supervised learning could be used to teach obstacle avoidance by sending a set of sensor data into the system, and have it accommodate itself to the desired motor commands.

The learning methods described in section 2.3.1 and 2.3.2 are examples of supervised learning.

2.4.2 Reinforcement learning

This type of learning can be described as learning by trial and error. After a sequence of actions, such as a run through a maze, the system is evaluated and feedback is given. The feedback is then used to improve the system.

Unlike supervised learning, the system is not constantly watched and evaluated, which allow for a more autonomous behavior. It is not necessarily so important *how* a task is done, as long as the result is satisfactory. Because of this, reinforcement learning is especially useful in robotics.

In Maes & Brooks (1990), reinforcement learning methods were used to teach the robot *Ghengis* to walk. The robot had 6 legs, and was controlled by a behavior-based architecture. Reinforcement learning was used to fine-tune positive and negative feedback in the system, determining the order in which behaviors were to be activated.

The difficult part when making use of reinforcement learning is the design of the evaluating system, the *critic*. The performance of the system must be translated into something measurable, and one has to make sure that reward and punishment are given in a way that will lead to the system developing in a positive direction (Kaelbling et al. 1996).

2.4.3 Unsupervised learning

Unlike the two above described groups of learning methods, unsupervised learning has no teacher to guide the system in the right direction. Instead, the system is presented with a set of input data, and is to classify the data based on similarities.

Unsupervised learning is usually faster than supervised learning methods such as the backpropagation algorithm (see 2.3.2), and is therefore applicable to real-time systems.

Unsupervised learning is frequently used in neural networks, such as the Kohonen and Hopfield networks described in sections 2.3.4 and 2.3.3. In a robotics context, unsupervised learning can be used to cluster sensor input. In Nehmzow (1999), the robot *FourtyTwo* was able to recognize and move towards boxes by using unsupervised learning and Kohonen networks.

Hebbian learning

Hebbian learning is a commonly used form of unsupervised learning, based on the theories of the American psychologist Donald Hebb on neural network learning. The essence is that if neuron A often contributes to activation of neuron B, the synaptic weight from A to B should be strengthened.

In its original form, known as *Hebb's Rule*, this can be written as:

$$\Delta w_{kj} = \eta o_k o_j \tag{2.10}$$

where o_k is the output of node k and η is the learning rate. A problem with equation 2.10 is the potential for infinite weight growth (or decrease) and network saturation.

One way to prevent this from happening is to introduce a non-linear forgetting factor, as done in the *Oja* rule:

$$\Delta w_{kj} = \eta(o_k o_j - |w_{kj}| o_j^2) \tag{2.11}$$

By using w_{kj} as a part of the forgetting term, the synaptic weights' abilities to increase to infinity is removed, while still preserving the "fire together, wire together" philosophy of Hebb's Rule.

The Hopfield networks from section 2.3.3 are examples in which Hebbian learning is used, mainly equation 2.10 with some minor modifications.

2.4.4 Imitation learning

As the name implies, the main focus of this technique is to have the system learn by observation. In robotics, imitation learning can be utilized by having one robot perform the task that is to be learned, being controlled by a human operator, while another robot observes. Another way to do this is to record sensor values and actuator commands and later use these for learning in other robots.

Depending on the approach and type of task to be learned, imitation learning can belong to either three of the categories described above.

2.5 Emergence and self-organization

Self organization is the appearance of structure or pattern without any external or centralized forces imposing it (Heylighen 1999).

In nature, there are several examples of self-organizing systems, spontaneous magnetization is one of them. The magnetic field emerging from a material is determined by looking at its sum of particle spins. Spontaneous magnetization occurs when the material is cooled down to a threshold temperature[3]. Below this temperature, the particle spins will align themselves in a certain direction.

Some general characteristics of self-organizing systems include robustness and fault-tolerance. All parts of the system contribute equally to the global state, no one is more important than the others. So even though a few parts break down, the system will continue to function, adapting to the changes. In the example above, a particle spin can escape from its stable position, but the other spins will quickly force it to realign itself.

When global properties of a system arise out of the local interactions in a system it is called *emergence*. Emergence and self-organization are often closely coupled; self-organizing systems often have emergent properties, such as the magnetic field in the example above.

Emergent properties often arise in behavior-based (and reactive) control systems. In Maris & te Boekhorst (1996), a group of simple robots are programmed to do obstacle avoidance in an environment filled with boxes. The robots are equipped with left and right IR sensors and bumper sensors (which are only activated after crashing into objects). The robots will then avoid obstacles detected by the side sensors, but crash into obstacles located directly in front of it. The pressure from one box is not high enough to activate the bumper sensors, resulting in the box being pushed until it hits another obstacle, such as another box. After a while, the boxes will be arranged in clusters, as can be seen in figure 2.11.

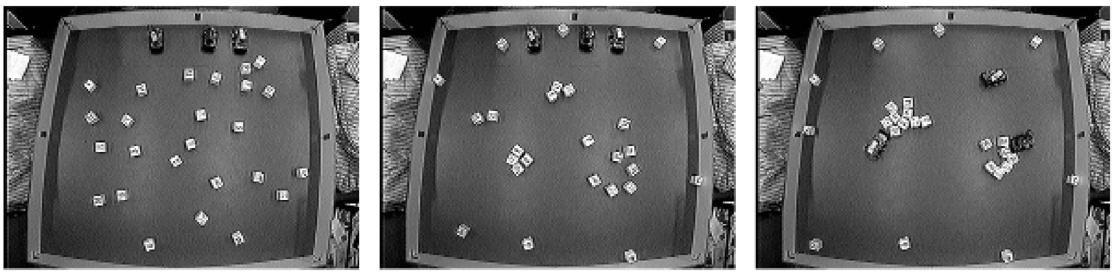


Figure 2.11: Clustering of boxes by DidaBots. Courtesy of the Artificial Intelligence Lab, University of Zurich.

[3]Also known as the Curie temperature, a material-specific constant. For iron, the Curie temperature is 768^o C

3 Design and methodology

This chapter will describe the system created and give a short introduction to the e-puck robot and its sensors and actuators.

During development, emphasis has been on creating a flexible system, where functionality can easily be added. The system is object oriented, and all code has been written in the Python programming language[1].

An introduction to using the system with Webots can be found in appendix B, and the structure of the code repository is described in appendix C.

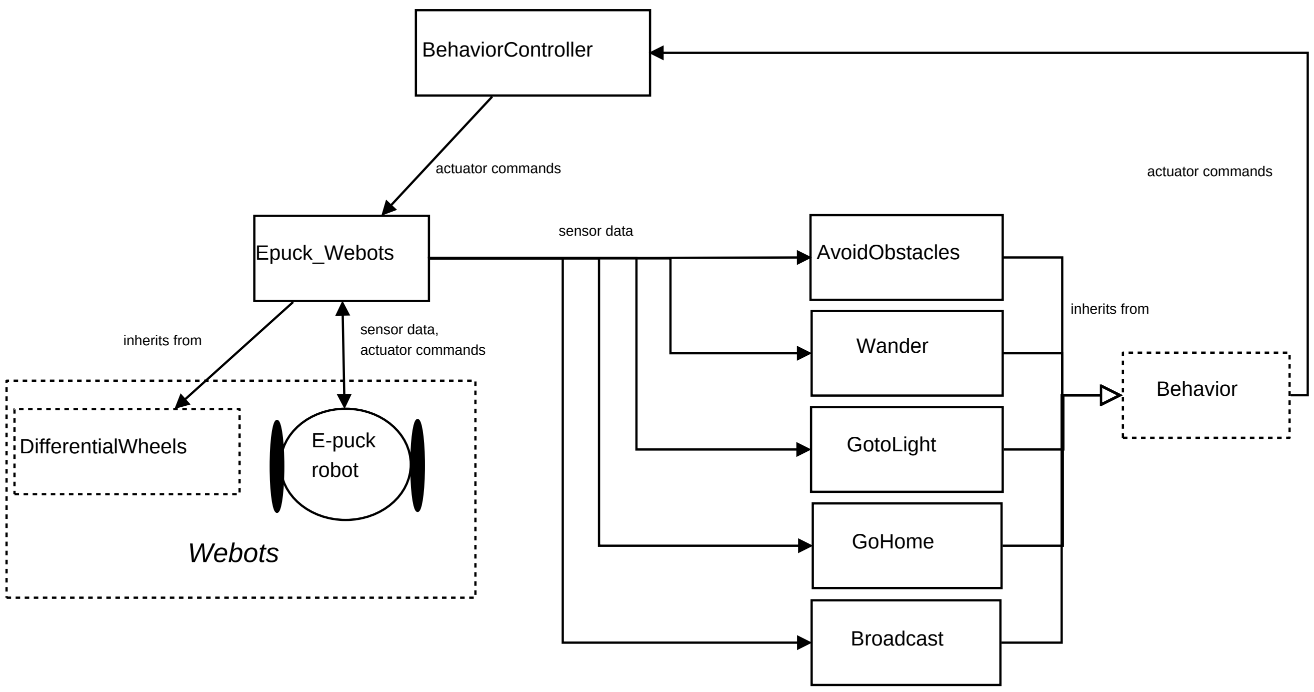


Figure 3.1: Overview of the system, indicating the most important flows of data.

[1]`http://www.python.org/`

3.1 E-puck robot

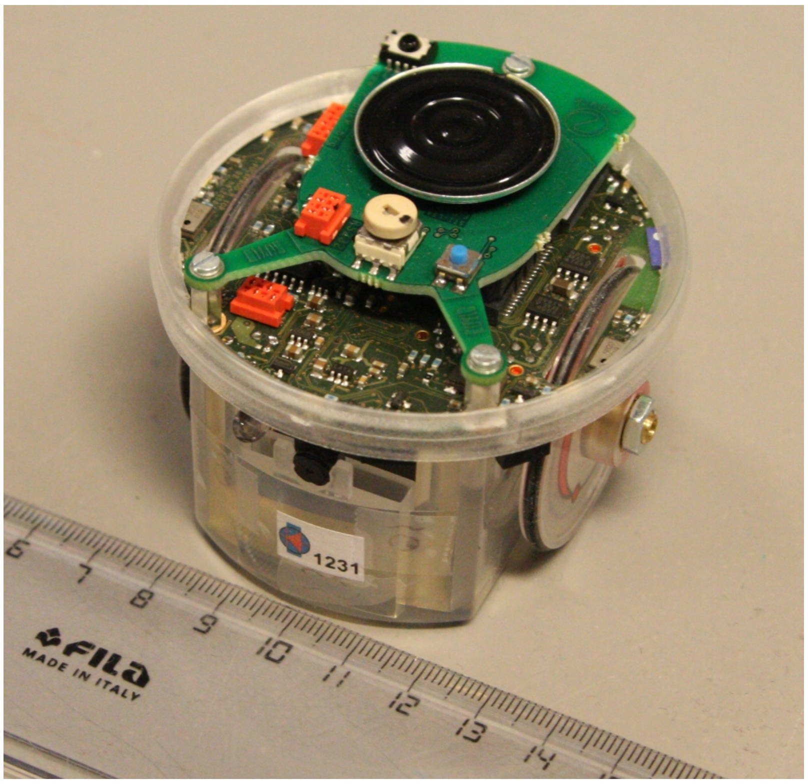


Figure 3.2: The e-puck robot

The e-puck[2] robots used in this project have several sensors and actuators used to interact with their surroundings:

- Eight infrared (IR) proximity sensors, distributed around the body. See figure 3.3.
- A color camera, with a 640x480 resolution. Due to hardware limitations, only a 40×40 part of the picture can be acquired at a time. When using color images, a frame rate of 4 fps is possible. For gray-scale images, the frame rate is doubled.
- Three microphones. See figure 3.4.
- 3D accelerometer.
- Two wheels, with the ability of forward and backward motion.

[2]`http://www.e-puck.org/`

- A high-intensity red LED, located directly above the camera, that can be used for tracking purposes.
- Red LEDs, distributed around the body, to be used for interaction (with user or other robots).
- A speaker.

In addition to this, the robot has a standard RS232 (serial) interface, infrared receiver and a bluetooth controller used for communication between robot and computer. The bluetooth link can also be used for robot-robot communication. In this project however, only computer-robot communication has been used.

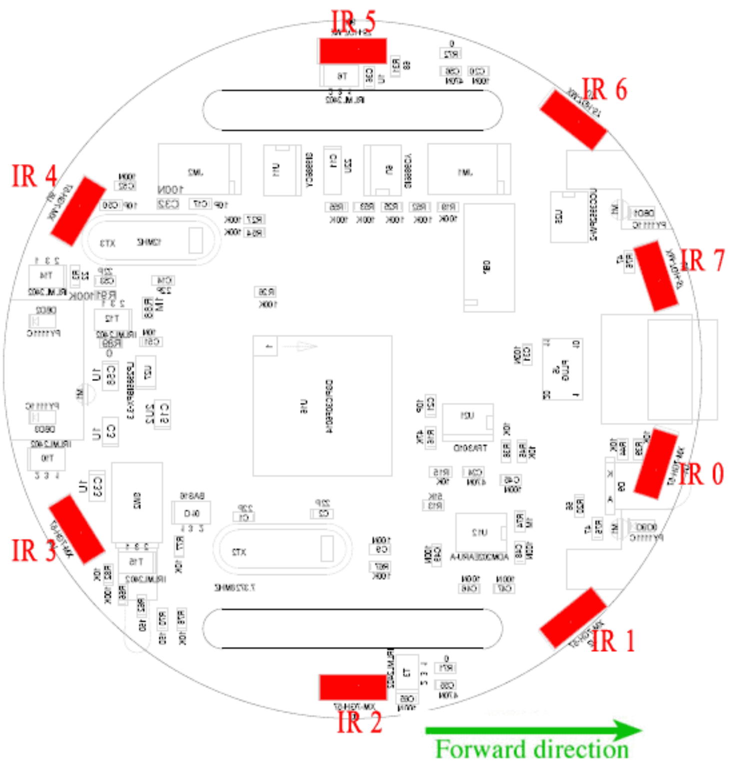


Figure 3.3: Diagram of the e-puck, with 8 IR proximity sensors. Courtesy of `http://www.e-puck.org/`

The e-pucks are able to run any kind of customized software (within the limits of its hardware), but for this project the e-pucks have been running with the SerCom[3] software. This software enables the user to acquire sensor input, control motor

[3]Downloadable from `http://www.e-puck.org`

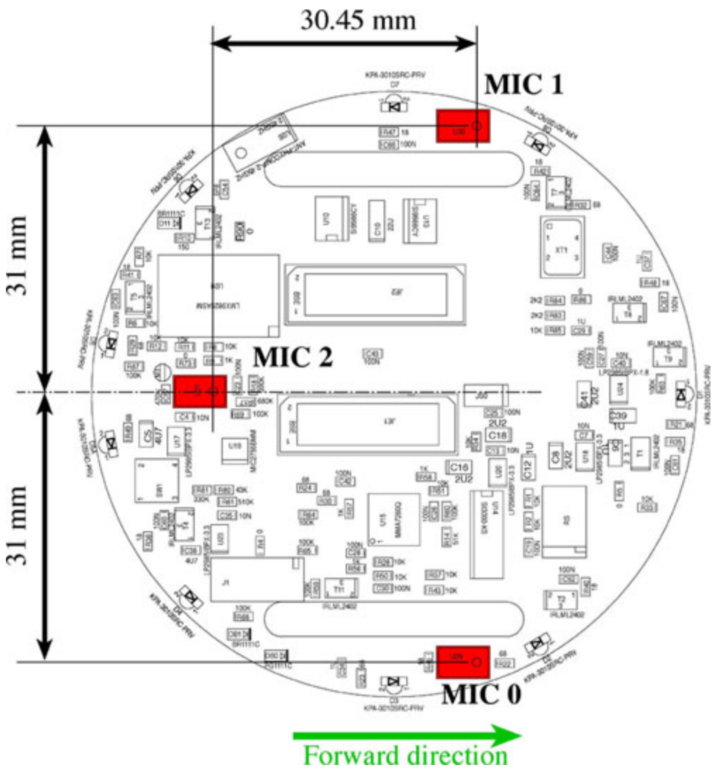


Figure 3.4: Diagram of the e-puck, showing the location of the three microphones. Courtesy of `http://www.e-puck.org/`

speeds and more via a bluetooth/serial connection to a computer. For a full list of supported commands, see appendix D.

