Evolving vision-based navigation on wheeled robots

Matthijs van Leeuwen, Jilles Vreeken & Arne Koopman

Adaptive Intelligence Laboratory, Intelligent Systems Group,
Institute for Information and Computing Sciences, Utrecht University
Correspondence e-mail address: mleeuwen@cs.uu.nl

We evolved spiking circuits for vision-based navigation on a wheeled robot in a randomly textured environment. We investigated two different fitness functions of which one is more explicit in rewarding target behaviour: navigating the arena. The first experiments were done with proximity sensors, the rest with vision. From comparison of the evolved individuals we conclude that more explicit rewarding may result in better behaviour. In earlier work by Floreano & Mattiussi [4], vision experiments were aided by pre-processing visual data, feeding contrast data into the spiking circuit. We show that using a filter simplifies the task, but that efficient and robust strategies can be evolved by using raw vision data as input, indicating that spiking circuits are even more powerful in real-world tasks than already assumed.

Evolutionary robotics is a relatively new field in the artificial intelligence research area. Even so, the results that have been achieved so far are all very promising [17,20,22]. The research done in this field follows a bottom-up approach of biologically inspired techniques, of which artificial evolution and neural networks are examples. Nolfi and Floreano [17] pioneered the field and their experiments, in which they evolved neural networks to control real robots, are still regarded as benchmark references. Most of their experiments were conducted on a Khepera robot [16]; a small, versatile and robust robot that we've also used in our experiments.

In previous work [10], we have shown that spiking neural networks are well able to solve temporal tasks by evolving networks for certain time series. Furthermore, results of evolutionary experiments with spiking networks [4,5,12,22] indicate that these circuits are well suited as real-time robot controllers. The networks in these experiments were only provided with vision-data as input to control the robot.

The experiments we present in this paper have been largely based on those done by Floreano and Mattiussi [4] with a vision-sensor equipped Khepera. However, we varied many of the experimental settings such as fitness function, vision pre-processing filter and network size. Besides, we explored the performance of different fitness functions with two sensor modalities: proximity and vision. The task the Khepera was asked to do was the same for all experiments: navigate around the environment as fast as possible without colliding with objects or walls, driving straight ahead as much as possible.

We will first introduce the important concepts artificial evolution and spiking neural networks, after which we will proceed with the implementation of these in our software and the set-up we used for our experiments. We will then present our results, analyse the evolved behaviours and conclude with a discussion.

Artificial evolution

Genetic algorithms [8] are widely used for optimisation and search problems, in particular when the parameter-space to explore is extremely large. When applied to general search problems, large sets of possible solutions are recursively evaluated and 'reproduced' to form new generations of possible solutions. This process is inspired on evolution as found in biology: natural selection favours individuals that are 'better' suited for survival as these have a higher chance to reproduce. Slowly but sturdy, in terms of generations, this process steers the population towards a better adaptation to the environment. And although crossover and mutation may introduce bad combinations or mutations occasionally, on the long run these processes guarantee genetic diversity and introduce new combinations and mutations that can make individuals perform better.

This concept is exactly the same for artificial evolution: at the start of an experiment we use a randomly initialised population of a certain number of individuals. Each individual is then evaluated on some task for which performance ('fitness') can be measured. Individuals that perform better have a higher chance to be selected for reproduction; the process in which two parents are combined and crossover and mutation is applied. Each new generation should then be a little bit better, but as crossover and mutation are random, we should only look at performance differences on the long run.

When we apply artificial evolution on robots, it is called evolutionary robotics. A very important aspect in this case is that we are dealing with real robots in real environments, which is completely different from simulations [17]. We now can be a little more specific about what individuals are: in evolutionary robotics, each individual has a genome that encodes a robot controller. A robot controller can be described as 'something that uses sensor data to control a robot'.

The genome that encodes the robot controller can be anything: one could use a single bit-string of fixed length, though it is also possible to model more complicated genomes consisting of i.e. multiple chromosomes of variable length and build from larger a alphabet like {A, C, G, T}. Such a genome is in fact the genotype of an individual. In order to evaluate the fitness of an individual we have to map this to its corresponding phenotype: the robot controller. As mentioned above, such a robot controller could be anything that accepts input and provides output; only controllers that show sane behaviour will acquire good scores though. Neural networks are often used as robot controllers, as they are very versatile and it's fairly easy to encode network parameters as topology and synaptic weights in the genome.

For the tasks investigated in evolutionary robotics it is often hard (if not impossible) to design a robot controller by hand, the correlations to be made are simply too complex: spatial-temporal information is often very important and robot controllers (like spiking neural networks) that can process these have many internal dynamics. Often no appropriate training methods exist for the controller architecture of choice, making artificial evolution a good alternative: evolution is able to explore a large search space and can combine partial solutions to form (more) complete solutions.

Most learning methods that are available for neural networks are supervised: back-propagation [18], spiking backprop [2] and RTRL [21] are only a few well-known examples (for different types of networks). In robotics, it is seldom possible to use supervised learning, as this requires a target output for each possible input. For most tasks that we want our robots to do though, we only have an informal description of the target behaviour. When supervised learning is not an option, unsupervised training remains a possibility. Hebbian learning [1] is such a method, but as it is primarily useful for clustering tasks, it is not specifically what we want either.

