Bio-inspired Adaptive Machines

"Evolution of vision-based navigation Simulation using a simple, ultra-fast simulator"

May 2004 - EPFL

Urs Keller
Alok Menghrajani

Abstract
The project consists in exploring how various parameters (like field-of-view, number of pixel, arena shape, quantity of contrast in environment, etc) affect the evolution of vision-based navigation. The objective is to find suitable parameters in order to evolve controllers for a Khepera in a few different environment types.

Details of the simulations
We used the goevo 1.0 simulator to test different controllers for a robot. The controllers are based on 744 genes, and are evolved using genetic algorithms. The objective for the robot is to move the fastest possible in the virtual world (the worlds are 2D "rooms" with walls). Based on the motor speeds, each controller receives a fitness value (the greater the average motor speeds is, the greater the fitness value will be), and the genetic algorithm will try to optimize this fitness function by modifying the genes. The robot receives as an input 25 pixels from a linear camera, which's field of view (FOV) can be set.
Note: We dropped the infrared sensors experiments due to problems with the simulator.

We ran each simulation for 200 generations. Since such a simulation takes about 45 minutes, we used multiple computers at the same time to gather our data. The complete simulation results are available on our DVD, along with video files demonstrating some of our controllers.

Note: It is not possible to achieve a fitness of 1.0, because that would correspond to a robot going straight at full speed. Unfortunately there is no easy way (e.g. based on user input) to know what the approximate maximum fitness for a given world is.

Parameters for the simulator
The following parameters can be modified in the simulator:

Robot's options:

- Field of view. We worked in the range 20°-130°.
- Number of pixels the robot receives as input. We kept this one fixed to 25.
- Preprocessing, three options: raw vision, spatial difference and mean spatial difference.
- Which world files to use. We used rect_high.world and corridor.world.

Experiments and neural network options:

- We left everything as explained in the hand out.

Evolution settings:

- Max. number of generations: We always ran the simulations up to the 200th generation. This way we were able to always collect complete data.
- Number of epochs: The number of epochs is the number of times each individual is run to evaluate its fitness. We set this to 2, but then realized our controllers laked stability, so we reran some simulations with an epoch of 20.
- Life time (in seconds): 40.
- Cycle time (in ms): Lowering this time will increase the rate at which the camera/sensors run. We used the default value of 100.
- Evolution mode: We used simulation to generate the datasets and normal to look at the behavior of specific controllers.
- Mutation rate: we used values in the range 0.001 – 0.500.
- Crossover rate: we used the default value 0.1.

Experiment 1 : The effect of the field of view parameter

Our first experiment is to see how the field of view parameter affects the robot.

We performed 2 times 7 runs, with the following FOV values:
20°, 40°, 60°, 80°, 100°, 120° and 130°.
Note: the configuration file has the angle in radians divided by two (half of the FOV).

We had the default mutation rate: 0.05

We used the following two seeds for the random number generator:
1083150669, 2083250669

Table 1 shows our results. The second column shows the number of generations we had to wait for before a given controller performed better than 0.80. The third column shows the number of generations we had to wait for before a given controller performed better than 0.85. These values represent controllers that usually are good enough; i.e. controllers that don't get stuck in walls.

Figure 1 is a plot of the best fitness after 200 generations vs. the FOV. You can clearly see that the optimal FOV is somewhere around 80°.

Figure 2 shows the fitness value as the generations is evolved (using the data from the first dataset). This chart is hard to read, but it basically shows that each generation is not necessarily better than the previous one and that a FOV too narrow or too broad will prevent the generations from being able to rapidly evolve.

What happens when the FOV is too narrow? Imagine that the FOV is very small (e.g. 0.1°). The robot only sees what's exactly in front of it. Since the only way the robot can tell the distance to a wall is by analyzing the white-black-white pattern (by analyzing the width of the black pattern), by having a small

FOV	fitness > 0.80	fitness	fitness > 0.85	fitness	best	fitness
FIRST RUN						
20	78	0.802			128	0.822
40	14	0.826	15	0.864	139	0.910
60	8	0.809	13	0.851	53	0.916
80	5	0.809	8	0.858	188	0.924
100	6	0.809	9	0.854	61	0.924
120	8	0.808	13	0.894	86	0.916
130	11	0.824	15	0.850	67	0.897
SECOND RUN						
20	15	0.849	184	0.851	184	0.851
40	11	0.825	80	0.883	111	0.893
60	8	0.818	10	0.851	54	0.912
80	8	0.834	9	0.868	64	0.920
100	5	0.844	6	0.858	78	0.924
120	8	0.817	11	0.876	199	0.904
130	14	0.806	18	0.874	198	0.904

Table 1 – Summary of the FOV dataset.

FOV, the robot cannot tell the distance to the walls. What about when the FOV is too large? Since the robot's camera is always 25 pixels, a larger field of view will reduce the "resolution" of the camera (in terms of pixels per angle of vision). When the FOV gets too large, the robot doesn't really see anything specific.