Low-level communication between the computer and e-puck is handled by the Webots simulator, enabling the user to easily switch between running the software on simulated and real e-pucks.

3.2 Physical environment

When running the system using physical e-puck robots, they have been operating on a 124 × 152 cm table, with a 10 cm high edge (to avoid the e-pucks from falling down). Both the edges and the table surface are painted in a light gray shade (the surface is painted slightly darker than the edges).

Additionally, colored boxes are spread around the table. Red boxes are used as an indication of "home", while the others are being used as obstacles. All materials are made out of wood.

For light detection, an ordinary desktop lamp has been used.

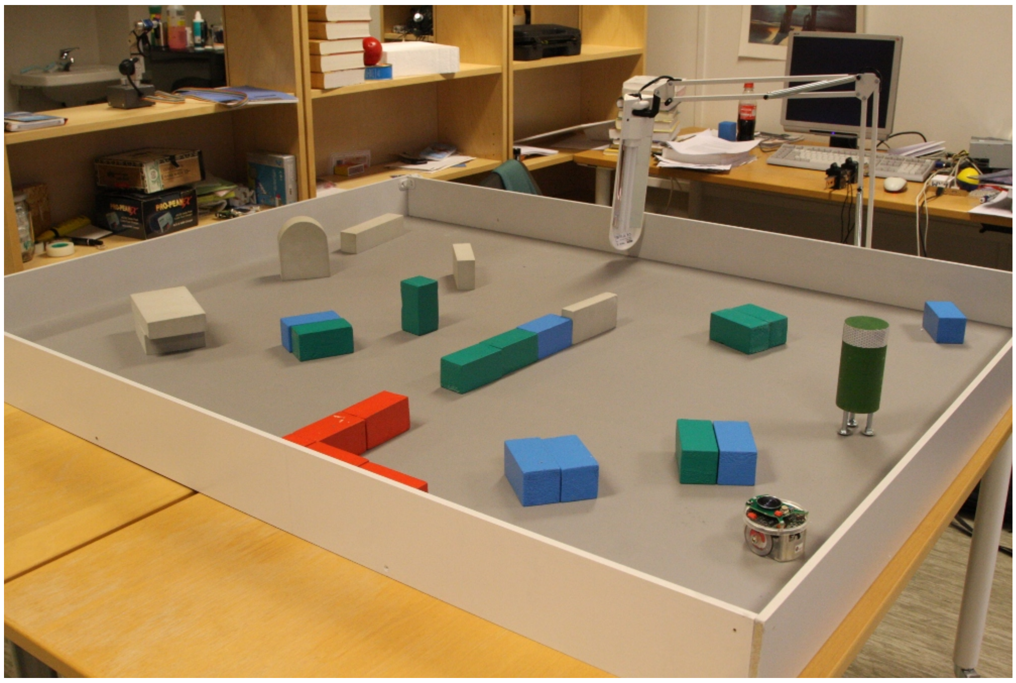


Figure 3.5: The table on which the e-pucks have been running. The collection of red boxes on the left side is the "home".

3.3 Webots

Webots[4] is a robot simulator with a built-in software implementation of the e-puck robot and also some other popular commercial robots such as the Khepera and Aibo. Webots also has the capability of controlling physical robots, so that with just a few mouse clicks it is possible to switch between simulated and physical robots.

In the previous version of this project e-puck-computer communication was handled by a "homemade" solution using the serial communication library in Python. Although this solution to a degree supported simulation through Breve[5], Webots makes it much easier to switch between simulated and physical e-pucks. This is the main reason Webots have been used for low-level e-puck communication in this version.

[4]Webots: `http://www.cyberbotics.com`
[5]The Breve simulation environment: `http://www.spiderland.org/`

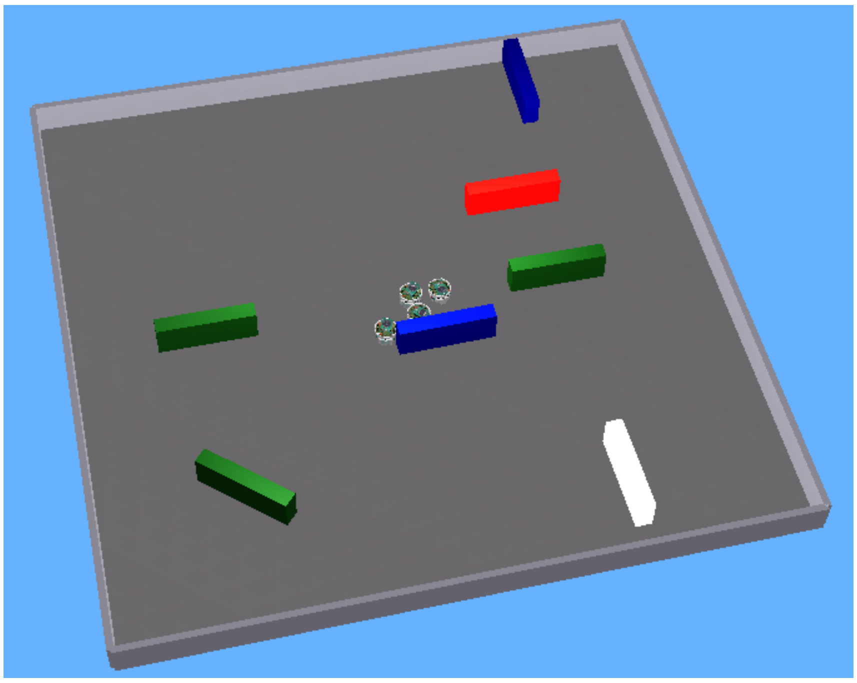


Figure 3.6: The simulated e-puck environment in Webots with four e-puck robots. The red box is “home”, while the white one represents “light”.

The simulation environment created in Webots (see figure 3.6) is very similar to the physical environment described in section 3.2, but for practical reasons the width of the boxes is larger.

The e-puck robots in Webots do not have exactly the same properties as the physical robots. Not all physical sensors and actuators have been implemented, for instance microphones and the speaker are only available as experimental features. In addition to this, the simulated e-pucks have been given a compass and a signal emitter and receiver, making it possible to experiment with localization and cooperation when using more than one e-puck simultaneously. Webots supports simulated radio, infrared and serial transmissions, and for this project radio signals have been used.

These improvements are in no way essential to the system, and most of the behaviors described in section 3.7 will run on both simulated and physical e-pucks.

The practical issues with using Webots will not be further discussed here, but appendix B.2 gives a short introduction to using Webots with e-puck robots.

3.4 ANN Generator

In order to be able to easily create and test out different neural network configurations, an ANN generator has been created. The generator takes in all necessary information as parameters, and from this creates an `ANN` object. Currently, the learning rate, execution order of the layers, number of training rounds and the maximum number of rounds to execute each layer before breaking must be set by the user. It is also necessary to supply the generator with information on all links and layers in the network (number of nodes, arc topology, learning functions and similar).

The easiest way of providing this information is through a text file which can be parsed into a form recognizable by the ANN generator. The syntax currently supported by the parser is best explained by looking at some setup files, appendix E provides a few examples.

The ANN class has native support for unsupervised learning, which learning function to be used can be specified for each link in the setup file. When invoking the ANN object's `train()` method, the training patterns are run through the network one by one, and learning is activated for all plastic links.

Supervised learning is not built into the ANN class, but can be done by passing the ANN object as a parameter to the supervised learning functions. Currently, delta and backpropagation learning is supported.

It is possible to save and load the ANN object to/from a text file, making it possible to store well-performing networks for later use without having to re-train them. One can also add new links at run-time, either with randomly initialized weights, or with a user defined weight distribution from a text file. Saving the weight configurations to a text file is also supported.

Appendix B.4 gives a short introduction to using the ANN generator.

3.5 The behavior control system

The behavior control system is responsible for coordination and execution of behaviors on the e-puck. It consists of a behavior controller, and several pre-defined behaviors. Each behavior attempts to reach their own goal, and the job of the controller is to determine which action to take based on the input from the behaviors.

3.6 The behavior controller

The behavior controller implemented in this project uses a type of command fusion loosely inspired by DAMN (see 2.1.5). The controller receives input from the behaviors, in the form of *suggested moves*. A *suggested move* is simply a weighted list of eight directions, the directions are numbered as shown in figure 3.7.

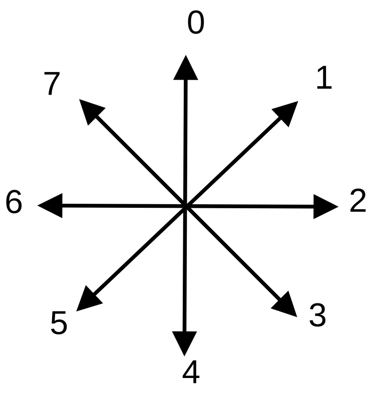


Figure 3.7: There are eight possible directions for the e-puck's next move. 0 is defined as no change from current direction, while 4 is a 180° turn.

Each time the controller is executed, the suggested moves are summed into one list, and the direction with the highest sum is chosen. This ensures that input from all behaviors are taken into consideration, and the direction that suits the most behaviors will be selected. The scaling factor of a list determines its priority. For instance obstacle avoidance outputs large negative numbers in directions with obstacles, while random wandering have a scaling factor of < 1 and will thus output small numbers.

In most situations, it is preferable to have as small changes in the direction as possible. Therefore, the weighted list

```
[0.5,0.25,0,-0.25,-0.5,-0.25,0,0.25]
```

is appended to the list of moves. The weights are so small that the forward direction will not be chosen in situations where there are reasons *not* to keep moving forwards, while still large enough to prevent situations where the robot wanders aimlessly back and forth.

Not all behaviors will be executed in each time step, section 3.7 will talk about this in greater detail.

It is possible to run the behavior controller in both parallel and sequential mode[6]. Externally, there should not be any major differences between the two, besides the time between execution of behaviors. The threaded version uses time (in seconds) to determine how often a behavior should run, while the sequential version uses an internal counter.

3.7 Behaviors

Each behavior is able to execute on the robot without being dependent on any other behavior. Just like the behavior controller, behaviors can be run either in parallel, by using threads, or sequentially.

Externally, a behavior should communicate only with the e-puck and the behavior controller. Behaviors are able to communicate with each other, but only via the controller. Sensor data from the e-puck is collected by each behavior as needed. Behavior output (typically suggested moves) is usually sent to the behavior controller and not directly to the e-puck. There are some exceptions, but these will be discussed below.

At creation, a behavior should be assigned a sleep-time and a scaling factor. The scaling factor is used for adjusting the weighted list of suggested moves so that high priority tasks such as obstacle avoidance will always produce higher weights than for instance random wandering.

The sleep-time is used to determine how often a behavior should execute. Setting a sleep-time is necessary for two reasons. The most important one is that there is a limit on how much the e-puck is able to accomplish in each time step. Some behaviors, especially those in need of camera data, take several seconds to finish on a physical e-puck (and somewhat less time on a simulated e-puck). And there is no real need for the e-puck to use the camera vision in each time step. If no red blocks were visible half a second ago, there probably won't be any around until the e-puck has been able to move a distance away.

The second reason is related to thread starvation. With all threads continuously competing for execution time, chances are that a few behavior threads will be run

[6]Due to the Global Interpreter Lock (GIL) in Python, the so-called parallel version is actually not very parallel. The main difference is that the threaded version executes the behaviors in an "intertwined" fashion, whereas the sequential version executes them one by one. For low-priority behaviors with a long execution-time, the threaded version will allow the higher priority behaviors to run more frequently. It ought to be possible to circumvent the GIL by using C-extensions in Python, but so far I have not needed "real" parallelism in the system.

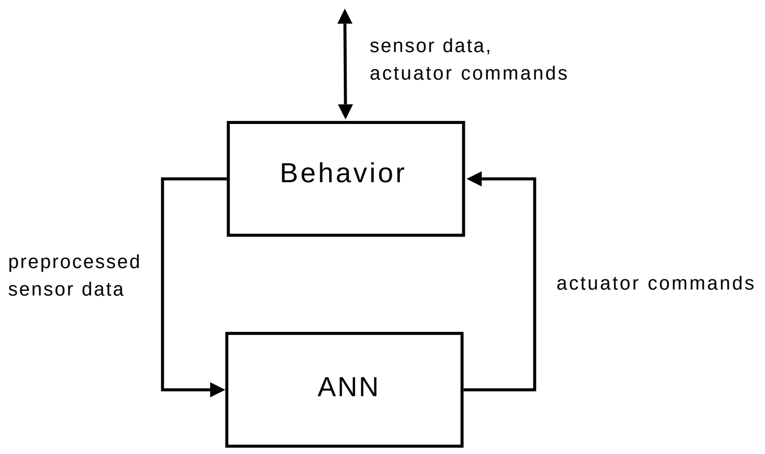


Figure 3.8: Close-up look at a behavior from figure 3.1, controlled by an ANN. Typically, the behavior does some form of preprocessing of the data in order to transform it to a list structure recognizable by the ANN. Then, the ANN calculates an appropriate actuator command based on the input, which is passed on to the behavior controller.

consequentially, while others will almost never be allowed to run[7].