That we only have an informal description of the target behaviour is no problem for artificial evolution: the only thing we need is a fitness function, a function that measures the performance of an individual. Even though a good fitness function is not always easy to find, it's not impossible. When we want to evolve robot controllers that are able to drive a wheeled robot ahead, for example, we could sum all positive actual wheel speeds to determine the fitness of an individual.

It is probably unnecessary to mention that there are many parameters that can be chosen for artificial evolution. The parameters we used are mostly the same as in the experiments by Floreano and Mattiussi [4]. The population consists of 60 individuals and the first generation is randomly generated. Each individual has a single bitstring that encodes a spiking neural network (the encoding will be discussed later) that can be tested on the real robot; each individual is tested for 2 epochs of 30 seconds each.

When fitness values for all individuals in a population have been measured, reproduction can be done. There are several ways for doing this, we have used truncation selection: take only the best 15 individuals of each generation and dispose of the rest. These 15 individuals are reproduced, which means that they are coupled in parent pairs on which the genetic 'operators' are applied: crossover and mutation. We used basic single-point crossover, cutting the parent DNA strings at the same (randomly determined) point and switching the ends (crossover is applied on each parental pair with a chance of 10%). The mutation operator toggles every single bit of the genome with a certain chance (5%). For improved evolutionary stability, elitism was used in reproduction: by always retaining the best individual (without modification of the genome) we ensured that our search always kept the current best solution.

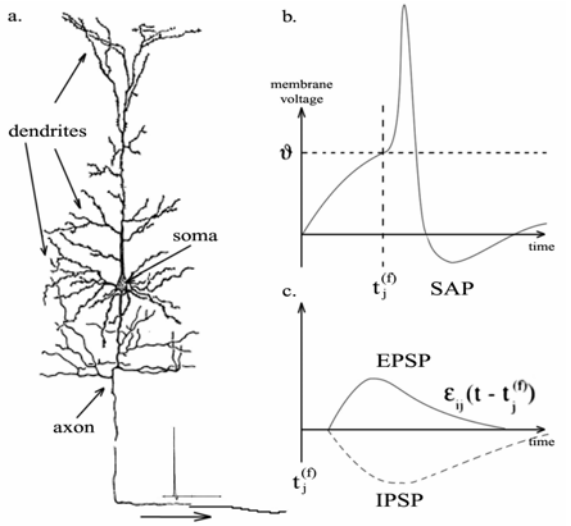

Figure 1. (a) Schematic drawing of a neuron. (b) Incoming postsynaptic potentials alter the membrane voltage so it crosses threshold value ϑ; the neuron spikes and goes into a refractory state. (c) Typical forms of excitatory and inhibitory postsynaptic potentials over time. [7]

Spiking neural networks

Classic models of artificial neural networks are inspired on biological neurons, but mimic their behaviour in a very simplified manner. Real neurons show very complex dynamics in their signalling behaviour. The resulting signals, or spikes, will eventually bring their message after having been processed by various complex mechanisms. For a long time it has been taken as a safe assumption that these individual signals carry no information, that just the average activity of a neuron is of importance to describe its activity.

As a result, traditional neuron models do not employ individual pulses, but have output signals that typically lie between 0 and 1. These can be seen as the normalized firing frequencies of the neuron. This type of signal is called rate coding, where a higher rate of firing correlates with a higher output signal. Pulse coding does not use such averaging mechanisms, but use individually timed spikes.

Recent discoveries in the field of neurology have shown that neurons in the cortex perform analogue computations at incredible speeds. Thorpe et al. [19] demonstrated that humans analyse and classify visual input (i.e. facial recognition) in under 100ms. As it takes at least 10 synaptic steps from the retina to the temporal lobe, this leaves about 10ms of processing-time per neuron. Such a time-window is much too small to obtain a reliable activity-average for rate coding (see also fig. 2) [19,13]. This does not mean that rate coding is never used, though for calculations where speed is an issue pulse coding schemes are favoured [14].

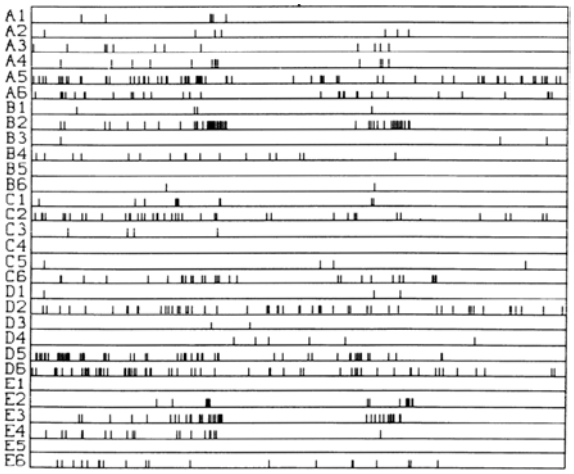

Figure 2. A 4 second recording of the neural activity recording from 30 neurons of the visual cortex of a monkey. Each vertical bar indicates a spike. The human brain can recognize a face within 150ms [19], which correlates to less than 3mm in this diagram; dramatic changes in firing frequency occur in this time span, neurons have to rely on information carried by solitary spikes. [11,20]

Real neurons send out individual signals: short and sudden increases in voltage. Due to their nature and form (see fig. 1a) these signals are commonly referred to as 'spikes' or 'pulses'. These electrochemical action potentials travel from the soma down its dendrites to the synapses. These chemicals cannot cross the synaptic gap, but they induce the release of neuro-transmitters that can cross and by doing so a post-synaptic action potential (see fig. 1c) is formed. While spikes are identical in form, the strength of post-synaptic potentials is influenced by many variables, like the amount of released neuro-transmitters and the synapse's capability to replenish these. Therefore a synapse is no simple signal-conversion device, but a highly complex signal pre-processor. This pre-processing and its many variables make development and learning possible in neural networks. The term 'synaptic plasticity' is much preferred above 'learning', as it better describes what is at hand: long- and short-term changes in how synapses process signals [1,13,19].