The internal structure of a behavior is irrelevant to the system, as long as the above-mentioned requirements are met. One possible internal organization is using an ANN, as in figure 3.8. This approach has been used for obstacle avoidance in this project. Other behaviors, such as random wandering, is fairly simple, and have no need for an external controller.

The next few sections will take a look at the behaviors created to date.

3.7.1 Obstacle avoidance

The e-puck robot is fairly lightweight, and has a plastic ring around its body protecting it from impact damage. Still, crashing into obstacles is generally not a sought-after behavior. Therefore obstacle avoidance has been implemented as the most basic behavior. The sleep-time is set to 0.1 seconds, and the scaling factor is

[7]Thread handling is the responsibility of the operating system (OS) scheduler. Theoretically, it is possible to adjust the default priority of the threads (on the OS level). But Python threads must obtain the GIL before being allowed to run, and the GIL takes no notice of OS thread priorities. Additionally, the GIL seems to run the threads in "bulks" (5 times obstacle avoidance, then five times random wandering and so on), for no apparent reason. So adjusting thread priorities does not really have much effect in reality. Having the threads sleep in-between execution reduces the competition for obtaining the GIL and results in a much smoother-running system.

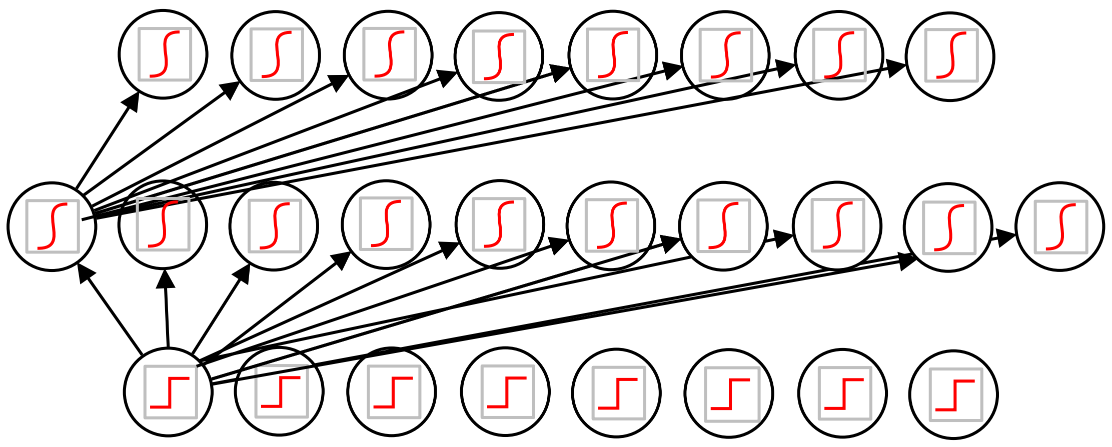


Figure 3.9: The ANN used for obstacle avoidance. All layers are fully connected with the next downstream layer, but for simplicity reasons only the links originating from the first node in each layer are drawn.

8, minimizing the risk of other behaviors overriding obstacle avoidance if there are obstacles present.

This behavior is governed by the three layer feedforward network seen in figure 3.9, which takes input from the e-puck proximity sensors. The network is created with the ANN generator (section 3.4), and both the setup file for the generator and the training data is included in appendix E.

The network has eight input nodes, one for each IR sensor, 10 hidden nodes and 8 output nodes. The output nodes represent the eight directions seen in figure 3.7. Each layer is fully connected with the nodes in the next downstream layer. The output is typically zeros in directions with no obstacles, and large negative numbers where there are obstacles.

3.7.2 Searching for light and homing behaviors

The homing and searching for light behaviors have been designed to be executed successively, only one is able to be active at a time. Upon reaching a success state the active behavior will pause itself, and activate the other.

Both behaviors capture images from the e-puck, process them, and transforms them into an array structure representing regions in the image (see figure 3.10). One complete run of the behavior consists of 4 or 8 images captured while the e-puck rotates around its own axis in 22.5 or 45 degree steps.

Splitting the image into regions is useful mainly for two reasons. First, it allows the

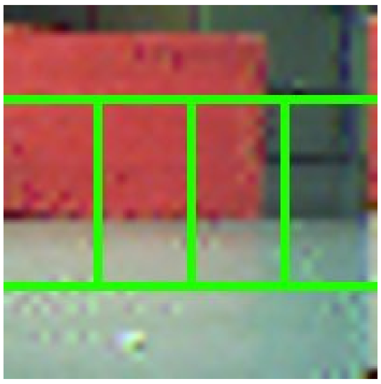


Figure 3.10: A test image taken from the camera of a physical e-puck, the green lines showing how the image is split into 4 regions, disregarding information in the upper and lower part of the image. The image has been scaled up from 40 × 40 pixels. Region height and width are user-specified.

e-puck to see where in its field of vision an object is located. Secondly, not all areas of the image contains useful information. In general, the lower part of the image will never be filled with any objects (other that the floor) unless the e-puck is very close to the object. And then the image will typically be very dark/black due to the lack of light between the object and the camera, hence it is not possible to gather much knowledge from it.

Light

An ordinary desktop lamp (see figure 3.5) represents light in the physical world, while in the simulation a white box has been used.

The camera of the physical e-puck will to a certain degree automatically set the white balance of the image, the result is that white surfaces will often look slightly grayish in pictures. So the only part of the images with a large concentration of white (and near-white) pixels are light sources. By counting the number of pixels the e-puck is able to determine the direction and distance to the light source.

In the simulated world, white surfaces are still looking white in pictures. So for this behavior to work as intended in the simulation, all white surfaces except the one representing the light must be removed.

Homing

A form of homing behavior has been implemented by defining home as "where the red boxes are", which is localized by making use of camera vision. For this behavior to function properly, all red objects not a part of the home-area must be removed from the table.

There are several methods that can be used for detecting red objects, the performance of a few of them will be discussed in section 4.5.

The method that has proven to be most appropriate for the homing task is to use the ratio between the red and green components of a pixel.

3.7.3 Signal localization

This behavior exists in two versions, one using sound and another using the signal emitter on the simulated e-puck.

Localization in the physical world

The version where sound is used makes use of the the three microphones on the e-puck (figure 3.4) for determining the direction to a sound source. In order to accomplish this, the sound localization behavior makes use of an ANN with maxnet functionality (see chapter 2.3.4), created by the ANN generator.

The network has two hidden layers, in addition to an input and output layer. The input layer has three nodes, one for each microphone, and two output nodes, representing the left and right e-puck wheel.

The input layer has a positive linear activation function, and simply passes the input on to the next layer. The first hidden layer is where the main functionality of the network is. Each node has a strong positive connection to itself, and a weak negative connection to the two other nodes. After a number of iterations, the node with the highest input will "win", while the value of other two nodes will go towards zero.

This result is sent to the next layer, which filters the values through a step activation function. The filter and output layers are fully connected, and has manually set weights. A [1] on the node representing the left microphone translates to direction 6, and the opposite for the right microphone. If only the node representing the back-microphone is non-zero, the output is $[0, 0]$, or direction 0. This is because of the amplitude differences described in section 4.1.

The setup file and weight data used to create the network can be found in appendix E.

Localization in the simulation world

The simulated e-puck does not have functioning speaker and microphones[8], so in order to also be able to perform signal localization in the simulated world a radio signal emitter and receiver have been added to the e-puck. Two separate layers are used, `Broadcast` and `GotoSound`. `Broadcast` simply broadcasts a data packet each time it is run, while `GotoSound` reads and processes received packages. It is possible for a robot to have both layers active, but it will only be able to receive signals from other robots, not from itself.

Signals are transmitted as discrete data packages and are stored in a FIFO[9] type queue structure by the receiver, as shown in figure 3.11. The Webots API guarantees atomic transmission of packages, and that they will be received in the order in which they were send by the emitter, but no promises are made with regard to scheduling. So a packet may be received at different time steps by two receivers, but always in the correct order.

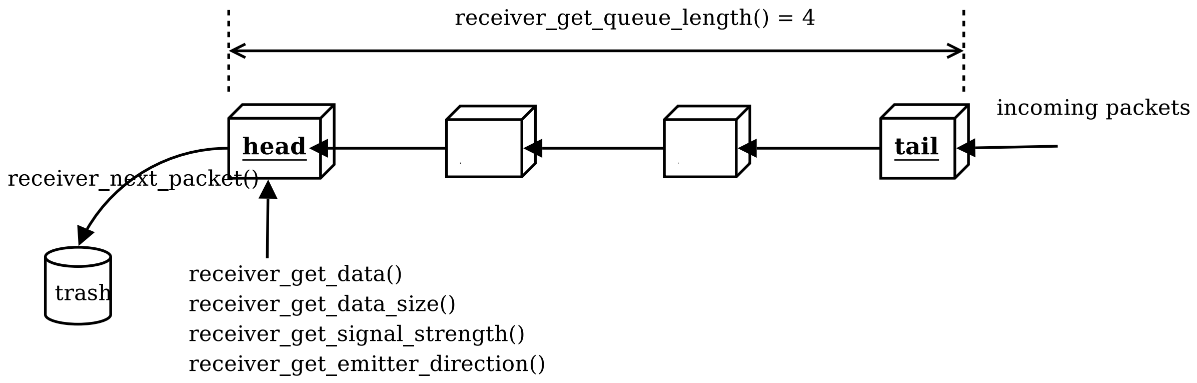


Figure 3.11: Illustration of how data packets are received and processed by a signal receiver. Courtesy of the Webots Reference Manual available at `http://www.cyberbotics.com/cdrom/common/doc/webots/reference/reference.html`.

The signal receiver has a built-in localization feature, returning the direction of the signal relative to the coordinate system of the receiver. The direction is given as a vector normalized to length 1.0, with the z-axis pointing in the backward direction of the receiver, and the x-axis to the right[10]. As in the global coordinate system, the

[8]Sound functionality is implemented as an experimental feature, and I at least have not been able to make it work properly.

[9]FIFO - First In, First Out

[10]The Webots reference guide states that the opposite is correct, that the z-axis points in the forward direction and the x-axis to the left, but in my simulation world this is not the case.

y-axis defines height and is of no relevance for this simulation world. So an emitter located approximately in front of the receiver would result in a vector similar to $\begin{bmatrix}-0.098 & 0.0002 & -0.995\end{bmatrix}$.

The vector is then used to determine the direction in which the e-puck must move to travel to the broadcasting robot.

3.7.4 Random wandering

Random wandering is there to enable the e-puck to change its direction when no interesting sensor input is received. The logic within the behavior is fairly simple, a random integer between 0 and 7 is drawn, representing one of the directions from figure 3.7. Wandering only makes sense when no other behavior gives any interesting output, so this behavior should have a scaling factor below 1. Experimentation shows that 0.1 is an appropriate value.

3.8 Implementation details

As mentioned in the chapter introduction, the system(s) created in this project are modular, object oriented and fairly independent of each other. The behavior control system can control any type of robot, not just e-pucks, with little or no modifications needed (and of course software taking care of low level communication will be needed for the other robot if Webots can not be used).

Modifying the framework is also relatively easy. To add a new behavior controller or behavior, subclasses must be added to respectively `BehaviorController` and `Behavior`. Subclasses should as a minimum implement their own version of `run ()`. Some behavior controllers will also need their own `initialize ()`.

Adding functionality to the e-puck and neural network generator frameworks is also done for the most part by subclassing (or simply by adding new class functions). For an overview of the code structure, see appendix C.

4 Results

This chapter will explore the capabilities of the behavior control system when exposed to different situations. It will start by looking at the e-puck and its sensors, and then continue on to adding behaviors to the system, one by one.

Unless otherwise is stated, all results concerning physical robots in this chapter have been obtained using the e-puck "Minch" connected to a computer via bluetooth.

4.1 E-puck sensor measurements

4.1.1 Physical e-puck

As described in section 3.1, the e-puck has several sensors and actuators. Each sensor has its own "personality", and this chapter will look at how the e-puck responds to different situations.

Figure 4.2 shows the response of all eight IR sensors to a light gray object. As can be seen from the figure, there are some differences as to how well the eight sensors respond to objects. Sensors 1 and 6 are located somewhat closer to the edge of the e-puck body than the other sensors (approximately 1 mm from the edge, versus 2 mm for the rest of the sensors), but it does not appear that this has any major impact on the results. There are no other visible physical differences between the sensors that can explain their output.

At distances greater than approximately 7 cm, noise becomes too much of a problem and the proximity sensors no longer give any useful output.

There may also be some problems related to the amount of IR radiation reflected from different objects. Some types of objects will absorb more IR than others. Assuming the IR radiation measured by the proximity sensors has a wave-length located just outside the visible light spectrum, it is logical to assume that dark objects will absorb more IR radiation than white objects.

The amount of reflected IR is also dependent on which material the object is made of.

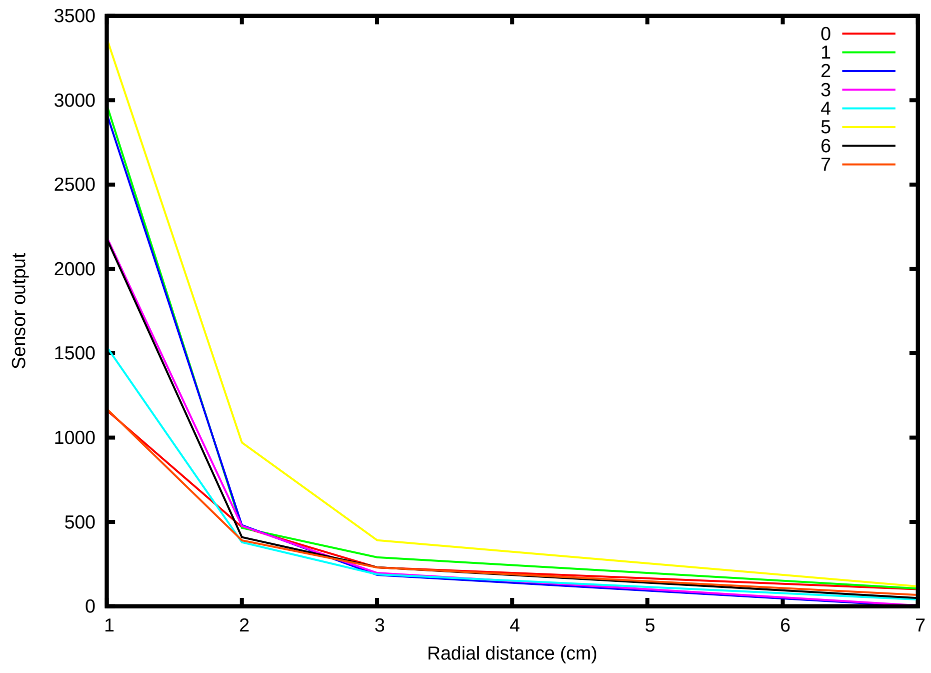


Figure 4.1: Diagram showing the values of the eight proximity sensors, as a function of radial distance from the e-puck body, when detecting a light gray object. Measurements are done at distances 1, 2, 3, and 7 cm from the e-puck body.

Figure 4.2 shows the differences between light gray, white, red and blue objects. The graph shows some minor differences in measured IR radiation from the objects, but as long as the sensors are used for proximity detection and not distance measurements this should not cause much problems. As expected, the sensors detect overall more reflected IR waves from the white object than from the other ones.

The three microphones on the e-puck (see figure 3.4) can be used to find the direction of a sound source. However, there appears to be some timing issues with the amplitude measurements of the microphones. When using sound sources that emit sound with a lot of amplitude and frequency variations, such as mobile phone ring tones or music, localization becomes difficult. Also, the microphones respond differently to different sound frequencies, as can be seen in figure 4.3. As to why this happens, I am not sure. One explanation is that either the speakers used[1] or the computer's sound card[2] is not able to accurately create and emit all sound frequencies with the same amplitude. Another theory is that the e-puck microphones

[1]A set of MidiLand MLi-150 stereo computer speakers.

[2]Integrated Intel HDA card on a Dell Latitude D420 laptop.

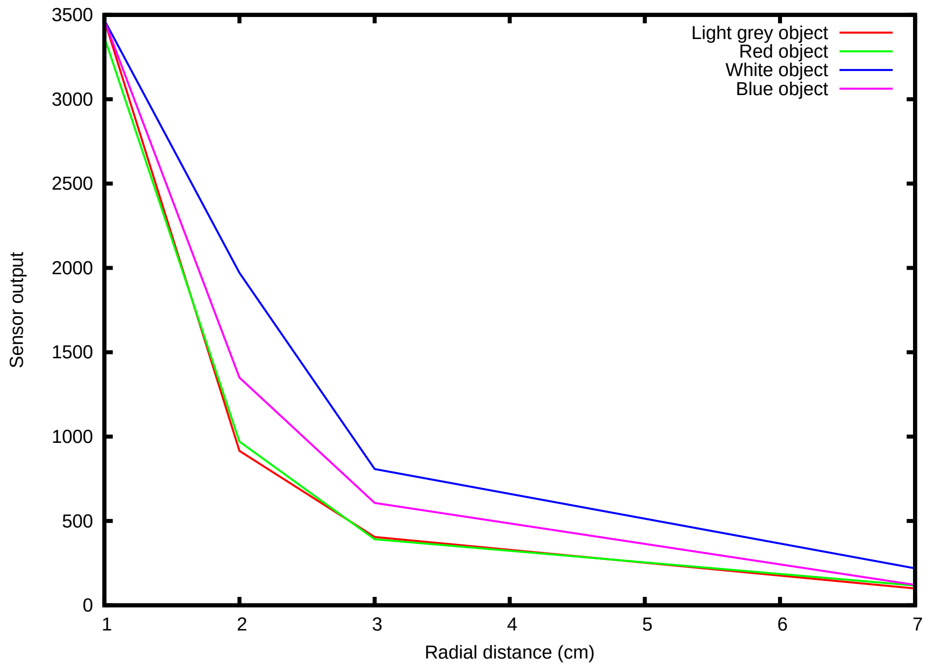


Figure 4.2: Diagram showing the values of one of the proximity sensors (sensor 5), when detecting a red, blue, white and light gray object. Measurements are done at distances 1, 2, 3, and 7 cm from the e-puck body. The white object is a sheet of paper, while the three others are made of wood.

are more sensitive to certain frequencies that others. However, no further testing with other equipment has been done.

Experiments indicated that the lower sound frequencies had a shorter range, and they were also more easily distorted by obstacles and walls. However, even though high frequency sounds yield better results, they are rather painful to listen to. A uniform sine wave with a frequency of 7-900Hz seems to be a fair compromise between the painfulness of the sound and how well the e-puck is able to detect its direction.

Also, the back microphone generally reports higher amplitudes than the two side microphones. This should be taken into consideration when comparing the three values.

As mentioned in chapter 3.1, the e-puck camera is only able to acquire images of size 40×40 pixels at a time. The image position can be moved around in the 640×480 pixel area covered by the camera, but for this project only the default image position has been used, at the camera's center. The horizontal viewing distance of the e-puck is approximately 6 cm for an object located 20 cm from the e-puck body, and

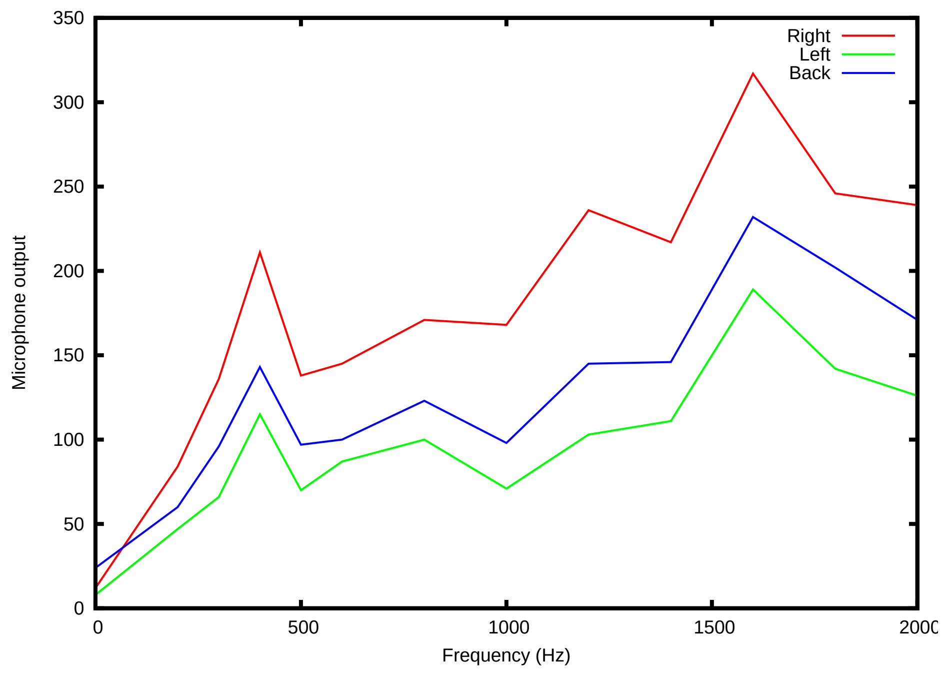


Figure 4.3: Diagram showing the microphone responses to sine waves with variable frequencies. The speaker is located approximately 15 cm from the right microphone.

3-3.5 cm for an object 10 cm from the e-puck. This results in a viewing angle of approximately $17^o - 20^o$.

Another observation is that the behavior of the e-puck is somewhat dependent on the battery level. The IR sensor values are especially sensitive to this, the more discharged the battery is, the more noisy the IR measurements are. The measured values do tend to be overall lower than they are when the battery is fully charged.

The same goes for the camera, image data seems to be lost (replaced by noise) more frequently when the battery charge is low.

Also, the e-puck sometimes refuses to respond to motor commands when the battery level is low.

The microphones does not seem to suffer from this, the noise level is fairly low, and independent of battery level. The software created for this project tries to accommodate for this by repeatedly probing the sensors when data is lost, and to output warnings to the user.

But still, the conclusion is to always (when possible) use the e-puck with a fully charged battery, and to never trust any results obtained with a low battery level.

4.1.2 Simulated e-puck

The simulated e-puck is a fairly good approximation of the physical e-puck, and functions in more or less the same way.

As can be seen from figure 4.4, the proximity sensors of the simulated e-puck does not appear to have the differences in response seen in the physical e-puck (see figure 4.1). This is as expected, there would have been no reasons for implementing this kind of inaccurateness from the physical world in software.

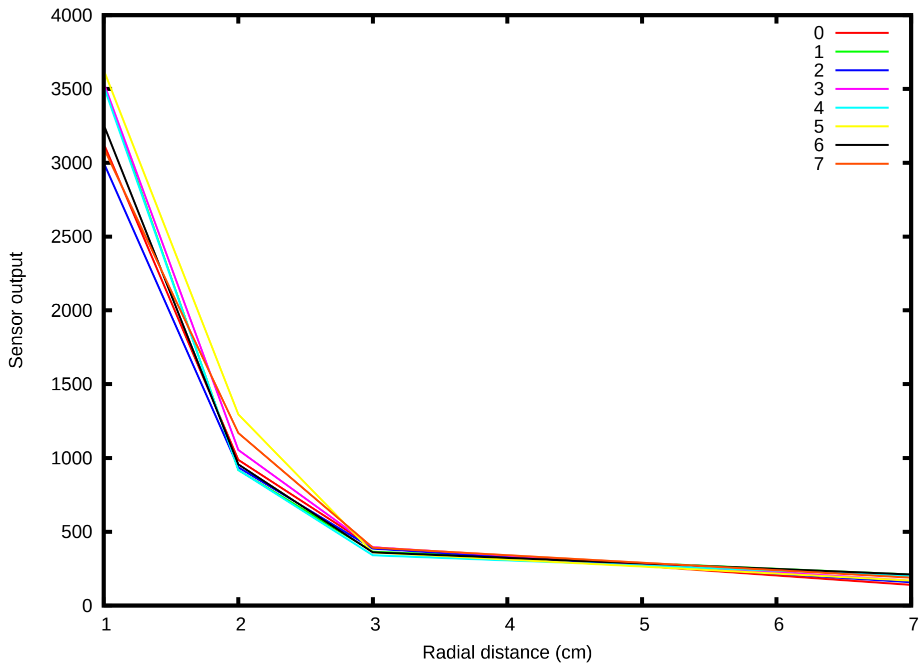


Figure 4.4: Diagram showing the values of the eight proximity sensors on the simulated e-puck, as a function of radial distance from the e-puck body, when detecting a light gray object. Measurements are done at distances 1, 2, 3, and 7 cm from the e-puck body.

Figure 4.4 does however show some small variations in sensor response, but this is most likely due to inaccuracies related to the measuring method used, aligning the e-puck so that the obstacle is directly in front of the sensor is more difficult in the simulated than in the physical world.

The overall level of the sensor response and level of noise appears to be of the same order of magnitude in both simulated and physical e-pucks.

4.2 Specific issues regarding Webots and the physical e-puck

In the previous version of this system, computer to e-puck communication was being handled by the *pucker* framework[3], using Python's serial communication module. Commands were simply sent and received from the e-puck with as little delay as possible, and for the most part the delay was negligible (given that the battery voltage is not too low, or that other factors such as distance degrades the bluetooth connection).

In this new version, Webots is used for both simulation and running physical e-pucks, and the communication process has become more complicated. Apparently, user commands (or data) is only transferred to and from the e-puck 1-2 times per second, sometimes even less frequently[4]. For some reason, the camera images are updated almost in real-time, so the problem is not the bluetooth connection.

This result in a rather “clumsy-looking” e-puck, bumping into objects before moving away from them, and sometimes rotating in a larger angle than intended because the stop-signal is received too late.

4.3 Obstacle avoidance

One of the most vital behaviors in the behavior control system (see section 3.5) is obstacle avoidance. Without this behavior running, the e-puck will take no notice of obstacles.

As described in section 3.7.1, the obstacle avoidance behavior primarily consists of a feedforward neural network gathering data from the e-puck's IR sensors.

Section 4.1 describes some problems regarding noise from the IR sensors, in addition to the sensors' different response to the color of the obstacles. It has therefore been necessary to threshold the sensor values, in order to minimize the risk of errors.

With the default speed values (half that of the maximum speed) the robot is able to move approximately 2 cm in 0.3 seconds, which is the average time in between each execution of the obstacle avoidance behavior. Looking at figure 4.1, the threshold should be at least 400 in order to detect obstacles within a 2 cm range. In practice, a threshold of 300 has proven to be appropriate.

Based on observations from running the system with obstacle avoidance, no special

[3]Created in 2009 by Keith Downing, Kjeld Massali and Dag Henrik Fjær.

[4]Tested by constantly probing the proximity sensors

considerations have been needed to compensate for the variations in sensor response with the physical e-puck seen in figure 4.1.

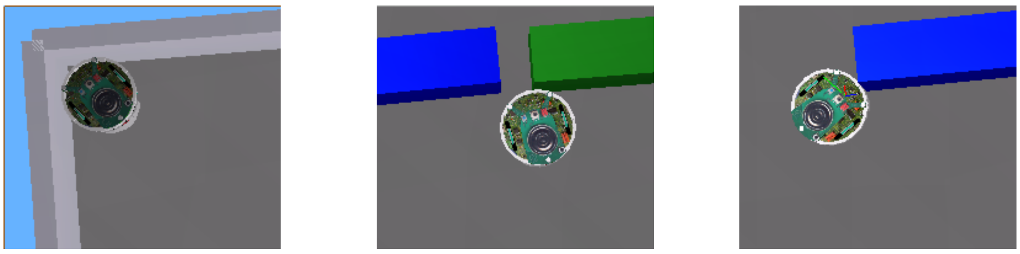

(a) Corner (b) Narrow gap between obstacles (c) Corner

Figure 4.5: Two different situations in which the e-puck proximity sensors are often tricked.

Some situations in which the proximity sensors are often "tricked" can be seen in figure 4.5. When the e-puck is located in a corner, such as in figure 4.5(a), the IR sensor readings will generally be slightly increased, compared to a similar situation with a rectangular obstacle. The most likely explanation is IR reflection, IR waves from front obstacle are reflected onto the left obstacle, before reaching the e-puck, resulting in an increased response on all affected sensors. This effect is the same in both the physical and simulated world.

Occasionally, all sensor values except those from the two back sensors are above the threshold, resulting in the e-puck turning 180 degrees when attempting to escape the corner.

In the physical world, the increase depends on the material and color of the obstacles, but in most situations this is not significant with regard to the e-puck's behavior in the vicinity of obstacles. In the simulated world IR response appears to be independent of material and color.

This "fear of corners" may cause some problems when traversing maze-like environments with turning corridors. The e-puck has a tendency to, instead of following the hallway, turn 180 degrees and return in the direction in which it originally came.

Another problem arises in situations such as figure 4.5(c). When the e-puck moves directly towards the outer corner of an obstacle, the proximity sensors often are not able to detect an obstacle until after it has crashed into it. This especially a problem in the physical world with gray colored obstacles, something which can be explained by figure 4.2. In the simulated world, the obstacle usually produces sensor values directly above the threshold value (around 400 - 500) when the distance between e-puck and the obstacle is at its minimum.

In figure 4.5(b) there is a narrow gap between the two obstacles, and this might lead to similar situations as above, no obstacles are detected on the front sensors until it is too late to avoid bumping into them.