Single spikes do not carry any information, it is their exact timing and placement amongst other spikes that matters. These signals are delayed during their journey through axons, synapses and dendrites and as their effect decays over time they provide the neuron a form of inherent notion of time and memory. This property is unique to neurons that employ pulse coding. It provides the network the capability to incorporate spatial-temporal information from sensory input directly into communication and computation [19,20]. Examples include the multiplexing of frequency, direction and amplitude of sound [5], as can be seen in barn owls.

A number of different models have been proposed that employ exact spike timing information in neural computation. Most of these fall in the class of integrate-and-fire neurons, a well-chosen name that describes exactly what these neurons do: integrate incoming signals and fire when a threshold is crossed. We've used the spike-response model that is based on the very realistic and complex Hodgkin-Huxley [5,7] model. It does not describe the exact chemical-concentrations in the neuron, but remains an abstraction of the neuron's dynamic state, making the spike-response model conceptually easy, computationally fast and biologically realistic.

As we've seen, all action potentials are basically of the same form. In other words, as their form does not carry any information, we can characterise them by the single property that does carry information: the firing-time $t_i^{(f)}$. The lower index i indicates the neuron, the upper index f the number of the spike. All the spikes a neuron has generated over time can then be captured by

$$F_i = \{t^{(1)},...,t^{(n)}\} \tag{1.1}$$

To describe the membrane potential of a neuron i we commonly use the variable u_i. Once the neuron's membrane potential goes over threshold value ϑ, the neuron will spike. This new spike will then be added to F_i, defining this set as

$$F_i = \{\, t \mid u_i(t) = \vartheta \wedge u_i'(t) > 0 \,\} \tag{1.2}$$

Once a neuron generates an action potential, its membrane potential suddenly increases very sharply, soon followed by a long lasting drop: the negative after-potential (see fig. 1b). The sharp rise above the threshold value ensures that the neuron cannot generate a new spike and is called absolute refractoriness. The following long-lasting decrease in membrane potential, called the negative spike after-potential, decreases the chances for the neuron to fire again. We can model this absolute and negative refractoriness with kernel η:

$$\eta(s) = -n_0 \exp\left(-\frac{s-\delta^{abs}}{t}\right) H(s-\delta^{abs}) - KH(s)H(\delta^{abs}-s) \tag{1.3}$$

$$H(s) = \begin{cases} 1 \text{ if } s > 0 \\ 0 \text{ if } s \le 0 \end{cases} \tag{1.4}$$

The duration of the absolute refractoriness is set by δ^{abs}, during which large constant K ensures that the membrane potential is vastly above the threshold value. Constant n_0 scales the duration of the negative after-potential. Now that we can describe what happens to a neuron once it fires, we need to be able to describe the effect on the membrane potential of postsynaptic potentials.

$$\varepsilon_{ij}(s) = \left[\exp\left(-\frac{s-\Delta^{ij}}{\tau_m}\right) - \exp\left(-\frac{s-\Delta^{ij}}{\tau_s}\right)\right] H(s-\Delta^{ij}) \tag{1.5}$$

In equation 1.5, we can use Δ^{ij} to define transmission delays (axons and dendrites are relatively fast, synapses slow) and $0<\tau_s<\tau_m$ are time constants defining the duration of the effect of the postsynaptic potential. The synaptic efficacy or synaptic weight, effectively resulting in the strength of the post-synaptic potential, is captured by variable w_{ij}.

We are nearly done, having all the means to describe a full spiking neural network, though one very important feature is lacking: how to handle information from the outside, information not encoded in spikes. Sensory information gathered by a robot is usually depicted in discrete integer values, with which we'll have to influence the membrane-potential directly. We use a task-specific function $h^{ext}(t)$ to transform these values to relevant membrane-potential influences.

$$h(t) = h^{ext}(t) + \sum_{j \in \Gamma_i} \sum_{t_j^{(f)} \in F_j} w_{ij}\varepsilon_{ij}(t - t_j^{(f)}) \tag{1.6}$$

With this addition neurons can become excited by outside influences, effectively transforming analogue input values into the signal the network can process: a spike. We can now describe the current excitation of a neuron by

$$u_i(t) = \sum_{t_i^{(f)} \in F_i} \eta_i(t - t_i^{(f)}) + h(t) \tag{1.7}$$

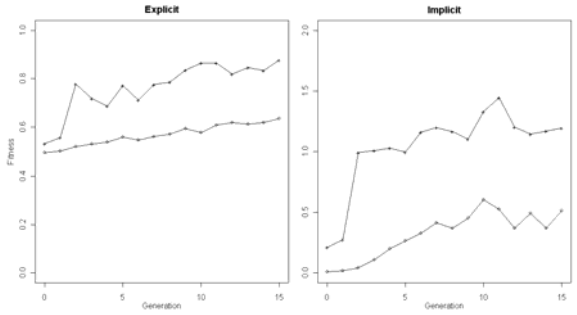

Figure 3. Proximity experiments: explicit and implicit fitness functions. o = average fitness of whole population; + = fitness of best individuals.

where the refractory state, effects of incoming postsynaptic potentials and external events are combined. Once combined with equation 1.3 it forms the spike-response model, a powerful yet easy to implement model for using exact spike-timing information in computation of neural networks.