When training the ANN for obstacle avoidance, nine training patterns have been used (see table E.2). One pattern for each sensor, and one for situations where all values are below the threshold of 300. Training the network with the backpropagation algorithm takes some time, so even though it is possible to re-train the network for each run of the system, the recommended method is to load a pre-trained network.

4.4 Wandering

With only the obstacle avoidance behavior running on the e-puck, the robot will move in a straight line until coming across obstacles.

In an environment with many obstacles, the addition of random wandering will not have much effect on the robot. But in situations where the robot may go for longer periods without any proximity sensor input, random wandering will help the e-puck change its direction more often and explore new areas of the playing field.

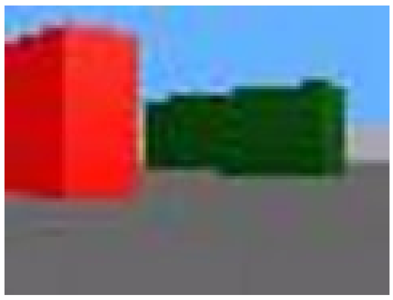
If the sleeping time of the wandering behavior is set too low, the e-puck will continuously zig-zag and swirl around, not being travel very far from the starting point over a reasonable amount of time. And if the sleeping time is too long, the behavior will essentially have no effect. The most effective value of the sleeping time depends on the speed of the robot, and the size of the playing area,

When running the e-puck on a table similar to that in figure 3.5 and 3.6, a sleeping time of 5 seconds is appropriate, allowing the e-puck to travel 20 to 30 cm before changing direction.

4.5 Searching for light and homing

Up till now, the e-puck has only moved around in a random pattern, dodging any obstacles on its way. Homing and light searching behaviors add a goal-orientedness to the system, having it search for locations on the table by using the integrated camera.

One problem with these two behaviors is that the total running time is approximately 1 second (somewhat longer on the physical e-puck, how much longer depends on the bluetooth connection). During that one second, the e-puck will have moved a distance away, and other behaviors may have altered its orientation, causing problems when trying to keep track of the location of interesting areas on the playing field. The solution to this has been to completely stop the wheels of the e-puck and pause all other behaviors during execution.


(a) Webots camera

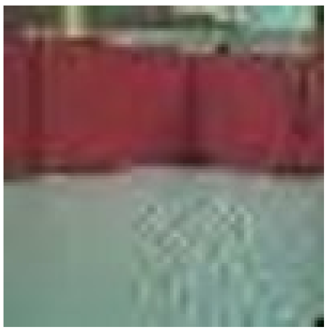

(b) Physical e-puck camera

Figure 4.6: Sample images from both the physical e-puck and the simulated e-puck in Webots. The red boxes are of approximately the same color in both worlds. The images have been scaled up in size.

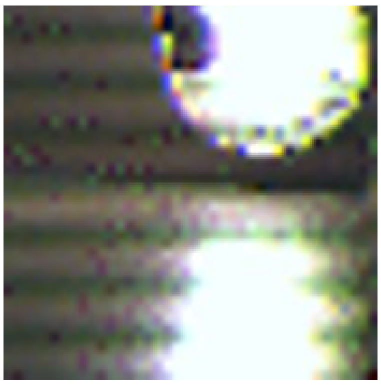
Camera images from the physical and simulated e-puck differ both in quality and color representation. Figure 4.6 shows sample images from the physical and simulated world, and as can be seen from the figure objects have more distinct boundaries and the colors are sharper in the simulated world. In general, the images from the simulated robot are easier to use in a vision system as they are less affected by noise and not so sensitive to shadows.

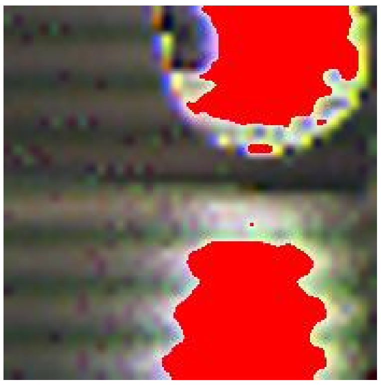
Search for light

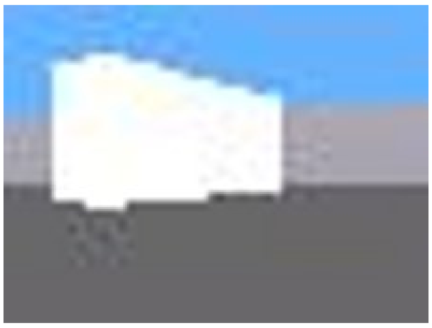
Figure 4.7 shows the result of thresholding a picture of the lamp using an intensity mask, labeling all pixels with an intensity greater than a given value[5]. In order to detect a sufficiently high concentration of white, the e-puck will have to be fairly close to the light source, approximately 10 cm, depending on the angle of impact and the ambient light conditions in the room.

In situations where other strong light sources are present, such as the sun shining through the windows in the physical world, the e-puck may detect this and indicate that it has found the light source (which is not directly wrong, as it has found *a* light source).

[5]Intensity is defined as $I = \sqrt{red^2 + green^2 + blue^2}$


(a) Original image, closeup of a desk lamp


(b) Thresholding using pixel intensity, $I \geq 400$.


(c) White box from the simulation world

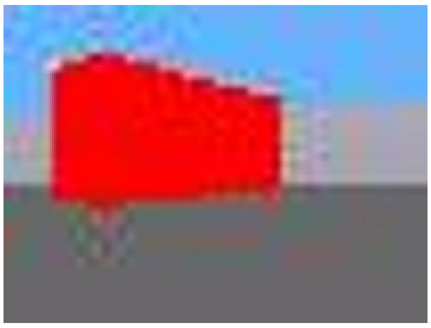

(d) Thresholding using pixel intensity, $I \geq 400$.

Figure 4.7: Diagram showing the result of intensity thresholding. All images have been scaled up from their original sizes.

In the simulation world, a white box is used to represent light[6]. As can be seen from figure 4.7 the intensity thresholding is much more straightforward in the image from the simulated world, only the pixels that are supposed to be white are white in the image, hence no risk of erroneous tagging.

Homing

There are several methods available for locating red objects in an image, two of them are demonstrated in figure 4.8. *Max* simply iterates over the image, marking

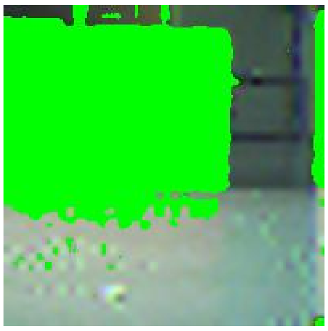
[6]To avoid messing up the lightning conditions on the table. Directing a sufficiently powerful lamp so that the e-puck camera can have eye contact with it affects the rest of the table much more in the simulated world than in the physical one.

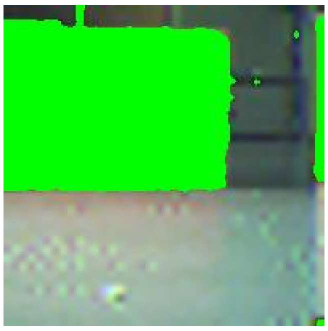
all pixels where the *red* component is the most dominant. For pure red objects, this method works very well, as can be seen in figure 4.8(b). It also works for some other shades of red, such as the red boxes covered by shadow in figure 4.8(d). However, in situations where non-red objects have a slightly red tint, the *max*-method fails miserably. An example of this is in figure 4.8(h), where almost the entire area of the gray floor has been marked as red. Adding constraints upon the minimal difference between the red and green/blue components will to a certain degree prevent situations like this, but then there is a risk of also excluding shades of red with a grayish tint.

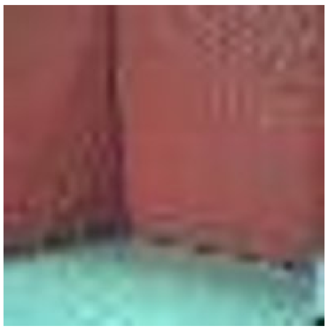
Therefore, it has been necessary to instead look at other methods. As can be seen from figure 4.8, a method using the ratio between red and green pixel components outperforms *max* in all test-images.

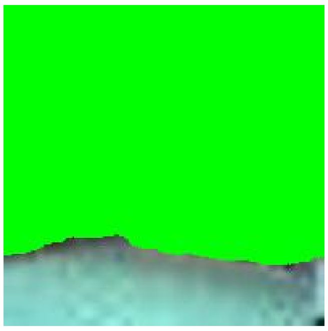
The r:g-ratio threshold determines the definition of “redness” in an image. A lower threshold would increase the number of false positives, while increasing it reduces the number of shades of red that will be labeled, as can be seen in figure 4.9. There is no such thing as a *correct* threshold value, it depends on both the situation and the camera used, but by experimentation a ratio of 1.3 has proved to be the most efficient in this project. As can be seen from figure 4.8, there are still some erroneously labeled pixels, especially in figure 4.8(i), but this is easily fixed by adding a lower bound in the pixel count method.

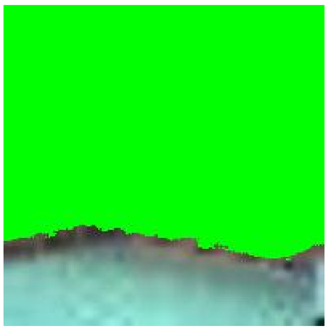
(a) Original image, red box.

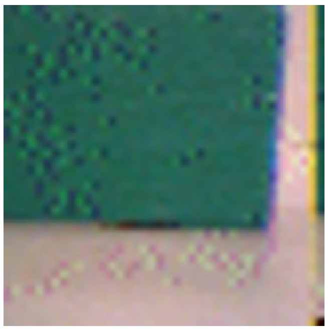

(b) Thresholding using *max*. Some gray pixels are erroneously labeled as red.

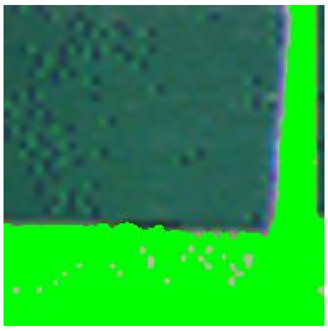

(c) Thresholding using the r:g-ratio of each pixel, $\frac{r}{g} \geq 1.3$.


(d) Original image, red boxes covered by shadow.


(e) Thresholding using *max*.


(f) Thresholding using the r:g-ratio of each pixel, $\frac{r}{g} \geq 1.3$.


(g) Original image, a green box (no pixels should be marked as red).


(h) Thresholding using *max*. Almost all gray pixels are marked red.

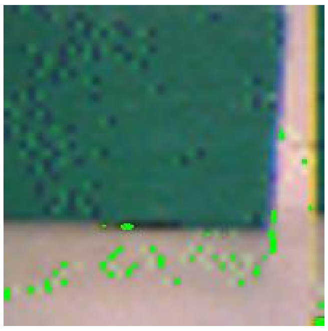
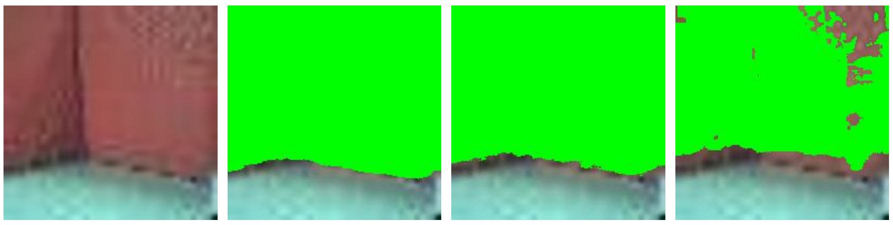

(i) Thresholding using the r:g-ratio of each pixel, $\frac{r}{g} \geq 1.3$. A few gray pixels are marked red.

Figure 4.8: Diagram showing the result of two different methods of measuring the amount of red in an image. *Max* marks the pixels where red is the dominant color (higher value than green and blue) while *r:g ratio* looks at the ratio between the red and green pixel components. All images have been scaled up from 40×40 pixels.



(a) Original image, red box covered by shadow (b) $\frac{r}{g} \geq 1.1$ (c) $\frac{r}{g} \geq 1.3$ (d) $\frac{r}{g} \geq 1.6$

Figure 4.9: The result thresholding with different values for the threshold ratio between red and green. Experimentation has shown that a value of 1.3 gives the best result. A smaller ratio tends to label too few pixels. All images have been scaled up from 40×40 pixels.

4.6 Signal localization

4.6.1 Using sound

The intention was to implement e-puck - e-puck communication by using sound (the SerCom-firmware has support for 5 predefined sounds). Unfortunately, the embedded microphones on the e-pucks have some difficulties localizing the sound emitted from the speaker of another e-puck. The microphone timing problems mentioned in section 4.1 manifest themselves. Also, the amplitude of the speakers is fairly low, reducing the radius in which localization is possible.

So instead an external speaker has been used for testing, emitting sine waves with a frequency of 700 Hz. In an area with few obstacles, the localization works fairly well when the e-puck is located no more than approximately 30 cm from the speaker.

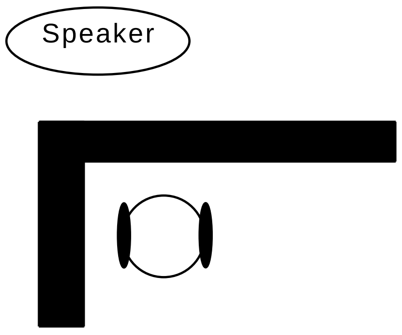


Figure 4.10: E-puck confused by a wall between itself and the speaker.

Since no success state is defined for this behavior, the e-puck will simply move towards the sound, then continue to wander around in the vicinity of the speaker.

In an area with many obstacles, the e-puck has more difficulties locating the speaker. First, obstacle avoidance has a higher priority than sound localization. In situations where the e-puck must continuously dodge obstacles, sound localization may never be allowed to control the e-puck. Also, it appears that the obstacles themselves tend to distort the sound (echoes or similar), in situations where the e-puck is located in areas with obstacles, output from the microphones sometimes indicate a wrong speaker location.

Situations like the one in figure 4.10 prove to be difficult for the e-puck. Sound localization will want e-puck to move straight forwards, while obstacle avoidance will move it either to the left or to the right. The result is that the e-puck will not be able to make any progress, obstacle avoidance will all the time move it away from the obstacle, while sound localization moves the e-puck towards it.

4.6.2 Localization using simulated radio signals

Localization in the simulated world suffers from none of the problems with distortion and echoes mentioned in the previous section. As mentioned in 3.7.3, a simulated radio signal emitter and receiver are used for localization, and the signal is not attenuated by neither obstacles nor the distance from the emitter[7].

As described in chapter 3.7.3, radio signals are sent as data packages. The packages are received automatically by the receiver, and stored in a FIFO queue until removed. If the rate of package sending is greater than the rate in which they are read and removed from the queue, packages will accumulate in the queue, and the longer the simulation is run, the more outdated the packages will be.

In itself this is not a great problem, in most cases it is not crucial to at all times have updated location data, a delay of a few seconds will still give accurate enough data. However, the localization information linked to a package uses the position (and orientation) of the e-puck *at the time the package was received.* So if the e-puck has moved at all in the interval between receiving and reading a package the localization data will be completely useless[8]. For instance a rotation of 90° will result in all localization data being offset by 90°).

A solution to this problem is to occasionally empty the receiver queue. It is also wise to make sure the sleeping time of the broadcasting layer is less than or equal to that of the receiving layer, to avoid unnecessary emptying of the queue.

A video can be found in the code repository that demonstrates what happens when the queue is not emptied[9]. In the video, three e-pucks attempt to follow an alpha e-puck broadcasting its position, but packages are accumulating in the receiver queue and the e-pucks are never able to locate the leader. A more successful demonstrating of signal localization will be discussed in section 4.8.1.

4.7 Threaded vs. non-threaded behavior control

As mentioned in section 3.5, the behavior controller can be run both with and without the use of threads.

The threaded implementation uses time (in seconds) as a measurement of how often a behavior is to be executed. To avoid the unnecessary complexity of implementing a timer in the behavior controller, an iteration counter is used instead.

[7]It is possible to limit the range of the signals, but this has not been done when performing the experiments described in this report.

[8]It took a fair amount of trial and error before this discovery was made

[9]Filename: *flocking-queue.mpeg*

In the current system, a scaling factor of 20 is used to translate time into iteration steps. The result is that the sequential program may execute each behavior more or less frequently over a given time period than the threaded implementation, depending on the performance of the computer used[10], and on how many behaviors that are active.

On a more (or less) powerful system, the scaling factor may need some adjustment.

4.8 Running the system

The following sections will describe a few runs of the systems with different settings and active behaviors. Videos illustrating the examples can be found in the code repository.

4.8.1 Alpha e-puck with flock

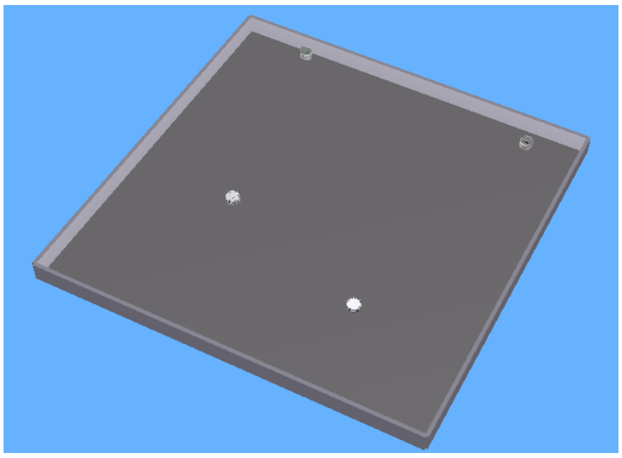
Figure 4.11 shows the result of a flock of three e-pucks following a leader. The simulation world is similar to that described in chapter 3.3, but all unnecessary objects have been removed. The e-puck leader is broadcasting its position approximately every second, while the rest of the e-pucks follow. The only other active behavior is obstacle avoidance.

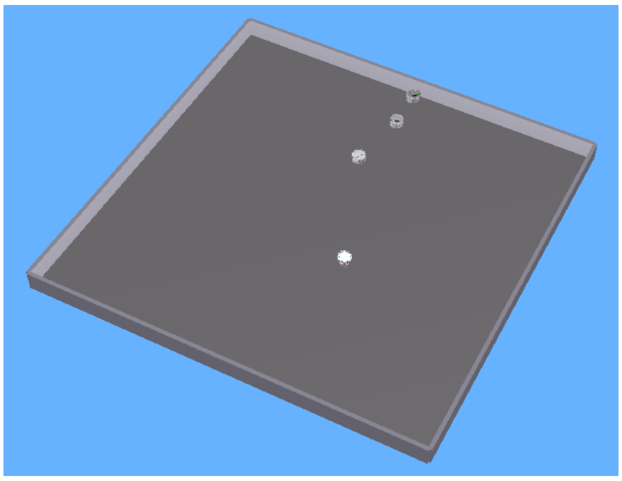
As can be seen from the figure and the attached video[11], the e-pucks start out moving in random directions, before the first signal is received and they start localizing the leader.

One thing to be noticed is that in areas where e-pucks cluster together, or in the vicinity of walls (or other objects), the e-pucks may wander off instead of following the leader for a short period of time. This is because signals are only received on average once every second, in the time between other behaviors (in this example obstacle avoidance) are in full control of the e-pucks. Signal updates once per second have proven to be a fair compromise between too much “free wandering” and constant small directional adjustments.

[10]A Dell Latitude D420 laptop with an Intel Core Duo 1.2Ghz CPU, running Ubuntu Linux have been used for the experiments in this report.

[11]Filename: *flocking.mpeg*


(a) Initial configuration


(b) Approximately 5 seconds after simulation start

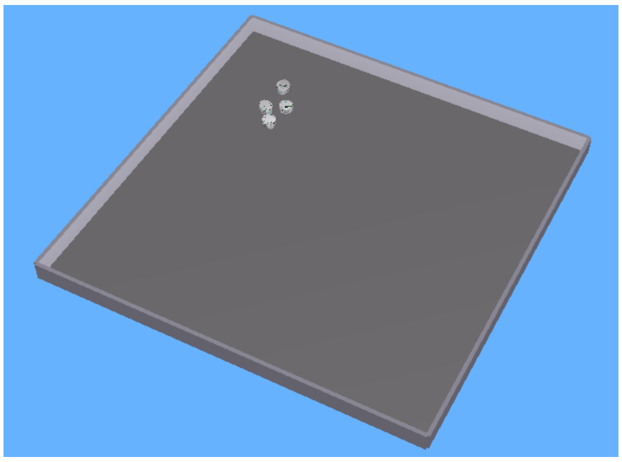

(c) Approximately 30 seconds after simulation start

Figure 4.11: Step by step illustration of 3 e-pucks following one alpha e-puck.

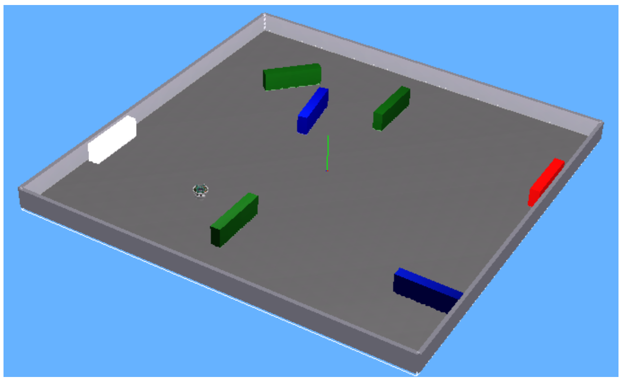
4.8.2 Locating boxes

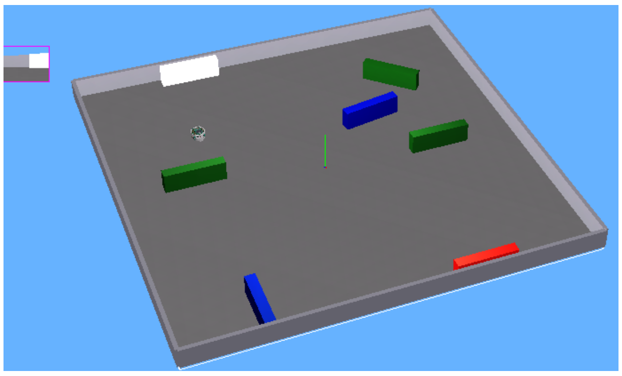
This example shows an e-puck with all behaviors active, with the exception of signal broadcasting and localization. Videos are available in the code repository[12].

Figure 4.12 describes step by step how the e-puck locates the white box. In 4.12(b) the e-puck have noticed a small area of white, and will start moving towards it, while in 4.12(c) the number of white pixels in the image is above the threshold, and a success state is reached.

In the attached videos, it is shown how the e-puck localizes first the white box and then the red one. As can be seen from the videos, every time one of the camera vision behaviors is run, the e-puck rotates around its own axis. Eight snapshots are made with 45° intervals (or fewer, if an interesting area is discovered). The "wandering off" described in section 4.8.1 is also visible here, especially noticeable when the e-puck is moving directly towards either the red or white box and then suddenly start to move in another direction. What happens is that even though behaviors such as random wandering have a very low priority, in some time steps higher priority behaviors are not run, thus the e-puck is allowed to diverge from its path for a few time steps.

[12]Filename: *boxes.mpeg* and *boxes-4epuck.mpeg*


(a) Initial configuration


(b) Moving towards white area

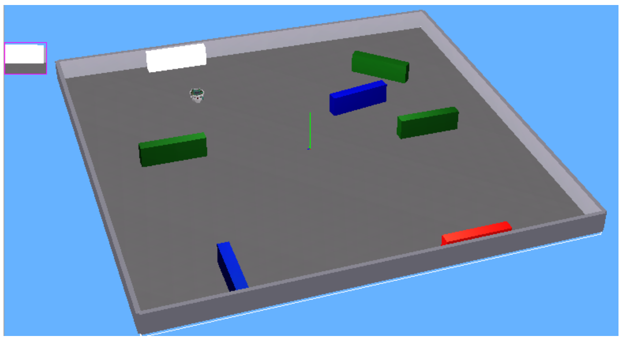

(c) Have found the white box

Figure 4.12: Step by step illustration of the e-puck moving towards the white box. Images from the e-puck camera are shown in the upper left corner.

4.8.3 Locating red box with help from flock

This example will look at how the e-pucks may help each other locate the boxes by using signal localization.

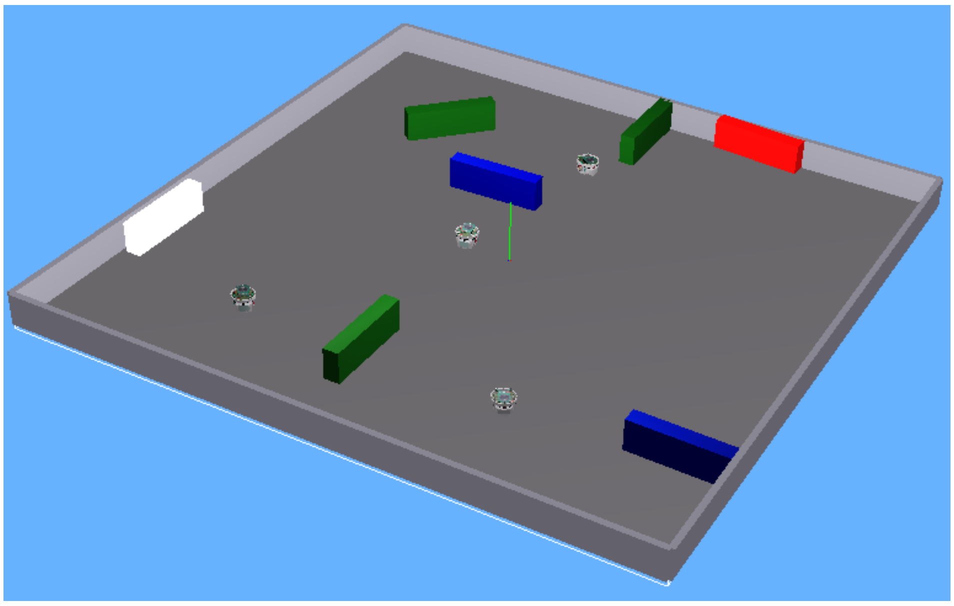


Figure 4.13: Initial configuration of e-pucks in the simulation world.

The arrangement of boxes in the previous example made the exploration relatively easy for the e-pucks. Not many obstacles were present between the white and red box, and the e-puck robots found both boxes relatively easy. Therefore the playing field has been rearranged in this example, and the e-pucks have been placed so that only one has direct visible contact with the red box at the start of the simulation as shown in figure 4.13. In order to more easily demonstrate what happens, all e-pucks will stop moving when they have located the red box.

Two videos are provided[13], one with broadcasting active and one without. In the case without broadcasting, three of the e-pucks are able to find the red box relatively easily, and only the e-puck in the lower left corner of the playing field (closest to the white box) does not manage to locate the box in the duration of the simulation.

In this case the e-puck only explored the lower left quadrant of the table. If it was to notice the red box it had to move over to the right side, past the green box. But this is above all a question of good or bad luck. In situations where the e-puck has no sensor input on which to base its decisions, it has to rely on the randomness of the

[13]Filenames: *cooperate.mpeg* and *no-cooperate.mpeg*

wandering-behavior. In some runs randomness takes care of the problem relatively fast, while in others the e-puck spends a long time wandering before locating the box.

In the example with broadcasting, the e-pucks start broadcasting after they have found the box. The start of the simulation is very similar to the one described above, three of the robots find the box relatively fast, while the last one are still in the lower left quadrant. But shortly after the first of the e-pucks have begun broadcasting, it receives the signal and is able to direct itself toward the box.

One small issue with the two above examples is that the last e-puck to find the box has some problems with the camera sometimes not being able to see the red box because of the e-pucks clustered in front of it.

5 Discussion

The previous chapter has shown the results of exposing the e-puck to various situations. This chapter will summarize these results, starting with an evaluation of the system as a whole.

The following evaluation criteria have been composed from Arkin (1998) (page 128) and Murphy (2000). The complete list with comments are presented in appendix A.

1. **Parallel implementation**: One of the main principles in reactive and behavior-based robotics is parallel execution of behaviors. The system implemented in this report is in theory completely parallel. All behaviors, together with the behavior controller are designed to be run in separate threads. What prevents parallelism in practice is limitations in the Python interpreter. As mentioned in section 3.7, Python prevents real parallelism. Also, the hardware that the system is to run on must be able to execute the required number of threads concurrently. As it is now, the system is *sufficiently parallel*, behaviors are executed in a close-to parallel fashion, and more than one[1] e-puck can run simultaneously.
2. **Hardware targetability**: The system is designed to run on any kind of hardware, provided it has an UNIX-type operating system. In theory, it ought to also run on Windows systems, but this has not been tested. Due to the hardware limitations of the e-puck robots, it has been necessary to run the system on a separate computer, but if the robots had been powerful enough, there are no reasons to believe the system should not be able to run on the robot.
3. **Niche targetability**: The system supports the creation of customized behaviors for performing given tasks.
4. **Portability**: Since no attachment is made to specific types of hardware, the system should be able to run on most types of "off the shelf" hardware. New functionality is easily added by creating new behaviors.
5. **Robustness**: The system is to a certain degree fault-tolerant. If some of the eight IR sensors were to stop functioning, the robot would still be able to do

[1]Up to 4-5 e-pucks simultaneously on the Dell Latitude D420 laptop used during development.

obstacle avoidance. In most situations, an obstacle is detected on more than one sensor, thus the effect of loosing one sensor would not be catastrophic. One major problem with the physical e-puck is that most sensors (and actuators) are affected by a low battery level, causing data to get lost in the transmission between e-puck and computer, and an increase in sensor noise levels. No measures have been taken to deal with the increase in noise, but in the case of lost data the system will keep asking until input is received. If, for some reason sensor data is lost temporarily the e-puck will get “back on track” once new data is received.