Evolving spiking circuits on a robot

In order to use the spike response model for our experiments, some simplifications to the standard model were made in order to avoid overly large genomes and limit the amount of computation needed. Our derivation of the model is mainly based on the model described by Floreano and Mattiussi [4]. The software we used is *i* [12], an application that has been designed in collaboration with ASL2 for evolutionary robotics with neural networks.

A first important note is that time has been made discrete: continuous time is assumed in the spiking response model, but discrete time steps were introduced to make implementation easier. Neurons are updated each time step, but input and output are respectively set and read only at specified intervals (more details below).

A spike is assumed to last one time step, after which one time step of absolute refractoriness holds. After this, the refractory kernel is computed for each neuron using a simplified version of equation 1.3

$$\eta(s) = -\exp(-\frac{s}{\tau_m}) \qquad (2.1)$$

where s is the number of time steps since the last spike time of this particular neuron, as we take into account only the refractory kernel of the last spike for each neuron. The formula for postsynaptic contributions of incoming spikes (1.5) remains the same, although only incoming spikes of the last 20 time steps are taken into account.

We will now describe the architecture of the networks and the handling of in- and output, as both of these are unconstrained in the general model. We distinguish receptor neurons and interneurons: receptor neurons are used for feeding input (sensor data) into the network and only have outgoing synaptic connections; interneurons may have both incoming and outgoing synaptic connections and thus can be fully interconnected and recurrent. The number of receptor neurons is provided beforehand by the task; the number of interneurons is at least the number of required output values, as one interneuron is used for each output. We added one additional (bias) receptor neuron with a constant input of 1 to all networks to ensure basic network activity.

In our experiments, all neurons (receptor and interneurons) in the network are updated each time step, but a certain number of these updates together form a sensorimotor cycle. For the experiments described in this paper, the number of updates per cycle was fixed to 20. At the first update of a cycle, new input values (in the range [0,1]) are fed into the receptor neurons. At each update, each receptor neuron stochastically determines its $h^{ext}(t)$, based on the current input:

$$h^{ext}(t) = flip(input) \qquad (2.2)$$

where *flip*() returns 1 with probability equal to its argument, 0 otherwise. As a membrane potential of 1 crosses the threshold, this would normally gives a spike (please note that receptor neurons also have a refractory period; the maximum fire rate for all neurons is 0.5).

It is clear that rate coding is used for encoding the input; the same coding method is used to determine the output of the network. For each interneuron that is used for output, all spikes during the last n updates of a cycle are counted and divided by the maximum number of spikes that could have occurred, giving a value properly scaled between 0 and 1. The number of updates at the end of each cycle that contribute to the output value is always the same in our experiments: the spikes in the 20 last neuron updates of a cycle determined the output value (as the number of updates per cycle is also 20, this means each update is used for output calculation).

Now that we know how one network cycle is done, coupling the network with the sensors and motors is not very complicated: at the start of a sensory-motor cycle, the sensors are read and the returned values are used as input for the network. At the end of the cycle, the output given by the network is used to set the motor speeds of the robot. This is done in a push-pull way: two outputs are used to determine one motor speed, subtracting one output from the other to get a value in the range [-1, 1]. Each cycle has a fixed length of 100ms, therefore sensors are read and motor speeds are set 10 times a second and one lifetime of 30 seconds consists of 300 cycles.

Although we used evolution to search for network parameters as much as possible, we chose some parameters fixed to avoid overly large genomes and being unable to evolve anything useful from such a huge parameter-space. Let us first discuss the fixed parameters before proceeding with the genotype-phenotype mapping. The threshold ϑ was set to 0.5, which means that only a few recent incoming spikes may evoke a spike. The synapse and neuron time constants, τ_s and τ_m, were set to 10 and 4, respectively; the synaptic delay was set to 2 network updates, synaptic strength is fixed to 1.

We now arrive at the genotype-phenotype encoding. The genome, a binary string, consists of a certain number of blocks, each block encoding one spiking interneuron. The first bit of each block encodes whether the interneuron is excitatory or inhibitory: a spike coming from an excitatory synapse adds to the membrane potential, the opposite for inhibitory connections. The remaining bits of each block encode the existence (i.e., whether there is a synaptic connection or not) of incoming synapses from all receptors and interneurons.

An important parameter in the experiments we haven't discussed yet is the fitness function, which very much depends on the task to be evolved. We will now describe the experimental setup used, the tasks and the fitness functions.

Experimental setup

We already mentioned before that we used the Khepera robot for our experiments. The Khepera is a robot with two wheels that can be controlled separately, making it very easy to manoeuvre the robot. A standard Khepera is equipped with wheel encoders that can read the actual wheel speeds and infrared sensors that give proximity readings. For the

Experiment	SNN	Fitness	Behaviour			
	#inter	Gen12	Solutions?	# generations	Straight & turn?	Wall-following?
Proximity						
Implicit	0	0.56 / 0.76 [1)]	Few	12	No	No
Explicit	0	0.62 / 0.83	Many	12	Yes	No
Vision						
Implicit, Laplace	0	1.02 / 1.76 [1)]	Some	20	Yes (not perfect)	Yes
Explicit, Laplace	0	0.57 / 0.83	Many	12	Yes	Yes
Explicit, no filter	4	0.51 / 0.84	Some	18	Yes	No
Explicit, no filter	2	0.50 / 0.53	Some	25	No	Yes
Explicit, no filter	0	0.50 / 0.52	Some	60	Yes	Yes

Table 1. Overview of all results. Properties of each experiment are given (proximity or vision; implicit or explicit fitness function; Laplace filter or no filter, raw vision), also the number of additional interneurons added to the spiking neural networks (SNN: #inter). Under Fitness, the fitness values of the 12th generation of each experiment is shown (average of whole population / fitness of best individual). Under Behaviour, we find how well the resulting individuals perform: whether good solutions have been found (Solutions?), the number of generations necessary to obtain good solutions (# generations) and which strategies have been found by evolution (Straight & turn? and Wall-following?). [1)] Please note that these values cannot be compared with the values given by the explicit fitness function.

vision experiments, we used the so-called kevopic turret, a generic I/O add-on for the Khepera, in combination with a TSL3301 camera. This camera gives an 8-bit greyscale image of 102 pixels with a 150° field of view; 16 equally spaced pixels are selected from this linear image.

Proximity – The objective of the first series of experiments was to evolve spiking controllers that show simple Braitenberg behaviour [3] using only proximity readings as network input. The behaviour we intend the robot to show is that of some of the simplest vehicles Valentino Braitenberg [3] described: drive straight ahead unless a sensor senses something nearby. With proximity sensors this means that the robot has to drive straight ahead until a wall is encountered and the Khepera should steer away. Each of the 8 IR proximity sensors was used to feed input into one of the 8 receptor neurons of the networks, 4 interneurons were used to determine the wheel speeds of the two wheels in push-pull mode (as described above). The environment was a wooden arena with white walls and floor of 50 x 60 cm.

Vision – The second series of experiments were conducted with the Khepera equipped with the kevopic turret and TSL3301 camera. We used the same arena as in the proximity experiments, but added randomly spaced vertical black and white stripes to the walls and made sure lighting conditions were always the same. The 16 vision greyscale values were used to determine the input for 16 receptor neurons, but this sensor data was presented in different ways: in the first experiments, a Laplace filter was applied to the raw vision data and the resulting contrast values were used as network inputs. In later experiments, no filter was used and the raw 8-bit vision values were fed into the network directly after scaling into the range [0,1]. As before, 4 interneurons were used to determine the wheel speeds in push-pull mode. In these experiments, the objective was to evolve controllers able to show the same behaviour as in the proximity experiments: using only vision, the Khepera was asked to navigate around the arena with black and white striped arena as fast as possible without hitting the wall.

In almost all experiments, only the 4 interneurons used to determine wheel speeds were used, but additional interneurons were added in a few exceptions (which will be mentioned later). At the start of each epoch of an individual, a pre-programmed Braitenberg behaviour (based on infrared proximity-sensor data) was executed for one second to start with a more or less random starting position.

We mentioned that we used two different fitness functions for the experiments, which we called implicit and explicit versions of more or less the same function; essential is that both are functions of the measured actual speeds of the robot wheels. Both fitness functions have the same general form:

$$fitness = \frac{\sum_t f(t)}{T} \tag{3.1}$$

Fitness $f(t)$ is computed at the end of each sensorimotor cycle t, summed for all cycles and divided by the total number of cycles T (of one life time). The implicit version of $f(t)$ we used is:

$$f(t) = \begin{cases} 0 & s_l < 0 \vee s_r < 0 \\ s_l + s_r & otherwise \end{cases} \tag{3.2}$$

where s_l and s_r are the actual wheel speeds of the left and right wheel, respectively, normalised in the interval [-1, 1]. This function rewards an individual only when none of the wheels is moving backward and is equal to the function used by Floreano & Mattiussi [4]. The more explicit version:

$$\begin{aligned} f(t) &= 0.1 * drive(t) + 0.5 * rotate(t) \\ &\quad + 0.4 * forward(t) \\ drive(t) &= \frac{|s_l| + |s_r|}{2} \\ rotate(t) &= 1 - \frac{|s_l - s_r|}{2} \\ forward(t) &= \frac{\max(s_l, 0) + \max(s_r, 0)}{2} \end{aligned} \tag{3.3}$$

This fitness function is divided into three terms: the first term rewards all wheel movement and is meant to encourage all motion; the second term gives a penalty when the robot rotates, as we want individuals moving straight ahead; the last term rewards all forward motion, as we prefer individuals moving forwards. This function is more explicit, because it gives more 'hints' as to what the robot should do. The implicit function only says to drive forward, this function is more explicit by rewarding moving forward and penalising rotation, hopefully making the fitness landscape smoother and therefore making it easier for evolution to find good solutions.

Results

We ran quite a few experiments (running continually day and night) in order to try different experimental settings and to see whether we could repeatedly obtain the same results. Due to time limitations, we weren't able to repeat all experiments more than once, and therefore we are unable to use average fitness values in the results of each experiment. We picked a number of experiments that gave realistic

results and of which we are sure that we would achieve something similar if done again.