6. **Modularity**: The system is highly modular. Behavioral reuse is possible, and the system can with small modifications be run on other types of robots.
7. **Timeliness in development**: The behavior control system is object oriented, and sub-classing will make sure new behaviors inherits the basic properties needed. Also, Python code is generally fairly easy to understand (at least compared to lower level languages such as C).
8. **Run-time flexibility**: Currently, the system is not very flexible once execution has started. At run-time, it is possible to adjust the behaviors, for instance by a startup script.
9. **Performance**: The behavior controller and most of the behaviors are able to operate in (close to) real time. Running more than 4-5 e-pucks simultaneously may however cause some lag in the system.

The next sections will look at particular aspects of the system.

Action selection

The previous version of the system used priority based action selection, with static and manually set priorities. Each behavior would output a motor command that best suited the goals of that behavior, and the behavior controller chose the motor command with the highest priority.

In situations where there is little internal competition between behaviors this approach works OK, but problems arise in situations such as the one in figure 5.1, where the behaviors need to cooperate in order to reach an optimal decision. In this case, the IR sensors would show an obstacle directly in front of the e-puck, while the camera shows a large concentration of red behind the blue wall. The old system would have moved either to the left or right (direction 6 or 2 from figure 3.7), ignoring the red area, since obstacle avoidance has a higher priority.

The solution presented in this report is also priority based, but takes into consideration the output of more than one behavior. As described in section 3.5, this is done

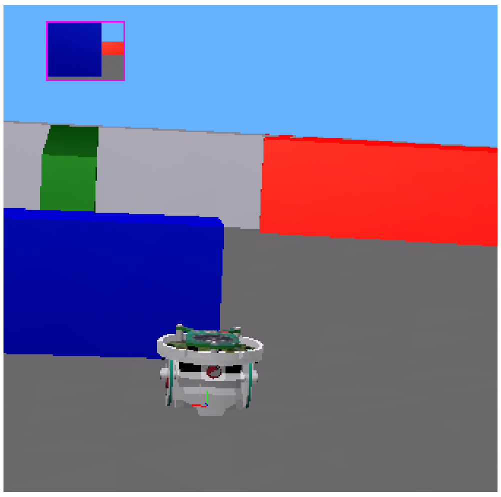


Figure 5.1: The e-puck is in a situation where both the obstacle avoidance priority and homing must cooperate in order to reach an optimal decision. The camera image from the e-puck can be seen in the upper left corner.

by having each behavior output a weighted list of possible moves. The priority of each behavior is used to scale the list; In a situation where behaviors disagree the highest priority behavior will give "its" directions a higher weight, thus increasing the chances of those directions to be chosen.

In the situation shown in figure 5.1, the output of obstacle avoidance would be like this: $\begin{bmatrix}-10 & 0 & 0 & 0 & 0 & 0 & 0 & -10\end{bmatrix}$ signaling that all directions except 0 and 7 is OK. The output of homing would be $\begin{bmatrix}1 & 1 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}$. Combining these two result in the behavior controller would result in direction 1 to be chosen, leading the e-puck directly towards the red box.

Memory

The system has little concept of memory. A type of short term memory[2] is implemented by storing the chosen directions for the past 10 time steps. A similar feature exists on a behavioral level, making it possible for a behavior to store its suggested moves and use them in the decision-making process in later time steps.

Currently, the short term memory only exists as a feature, it is not used for any practical purposes. One possible utilization of memory is to measure progress. Sequences of directions such as

```
0, 6, 6, 6, 2
```

[2]Implemented as a circular buffer of arbitrary length.

can be reduced to

```
0 4
```

In other words, for the past five time steps the robot has moved one step forwards and changed its orientation 180°, the rest of the time has been spent moving around in a circle.

In areas with a high density of obstacles or walls, such as in a labyrinth, the e-puck has a tendency to confuse itself and act like described above. In this situation it could have been useful to not only use the short term memory to determine if the e-puck was stuck in a dead end, but also to enable the e-puck to backtrack and escape.

Another use for short term memory is to create a type of volatile world model, representing the world as it has been perceived for the past 10 (or any other number of) time steps.

Such a model will never be 100% accurate. In the physical world, the wheels of the e-puck will spin when crashing into obstacles (and obstacles may also be slightly displaced from their original location), and the recorded wheel movements will no longer represent actual movements done by the e-puck. Another problem when acting in the physical world is the timing issues mentioned in chapter 4.2, the recorded movement may be a 45° turn, while the actual rotation may be 60° or more.

The same problems also exist in the simulation world, but to a lesser extent.

Acting in a group

The system can be run on several e-pucks simultaneously, and they can communicate by either using radio signals (in the simulated environment) or sound. This makes it possible to experiment with various cooperative or competitive behaviors, such as flocking or helping each other locate colored boxes.

A problem in the current system is that the e-pucks are not able to separate between obstacles and other e-pucks, from the viewpoint of the IR sensors, an obstacle is an obstacle. As seen in section 4.8.3 this caused some difficulties when all e-pucks were clustered in front of the red box. Similar problems arise in a flock with a high density of e-pucks. Even though there are no imminent risk of collisions, obstacle avoidance will be activated and attempt to separate the e-pucks.

If the system was able to discover other e-pucks, for instance by combining radio signals and camera vision, this could significantly improve the “social skills” of the e-pucks.

6 Conclusion

In this report, I have looked at different ways to autonomously control a robot. Pure deliberative and reactive systems have been discussed, in addition to hybrid and behavior-based approaches. Some artificial intelligence techniques useful in the design of a behavior-based controller were also examined.

Inspired by the ideas of behavior-based robotics, and in particular Brooks' subsumption architecture, I have implemented a basic system for behavior-based control of an e-puck robot. The system has been designed so that behaviors can be easily added and removed. It is also fairly easy to add new behavior controllers, without any need for modifications in other parts of the system.

Currently, the system has behaviors for obstacle avoidance, light detection, homing, signal and sound localization and random wandering. The system has a behavior controller, responsible for coordinating the output of all behaviors into robot movements. Action selection is inspired by ideas from DAMN (see 2.1.5), the output of each behavior is a weighted list of possible moves according to the goals of that behavior, and the move which get the most votes (highest sum) will be selected in each time step. This approach ensures that the goals of as many behaviors as possible will be taken into consideration.

One lesson learned during the year I have worked with the system is that seemingly simple ideas prove to be troublesome to implement, and often does not work quite as intended. During the first six months of the project, I worked solely with the physical e-pucks (without Webots), and it took some time before I got to know the e-pucks and all their oddities. If they suddenly refused to move (or moved in the wrong direction) it was just as likely that the battery was not properly charged, than that my implementation had any errors (not to say that my code was never to be blamed...).

Webots and the simulated robots are not quite as eccentric as their physical counterparts, but they too have their quirks. The problem with the receiver queue accumulating packages discussed in chapter 4.6.2 was not a problem until testing broadcasting combined with other behaviors, thus it was more difficult to discover the cause. And, not unexpectedly, delving into the experimental features of Webots may cause strange behavior in the system.

Additionally, Webots have a multitude of features and many of them are still undocumented and can be a bit tricky to discover. In that respect the Webots user group at Yahoo is very useful, the people there have valuable knowledge of the more secretive sides of Webots.

That being said, there have also been times when things just worked, magically. In the pre-Webots version of the system, one problem was that camera images took 1-2 seconds to transfer from the e-puck, slowing down the system a considerable degree. In the current version that let Webots take care of low-level communication details, the transmission delay is reduced down to a fraction of a second, causing much less problems.

6.1 Future work

During my work in the past year, much time has been spent developing the various behaviors and experimenting with different types of behavior controllers. The current behavior controller have been much improved compared to the previous version, and is able to combine output from multiple behaviors when making decisions.

One thing that would have been interesting is to incorporate ideas of sequences of actions into the controller, such as *navigate around the blue box then move towards the red one*, as opposed to the current system which is only able to plan one time step at a time. This could have opened up new opportunities for e-puck cooperation, for instance when navigating in a maze-like environment with the aid of other e-pucks. Knowing that the goal is somewhere to the left, it could have moved forwards until a left turn was possible, instead of like in the current system where it would attempt to move left right away and become confused when obstacle avoidance interferes and prevents it from bumping into the wall.

Another subject to further explore is the various social interactions possible in a group of more than one e-puck. So far I have looked at various scenarios for the e-pucks helping each other and “follow the leader”-type tasks, but none where the e-pucks actually cooperated to carry out a task.

Bibliography

Amit, D. J. (1992), *Modeling Brain Function : The World of Attractor Neural Networks*, Cambridge University Press.

Andre, D. & Teller, A. (1999), Evolving team darwin united, *in* 'RoboCup-98: Robot Soccer World Cup II', Springer-Verlag, London, UK, pp. 346–351.

Arkin, R. C. (1989), 'Motor schema – based mobile robot navigation', *The International Journal of Robotics Research* **8**(4), 92–112.
URL: *http://dx.doi.org/10.1177/027836498900800406*

Arkin, R. C. (1998), *Behavioral Based Robotics*, MIT Press, Cambridge, MA, USA.

Arkin, R. C. & Balch, T. (1997), 'Aura: Principles and practice in review', *Journal of Experimental and Theoretical Artificial Intelligence* **9**, 175–189.

Arkin, R. C. & Balch, T. (1998), Cooperative multiagent robotic systems, *in* 'Artificial Intelligence and Mobile Robots', MIT/AAAI Press.

Brooks, R. A. (1985), 'A robust layered control system for a mobile robot'.

Brooks, R. A. (1990), 'Elephants don't play chess', *Robotics and Autonomous Systems* **6**(1&2), 3–15.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.26.6594*

Brooks, R., Brazeal, C., Marjanovic, M., Scassellati, B. & Williamson, M. (1999), The cog project: Building a humanoid robot, *in* 'Lecture Notes in Computer Science', Springer-Verlag, pp. 52–87.

Cybenko, G. (1989), 'Approximation by superpositions of a sigmoidal function', *Math. Control Signals Systems* **2**(4), 303–314.

Dudek, G., Jenkin, M. R. M., Milios, E. & Wilkes, D. (1996), 'A taxonomy for multi-agent robotics', *AUTONOMOUS ROBOTS* **3**, 375–397.

Goodrich, M. (2000), 'Potential fields tutorial'.
URL: *borg.cc.gatech.edu/ipr/files/goodrich_potential_fields.pdf*

Heylighen, F. (1999), The science of self-organization and adaptivity, *in* 'in: Knowledge Management, Organizational Intelligence and Learning, and Complexity, in: The Encyclopedia of Life Support Systems, EOLSS', Publishers Co. Ltd, pp. 253–280.

Hopfield, J. J. (1982), 'Neural networks and physical systems with emergent collective computational abilities.', *Proceedings of the National Academy of Sciences of the United States of America* **79**(8), 2554–2558.
URL: *http://www.pnas.org/content/79/8/2554.abstract*

Jenkins, O. C. & Mataric, M. J. (2002), Deriving action and behavior primitives from human motion, *in* 'In International Conference on Intelligent Robots and Systems', pp. 2551–2556.

Kaelbling, L. P., Littman, M. L. & Moore, A. W. (1996), 'Reinforcement learning: A survey', *Journal of Artificial Intelligence Research* **4**, 237–285.

Khatib, O. (1985), Real-time obstacle avoidance for manipulators and mobile robots, *in* 'Robotics and Automation. Proceedings. 1985 IEEE International Conference on', Vol. 2, pp. 500–505.

Laue, T. & Röfer, T. (2004), A behavior architecture for autonomous mobile robots based on potential fields, *in* 'In 8th International. Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences), Lecture Notes in Artificial Intelligence, Lecture Notes in Computer Science', Springer, pp. 122–133.

Maes, P. (1989), How to do the right thing, Technical report, Cambridge, MA, USA.

Maes, P. & Brooks, R. (1990), Learning to coordinate behaviors, *in* 'AAAI Proceedings', pp. 796–802.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.4894*

Maris, M. & te Boekhorst, R. (1996), Exploiting physical constraints: Heap formation through behavioral error in a group of robots, *in* 'In Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems IROS-96', pp. 1655–1660.

Mataric, M. & Brooks, R. (1990), Learning a distributed map representation based on navigation behaviors, *in* 'In Proceedings of USA-Japan Symposium on Flexible Automation, Kyoto,Japan', pp. 499–506.

Mataric, M. J. & Michaud, F. (2008), Behavior-based systems, *in* 'Springer Handbook of Robotics', pp. 891–909.

Minsky, M. (1986), *The society of mind*, Simon & Schuster, Inc., New York, NY, USA.

Moravec, H. (1980), Obstacle avoidance and navigation in the real world by a seeing robot rover, *in* 'tech. report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University doctoral dissertation, Stanford University', number CMU-RI-TR-80-03.

Murphy, R. R. (2000), *Introduction to AI Robotics*, MIT Press, Cambridge, MA, USA.

Nehmzow, U. (1999), 'Vision processing for robot learning', *Industrial Robot* **26**, 121–130.

Rosenblatt, F. (1958), 'The perceptron: A probabilistic model for information storage and organization in the brain', *Psych. Rev.* **65**, 386–407. (Reprinted in *Neurocomputing* (MIT Press, 1988).).

Rosenblatt, J. (1995), DAMN: A distributed architecture for mobile navigation - Thesis summary, *in* 'Journal of Experimental and Theoretical Artificial Intelligence', AAAI Press, pp. 339–360.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986), 'Learning internal representations by error propagation', pp. 318–362.

A Evaluation criteria for a behavior-based system

Evaluation criteria are composed from (Arkin 1998, page 128) and Murphy (2000):

1. **Support for parallelism**: Behavior-based systems are inherently parallel, how does the architecture provide for this?
2. **Hardware targetability**: How well the architecture runs on a physical robot.
3. **Niche targetability**: How well the system performs for the intended application.
4. **Portability**: Could the system be used for other applications or run on other hardware?
5. **Robustness**: How does the system handle failing components?
6. **Timeliness in development**: Does the system provide tools and development environments for implementation, or is it more a philosophical constuct?
7. **Run-time flexibility**: How well can the system be adjusted or reconfigured during execution?
8. **Performance effectiveness**

B Usage

B.1 System requirements

The programs have been tested on Ubuntu Linux, but ought to also run on most other flavors of Linux. It will probably work OK on Mac OS X, and might even work on Windows (the threaded version of `BehaviorController` will probably not work as intended on Windows though).