Some of the experiments had to be discarded because of erroneous results, an example is an evolutionary run where the fitness values were higher than the theoretical maximum quite often; although the communication with the Khepera is fairly robust, communication went wrong during a few runs, resulting in these illegal fitness values (i.e., incorrect actual motor speeds were received by the software).

The results of the selected experiments are summarised in Table 1. The series of proximity experiments consist of two experiments of 15 generations, though we repeated both experiments and found identical results in the repeats. All settings except for the fitness function were equal, which clearly shows in the fitness graphs (fig. 3): in case of the explicit function, the values start at 0.5 because individuals that do (almost) nothing don't get a penalty for rotation and all values are between 0 and 1, while the implicit function starts at 0 and gives values between 0 and 2. (Of course, these could also be scaled to the range [0, 1], but it would remain impossible to compare.) This doesn't tell us much more than that comparing absolute values is useless though; comparing evolutionary results is much more interesting. Looking at the graphs, we see that the fitness values change more or less the same over the generations: the maximum fitness first increases fast, but remains pretty stable after that, while the average fitness increases slowly but steady.

The two graphs displayed in figure 3 are typical for all achieved results: the maximum fitness sometimes increases with large steps, but can remain the same for a long while as well and can even drop sometimes, the average fitness always increases slowly. When fitness values change like this, one can be quite sure that the task is evolvable; at least evolution is able to find individuals with higher fitness values. Naturally, this doesn't guarantee the target behaviour, as designing a good fitness function remains a difficult issue.

The vision experiments were allowed to run for at least 20 generations, but the more complex experiments (without Laplace filter) were allowed a longer running time to see whether this could improve results. Fitness values of the 12th generation of each experiment are given in Table 1 and can be compared: it is clear that using a Laplace filter helps individuals a lot, as the maximum fitness of the 12th generation without input filter and without additional interneurons is only 0.52, not at all like the 0.83 when the Laplace filter is used! On the other hand, adding interneurons makes it easier to find good solutions, as the 12th generation of the experiment without Laplace and with 4 extra interneurons has a maximum fitness of 0.84, which we may consider the same.

It is therefore interesting to compare the trajectories of the maximum fitness of the experiments where raw vision data was used (fig. 4). With 4 additional interneurons, the maximum fitness increases pretty fast and evolution doesn't seem to have problems finding solutions that achieve high fitness values. Every time we remove two of these interneurons though, evolution needs more time to find solutions that have a higher fitness: more than 50 generations are required when no additional interneurons are added and thus only the 4 interneurons that are used for output are available. It is remarkable though that solutions are found whatsoever: we shouldn't forget that raw vision is used as input, that only 20 network updates a cycle are done and that only 4 interneurons are in the network! The important question is of course whether the solutions found in this case show a behaviour that is comparable with that of the traditional experiment (with Laplace filter).

In order to be able to compare the behaviours resulting from the various evolutionary runs, we now arrive at a very important part: the analysis of the behaviour of evolved individuals.

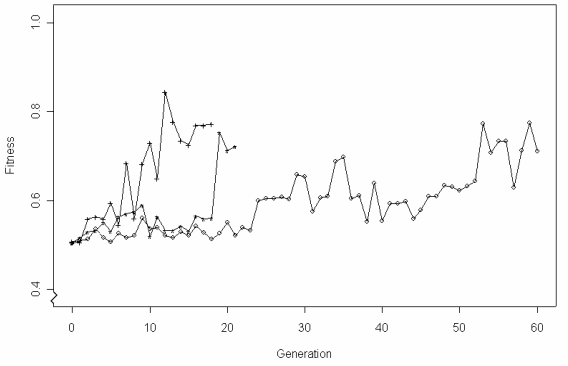

Figure 4. Vision experiments without input filter. Drawn is the fitness value of the best individual of each generation for three different evolutionary runs: + = 4 extra interneurons; * = 2 extra interneurons; o = no extra interneurons.

Behavioural analysis

Let us first introduce some of the typical behaviours that we have observed while watching the evolved individuals. Naturally, evolution does not provide only satisfying results: in almost every generation, individuals exist that do nothing, rotate on the spot or drive ahead as fast as possible without reacting on anything (such as a wall or even a collision). We will forget about these and focus on the good behaviours that we obtained, which can generally be divided into four categories (see fig. 5).

A first simple – but efficient – strategy that we observed is what we called 'Big turn, small turn': the robot turns either clockwise or counter-clockwise all the time and all it does is rotate (almost) on the spot when a wall is detected. This behaviour isn't entirely what we intended, as we wanted the robot to move straight ahead as much as possible.

A second strategy is to drive straight ahead until a wall is detected and the robot turns around about 180° immediately. This is very much like the third strategy, where the robot also drives straight ahead but doesn't turn around fully, it only turns away from the wall.

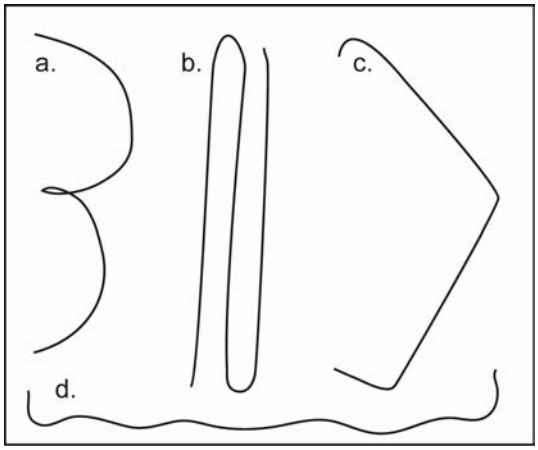

Figure 5. Four typical evolved behaviours. Approximations. Distance before turning from the wall differs with vision: bad individuals bump, good individuals hold a distance of between 5 and 10cm. (a) Big turn, small turn. (b) Drive straight, turn around. (c) Drive straight, turn away. (d) Wall-following.

The fourth and last strategy is very different from the previous ones, as it always actively uses the walls: it stays near the wall and follows it. Whereas the other strategies don't care where the wall is (they just get away from it when necessary), individuals using this strategy even actively seem to 'search' for the wall and stay near it as soon as it is found. The resulting behaviour is that the robot drives nearby all walls of the arena, not getting too far away from them but certainly not actively avoiding collision.

Now we know the behaviours we can expect, let's focus on the experiments. Table 1 displays for each experiment how many good strategies were evolved, how many generations were needed for this and whether the second/third (straight & turn) and fourth (wall-following) strategies were found.

When we take a look at the proximity experiments, we immediately see that although both fitness functions seemed to perform well accordingly to the fitness graphs, the resulting behaviours the give rise to vary. In case of the implicit fitness function, only a few well-behaving individuals were evolved and individuals seem to be rewarded for the wrong thing: many individuals simply drive straight ahead until they collide. As the starting position is often turned away from the wall, this gives a relatively high fitness value. The explicit fitness function resulted in much better individuals; certainly seen over the whole 12th generation this function did a significantly better job when we look only at behaviour. Many individuals are able to drive straight ahead and turn away just before (in some cases when against) the wall. As the sensivity and range of the proximity sensors isn't large, it's rather difficult to obtain better results: the sensors cannot detect the wall until the robot is only a few centimetres away from it. We haven't observed any wall-following individuals, which may be due to the short range of the proximity sensors: this makes it difficult to actively use the walls to stay close to them without colliding.

When we compare the results of the vision experiments with the two fitness functions (with Laplace filter), we see that although good individuals were evolved in both cases, the explicit function performed better again. Less generations were needed and more behaviours were found; less individuals that do 'big turn, small turn', more that do 'straight & turn' or 'wall-following'. Since so many well-behaving individuals are found when using a Laplace filter and the explicit function, the task seems rather easy and very simple input processing strategies (like 'turn whenever there is contrast, straight ahead otherwise') may be sufficient to accomplish the task at hand.

Using raw vision data as input for the spiking circuits seems a more challenging task for evolution though, and when we look at the behavioural results it looks like this is true. Where evolution needed no additional interneurons to evolve many good individuals in 12 generations when a Laplace filter was applied, already 18 generations were needed to evolve good individuals when no input filter was used and 4 interneurons were added.

When we reduced the number of additional interneurons, the number of generations required to achieve high fitness values increased fast. But when high fitness values were obtained, well-behaving individuals were also found! And this happened also when we added 2 or even no interneurons. Not as many good individuals were found as with Laplace filter and explicit fitness function, there were some nevertheless. In the run with 2 additional interneurons, no good 'straight & turn' individuals were found and the same for no additional interneurons and 'wall-following', but we think that this is because the task is harder and a bit more luck is needed to find a task; we could probably fill the 'gaps' by doing a few more runs.

Discussion

In this paper we have shown that circuits of spiking neurons are well suited for evolving neural controllers for vision-based navigation. We've extended the preliminary work done by Floreano and Mattiussi and have found results in agreement of theirs. More importantly, we found that these networks of spiking neurons are capable of more than solving the basic vision task they describe: even without the help of a Laplace (contrast) filter and with significantly less neurons artificial evolution finds behaviourally very fit individuals that solve the task of navigating an environment with walls randomly sized black and white stripes.

From our results we can conclude that this task is very simple if the network is provided with pre-processed vision data such as contrast-values: as early as generation 12 in each and evolutionary run, individuals are found that show very good behaviour. (We have even noticed good individuals in the first generations of some of our experiments, which we consider exceptions based on luck.) Without such a filter we see that the task becomes quite a tad more complex, as the behaviour shown in the 12th generation is far less well developed. However, if we allow evolution more time we see that individuals are found that show excellent behaviour, even if we severely limit the number of interneurons.

Due to constrains on both time and robots we have not been able to repeat our experiments as often as we would've liked. We have seen that by using a more explicit fitness function we can more effectively steer evolution towards desired behaviour. This was illustrated by our experiments in which the robot was asked to navigate just by infrared proximity information and even more so in the various vision experiments. By using explicit fitness functions one can punish undesired behaviour that could otherwise be well rewarded, i.e. circling at high speed with large radius at high speed.

The genetic encoding used in this experiment decreases the search space of evolution dramatically; each individual has to be tested on a real robot, larger genomes could possibly take many more generations before target behaviour is found. However, better results might be acquired by allowing single synapses, instead of all from one neuron, to be excitatory or inhibitory. Single neurons can then single-handedly differentiate the behaviour of others by operating as feature detectors. For harder tasks, it may be a necessity to evolve even more network parameters that were fixed now.