The programs have only been tested using Python 2.6, but will probably also work with 2.5 and 2.7. The software will not work with Python 3.0 without modifications to the code.

The following Python packages should be installed for the program to function correctly:

- Numerical Python (numpy)
- Python Imaging Library (PIL)
- PySerial

Other non-essential packages, such as `picle` may also be needed for the system to run without any complaining. For instance, the `picle`-module is only used for saving and loading a Python object to a text-file, and is not needed if these features are not used. Commenting out the appropriate `import`-statements should solve this problem.

In addition to Python, the Webots simulator must be downloaded and installed. A free 30 days trial version can be downloaded from `http://www.cyberbotics.com/`.

B.2 Introduction to using Webots with e-pucks

1. First, create a new project. Four new folders will be created in your project folder where project-specific files will be stored: `controllers`, `plugins`, `protos` and `worlds`.
2. To modify the simulation world you can either use the graphical interface

in Webots, or you can modify the world-file directly. Creating a world from scratch is rather tedious, so I recommend starting with a world file from one of the examples bundled with Webots, or take a look at the `Webots-e-puck.wbt` file in the code repository.

3. Add an e-puck robot controller (`Wizard` → `New robot controller`). The controller-file will be stored in the `controllers` folder. Take a look at the `e-puck_Webots.py` file in the code repository for some examples on how to program a controller in Python.

4. If you want to add features to your e-puck, you have to modify the `e-puck.proto` file. Copy from `[Webots install folder]/projects/default/protos/robots/e-puck.p` and save it in your project `proto` folder. It is important that you do not rename the file, it *must* be called `e-puck.proto`. The proto-file can also be useful if you need to know the name of an object, for instance the proximity sensors. (when enabling features in your controller, you have to use the hard-coded name from the proto-file).

For a more in-depth introduction, take a look at the Webots reference manual[1]. The Webots-users group at Yahoo[2] is a great place to get help should you run into difficulties.

Unfortunately there is not an overwhelming amount of information around related to creating robot controllers in Python, but the Python API in the Webots reference manual has some code examples. With a trial-and-error approach it is also fairly easy to translate C++ code samples from the documentation into Python.

The following code creates an e-puck object using files from the code repository, and prints out the proximity sensor values while moving forwards:

```
from e-puck_Webots import *

e = e-puck_Webots()
e.set_wheel_speeds(1,1)

while True:
    print e.get_proximities
    e.step(self.timestep)
```

[1]Available at `http://www.cyberbotics.com/cdrom/common/doc/Webots/reference/reference.html`.

[2]`http://tech.groups.yahoo.com/group/Webots-users/`

B.3 Physical e-puck communication

This section describes how to connect to the e-puck robot. The following procedures are written for Ubuntu Linux, but ought to work on most systems (with some modifications).

How to pair with an e-puck

1. Make sure your system has a functioning bluetooth controller, and that the e-puck has been flashed with a firmware supporting bluetooth communication (for instance SerCom[3]. You also have to install the `bluez-utils` package.
2. Find the mac address of your e-puck, using the command `hcitool scan`.
3. Using the mac address from the previous step, bind it to a system device with `sudo rfcomm bind rfcomm0 [mac]`. The device can afterwards be found in `/dev/rfcomm0`. (it is not necessary to name the device `rfcomm0`, any name could be used).
4. To check if the e-puck is connected, and that the communication channel is open (clean), use the command `rfcomm -a`. If output indicates that the channel is closed, or you get no output at all, the previous steps must be repeated. `rfcomm release rfcomm0` might also be useful.

Of course, a graphical bluetooth connection manager can also be used, just make sure you have control over which device (Unix) or COM-port (Windows) the e-puck is bound to (you will need this later, when connecting to the e-puck in Python).

Communicating with the physical e-puck in Webots

1. Make sure the e-puck is connected to the computer, and that your Webots-project works as expected with simulated e-pucks.
2. Start Webots, and load the e-puck project files. Double-click on the e-puck(s) you want to use, and chose the correct device path or COM-port in the drop-down list.
3. Close the e-puck window and run the project as you normally do with simulated e-pucks.

[3]SerCom is downloadable from `http://www.e-puck.org`

B.4 ANN Generator

First, a setup file describing the neural network must be created. The setup file contain all necessary information needed to create the network, such as the number of nodes in each layer, their activation functions, whether the layer is active during training and/or testing and so on. The links are specified in the same manner, detailing the link topology, prelayer, postlayer and initial weight settings.

Examples of the syntax used in the setup files can be found in appendix E.

This example creates a single layer (+ input) perceptron network from the setup file "settings.ann" (see E):

```
import ann

network=ann.create_ann("settings.ann")

trainingdata1=([0,0,0,0,0,0,1,1],[0,1])
testdata1=[0,0,0,0,0,0,0,1]

ann.delta_learning(network, ([trainingdata1])
ann.test([testdata1])

print ann
```

The network is trained to output `01` to `00000001` using delta learning. Other trainingcases can be added the same way (as long as they are all linearly separable).

Unsupervised learning is done by executing the built-in `train()` method, with the training data as argument. Also, an unsupervised learning function must be specified for each plastic link in the setup file.

B.5 The Behavior Control System

The controller can be run either with or without threads, specified by the `threaded` option in the configuration file, and with an arbitrary number of active behaviors.

The following example creates a behavior controller and two behaviors (obstacle avoidance and random wandering). It is assumed that the neural network governing obstacle avoidance has been created and trained beforehand:

```
from e-puck_Webots import *
```

```
e-puck_controller = e-puck_Webots()
initialize_behaviors("behaviors.conf", e-puck_controller)
```

The configuration file `behaviors.conf` looks as following:

```
##########
##Usage:
##All lines starting with '#' is ignored.
##All other lines should be on the form 'object variable1=value1
    variable2=value2 ...'
##########

#Setup of behavior controller:#

Global:threaded=False

layer Wander ann=None
layer AvoidObstacles ann="obs_avoidance.ann"
```

B.6 Main file

The file named `[controllername].py` located in the `controller/controllername` folder in the Webots project area is where Webots expect to find initialization data for the controller. The `controller/e-puck/e-puck.py` file in the code repository contains a working e-puck controller making use of the behavior control system.

C Code structure

Table C.1 gives an overview of the purpose and functionality of the different Python files in the code repository, downloadable from `http://folk.ntnu.no/majo/masteroppgave`.

All files are located in `controller/e-puck` folder (`e-puck` is the main e-puck controller). Additional controllers created are `interactive_e-puck`, enabling the user to control the e-puck by using the keyboard and `alpha_e-puck` which broadcast its location to all other e-pucks.

Additionally, some configuration files for neural networks and behavior controllers are provided.

Table C.1: Overview of the functionality of the files in the code repository.

Name	Description
`ann_activation_functions.py`	Various activation functions used to calculate activation levels for ANN-nodes.
`ann_supervised_learning.py`	Supervised learning functions operating on an ANN-object
`ann_unsupervised_learning.py`	Unsupervised learning functions, used by the `train()` method in the ANN class.
`ann.py`	Main class for generating an ANN.
`utilities.py`	Various generic utility functions.
`e-puck.py`	Main file, setting up and initializing all necessary objects.
`e-puck_Webots.py`	E-puck controller class file.
`image.py`	Various methods for modifying PIL images.
`behavior_layers.py`	Main class for the behavior controller and behavior classes.

D List of e-puck commands supported by SerCom

The following is a list of commands supported by the e-puch with the current version[1] of the SerCom software through a bluetooth/serial interface.

`"A"`	`Accelerometer`
`"B,#"`	`Body led 0=off 1=on 2=inverse`
`"C"`	`Selector position`
`"D,#,#"`	`Set motor speed left,right`
`"E"`	`Get motor speed left,right`
`"F,#"`	`Front led 0=off 1=on 2=inverse`
`"G"`	`IR receiver`
`"H"`	`Help`
`"I"`	`Get camera parameter`
`"J,#,#,#,#"`	`Set camera parameter mode,width,heigth,zoom(1,4 or 8)`
`"K"`	`Calibrate proximity sensors`
`"L,#,#"`	`Led number,0=off 1=on 2=inverse`
`"N"`	`Proximity`
`"O"`	`Light sensors`
`"P,#,#"`	`Set motor position left,right`
`"Q"`	`Get motor position left,right`
`"R"`	`Reset e-puck`
`"S"`	`Stop e-puck and turn off leds`
`"T,#"`	`Play sound 1-5 else stop sound`
`"U"`	`Get microphone amplitude`
`"V"`	`Version of SerCom`

[1]Version 2.0.0, from January 2008

E ANN setup files

8-node perceptron network trained for obstacle avoidance

The network has 8 input nodes, and 2 output nodes fully connected to each other. The input is assumed to be the eight IR sensors, while the output is motor commands to the right and left wheel of the e-puck.

Training data

The network has been trained using delta learning, with training data according to table E.1.

Table E.1: Obstacle avoidance training data. 300 is an appropriate threshold value between negligible and actual obstacles.

Active IR sensor	Input	Desired output
Front right	`[300,0,0,0,0,0,0,0]`	`[-1,0]`
Right	`[0,300,0,0,0,0,0,0]`	`[-1,0]`
Middle right	`[0,0,300,0,0,0,0,0]`	`[-1,0]`
Front left	`[0,0,0,0,0,0,0,300]`	`[0,-1]`
Left	`[0,0,0,0,0,0,300,0]`	`[0,-1]`
Middle left	`[0,0,0,0,0,300,0,0]`	`[0,-1]`

obs_avoidance_delta.ann

```
##########
##Usage:
##All lines starting with '#' is ignored.
##All other lines should be on the form 'variable=value'
##########
#Perceptron network#

Global:exec_order=[0,1]
Global:maxrounds=10
Global:learningrate=0.2
Global:traininggrounds=1

##Layers
#input
layer neurons=8 layername=0 activationFunction=Step threshold=100
    tactive=True tquiescent=False ractive=True rquiescent=False

#output
layer neurons=2 layername=1 activationFunction=VariableStep threshold
    =[-0.999,-1,1] tactive=True tquiescent=False ractive=True
    rquiescent=False

##Links
link prelayer=0 postlayer=1 topology='full' maxweight=0.0 minweight=0.0
```

3 layer feedforward network trained for obstacle avoidance

The network has 8 input nodes, 10 hidden nodes in the middle layer and 8 output nodes. The input nodes is assumed to represent the values of the IR sensors of the e-puck, while the output is a weighted list of directions as shown in figure 3.7.

Training data

The network has been trained using backpropagation learning, with training data according to table E.2.

Table E.2: Obstacle avoidance training data. 300 is an appropriate threshold value between negligible and actual obstacles.

Active IR sensor	Input	Desired output
Front right	`[300,0,0,0,0,0,0,0]`	`[-1,-1,0,0,0,0,0,-1]`
Right	`[0,300,0,0,0,0,0,0]`	`[-1,-1,-1,0,0,0,0,0]`
Middle right	`[0,0,300,0,0,0,0,0]`	`[0,-1,-1,-1,0,0,0,0]`
Front left	`[0,0,0,0,0,0,0,300]`	`[-1,0,0,0,0,0,-1,-1]`
Left	`[0,0,0,0,0,0,300,0]`	`[0,0,0,0,0,-1,-1,-1]`
Middle left	`[0,0,0,0,0,300,0,0]`	`[0,0,0,0,-1,-1,-1,0]`
Back left	`[0,0,0,300,0,0,0,0]`	`[0,0,-1,-1,0,0,0,0]`
Back right	`[0,0,0,0,300,0,0,0`	`[0,0,0,0,-1,-1,0,0]`
None	`[30,30,30,30,30,30,30,30]`	`[0,0,0,0,0,0,0,0]`

obs_avoidance_backprop.ann

```
##########
##Usage:
##All lines starting with '#' is ignored.
##All other lines should be on the form 'variable=value'
##########
#3 layer feed forward network, 8 input and 8 output nodes#

Global:exec_order=[0,1,2]
Global:maxrounds=3000
Global:learningrate=0.2
Global:traininggrounds=3000

##Layers
#input
layer neurons=8 layername=0 activationFunction=Step threshold=300
    tactive=True tquiescent=False ractive=True rquiescent=False

#hidden
layer neurons=10 layername=1 activationFunction=Tanh threshold=1
    tactive=True tquiescent=False ractive=True rquiescent=False

#output
layer neurons=8 layername=2 activationFunction=Tanh threshold=0 tactive
    =True tquiescent=False ractive=True rquiescent=False

##Links
link prelayer=0 postlayer=1 topology='full' maxweight=-0.5 minweight
    =0.5 plastic=True
link prelayer=1 postlayer=2 topology='full' maxweight=-0.5 minweight
    =0.5 plastic=True
```

Three layer network with maxnet-functionality for sound localization

Manual weights

The weights between layers 2 and 3 (output layer) have been set manually. Weight matrix:

```
0 1 0
1 0 0
```

goto_sound.ann

```
##########
##Usage:
##All lines starting with '#' is ignored.
##All other lines should be on the form 'variable=value'
##########
#Maxnet#

Global:exec_order=[0,1,2,3]
Global:maxrounds=100
Global:learningrate=0.2
Global:traininggrounds=1

##Layers
#input
layer neurons=3 layername=0 activationFunction=PositiveLinear threshold
    =100 tactive=True tquiescent=False ractive=True rquiescent=False

#maxnet
layer neurons=3 layername=1 activationFunction=PositiveLinear threshold
    =0 tactive=True tquiescent=True ractive=True rquiescent=True

#filter
layer neurons=3 layername=2 activationFunction=Step threshold=0.001
    tactive=True tquiescent=False ractive=True rquiescent=False

#output
layer neurons=2 layername=3 activationFunction=Linear threshold=0.0
    tactive=True tquiescent=False ractive=True rquiescent=False


##Links
link prelayer=0 postlayer=1 topology='1-1' maxweight=0.0001 minweight
    =0.0001

link prelayer=1 postlayer=1 topology='triangular' maxweight=-0.3
    minweight=-0.3

link prelayer=1 postlayer=1 topology='1-1' maxweight=1.0 minweight=1.0

link prelayer=1 postlayer=2 topology='1-1' maxweight=1.0 minweight=1.0

link prelayer=2 postlayer=3 topology='full' maxweight=0.0 minweight=0.0
     weights_from_file='maxnet-sound-weights.txt'
```