Future work will include the usage of online learning methods. Spike-timing dependent synaptic plasticity (STDP) is a form of competitive Hebbian learning that uses the exact spike timing information [1]. Neurological experiments show that long-term synaptic strengthening occurs when pre-synaptic action potentials arrive shortly before a postsynaptic spike and weakening when it arrives late. This mechanism leads to stable distributions of long-term potentation and depression, making postsynaptic neurons sensitive to the timing of incoming action potentials. This sensitivity leads to competition among the presynaptic neurons, resulting in shorter latencies, spike synchronization and faster information propagation through the network [1]. It is our opinion that this might lead to better, more differentiated behaviour at possibly lower network resolutions.

Spiking neural networks are computationally powerful [7,15,19] and very promising in regard to real-world tasks in which temporal information plays an important role, which we have shown by evolving circuits that can navigate a robot using raw vision only. We conclude with the remark that although a lot of progress has been made on the theory behind spiking neural networks, more effort is required to gain knowledge and experience on how to apply these networks effectively in real applications like robotics.

Acknowledgements
Our special thanks go out to Dario Floreano and Jean-Christophe Zufferey of the Autonomous Systems Lab at EPFL, for kindly lending us a Khepera, helping us tremendously with the hardware and all other support. Furthermore we would like to thank John-Jules Meyer and Marco Wiering for supporting us in all possible ways.

References

1. Abbot, L. F. & Nelson, S. B. *Synaptic Plasticity: taming the beast*, Nature Neuroscience Review, vol. 3 p.1178-1183 (2000).
2. Bohte, S.M, Kok, J.N. & La Poutré, H. *Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons*, Neurocomputing preprint (2000).
3. Braitenberg, V. *Vehicles*, MIT Press, Cambridge, MA (1984).
4. Floreano, D. & Mattiussi, C. `Evolution of Spiking Neural Controllers for Autonomous Vision-Based Robots'. In Gomi, T., ed., *Evolutionary Robotics. From Intelligent Robotics to Artificial Life*. Tokyo: Springer Verlag (2001).
5. Floreano, D., Schoeni, N., Caprari, G. & Blynel, J. *Evolutionary Bits'n'Spikes*, Proc. of the 8th Int. Conference on Artificial Life (2002).
6. Gerstner, W. *Spiking Neurons* in Maass, W. & Bishop, C. M. (eds.) *Pulsed Neural Networks*, MIT-press (1999).
7. Gerstner, W. & Kistler, W. *Spiking Neuron Models*, Cambridge University Press (2002).
8. Goldberg, D.E. *Genetic Algorithms in Search, Optimization and Machine Learning*, Reading, MA: Addison-Wesley (1989).
9. Iida, F. Goal-directed Navigation of an Autonomous Flying Robot Using Biologically Inspired Cheap Vision, Proc. of the 32nd ISR, p. 1404-1409 (2001).
10. Koopman, A., Van Leeuwen, M. & Vreeken, J., *Dynamic neural networks, comparing spiking circuits and LSTM*, Tech. Report UU-CS-2003-007, Institute for Information and Computing Sciences, Utrecht University (2003).
11. Kruger, J. & Aiple, F. *Multimicroelectrode investigation of monkey striate cortex: spike train correlations in the infra-granular layers,* Journal of Neurophysiology vol. 60, p.798-828 (1988).
12. Van Leeuwen, M., *Evolutionary blimp & i.* Internship report, Institute for Information and Computing Sciences, Utrecht University (2002).
13. Maass, W., Schnitger, G. & Sontag, E. *On the computational power of sigmoid versus boolean threshold circuits*, Proc. of the 32nd Annual IEEE Symposium on Foundations of Computer Science, p. 767-776 (1991).
14. Maass, W. *Synapses as Dynamic Memory Buffers*, Technische Universität Graz (2000).
15. Maass, W. & Bishop, C. M. (eds) *Pulsed Neural Networks*, MIT Press, Cambridge, MA (1999).
16. Mondada, F., Franzi, E. & Ienne, P. *Mobile robot miniaturization: A tool for investigation in control algorithms,*In T.Y. Yoshikawa & F. Miyazaki (eds.), Proc. of the 3rd Int. Symposium on Experimental Robots. Berlin, Springer-Verlag (1993).
17. Nolfi, S. & Floreano, D. *Evolutionary Robotics: Biology, Intelligence and Technology of Self-Organizing Machines*, MIT Press, Cambridge, MA (2000).
18. Rumelhart, D.E., Hinton, G.E. & Williams, R.J. *Learning representations by back-propagating errors,* Nature Vol. 323 (1986).
19. Thorpe, S., Delorme, A. & Van Rullen, R. *Spike based strategies for rapid processing*, Neural Networks, vol. 14(6-7), p.715-726 (2001).
20. Vreeken, J. *Spiking neural networks, an introduction.* Tech. Report UU-CS-2003-008, Institute for Information and Computing Sciences, Utrecht University (2002).
21. Williams, R.J. & Zipser, D. 'A Learning Algorithm for Continually Running Recurrent Neural Networks'. In: *Neural Computation*, 1, pp.270-280 (1989).
22. Zufferey, J.C., Floreano, D., Van Leeuwen, M. & Merenda, T. 'Evolving Vision-based Flying Robots'. In: Bülthoff, Lee, Poggio, Wallraven (eds), *Proceedings of the 2nd International Workshop on Biologically Motivated Computer Vision*, LNCS, Berlin, Springer-Verlag (2002).