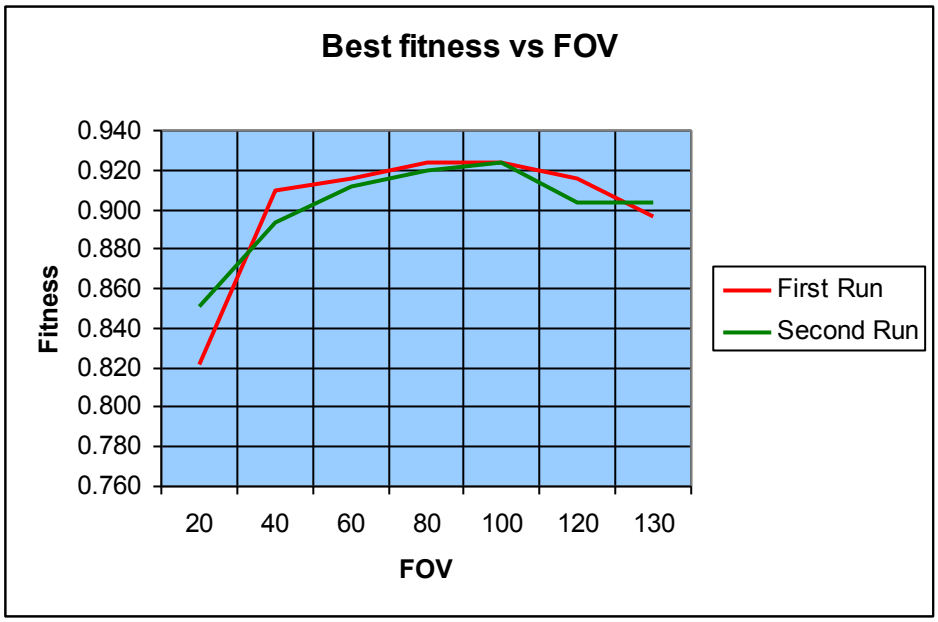


Figure 1.

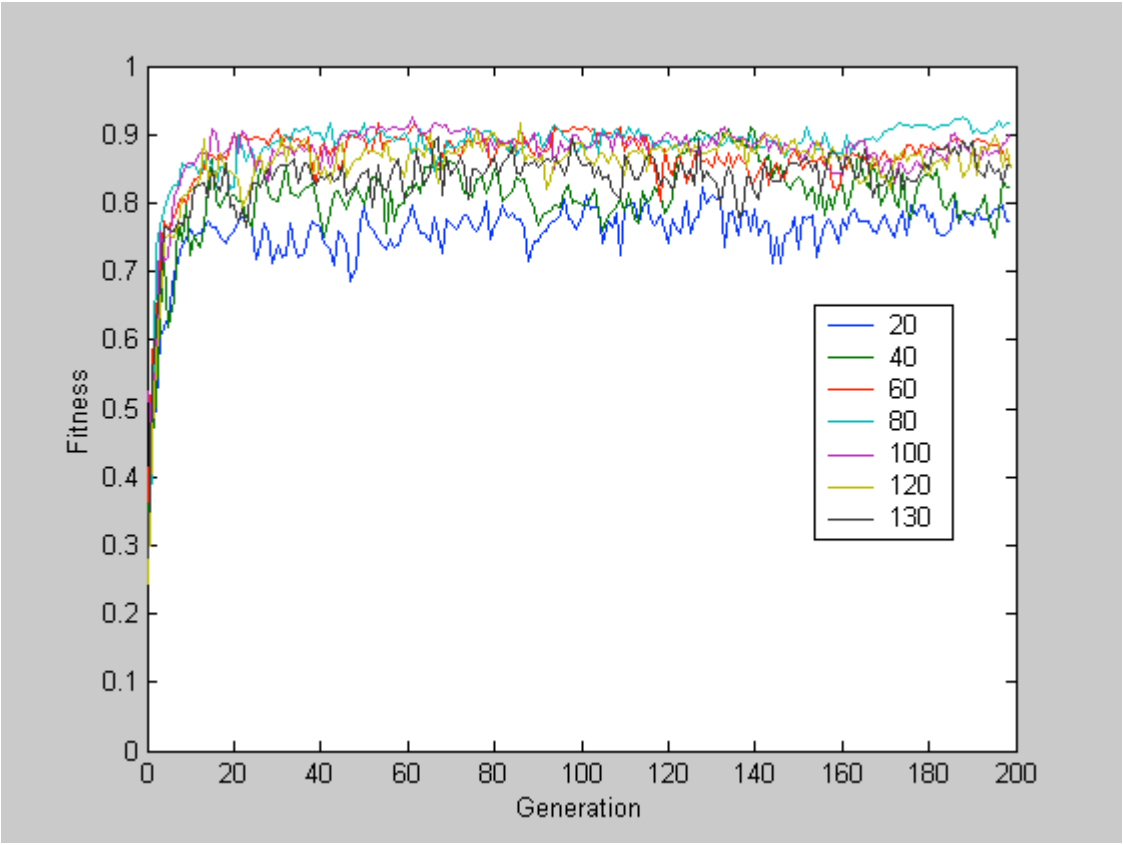


Figure 2 – Evolution of the fitness for different FOV values.

Once we have collected the dataset, we can test specific controllers in different conditions. We noticed that running a given controller multiple times gives us different fitness values. This is due to the way the random seed is implemented in the simulator. It is possible for a controller to have very different fitness values from one run to another (we had values ranging from 0.053 to 0.592). It turns out these large differences of values are due to the fact that the robot gets stuck in walls. If this happens at the beginning of the simulation, the fitness value will be very low. If this happens near the end of the simulation, the fitness value will be higher. So depending on when the robot gets stuck in the wall, the fitness value can be very different.

In order to reduce these random effects we decided to test the individual controllers using 100 runs.
We also tried to obtain more robust controllers by rerunning the 7 different FOV experiments with an epoch of 20. It turns out this is what we should have been doing right from the beginning!

It is important to understand that the robot has a memory (the neural network is not purely reactive). It is therefore possible that the robot learns a specific path without relying on the sensors values. Since the robot is reset at a specific position before each run, we should check how the robot performs when starting at a different position. However in order for the robot to be able to learn a path, it needs "lots" of memory. Our simple neural networks simply don't have enough neurons to allow this to happen. Table 2 shows the results of our comparison of the fitness of the best controllers based on different starting positions.

FOV	(30, 30)	(25, 30)	(25, 35)
20	0.549	0.627	0.573
40	0.591	0.636	0.588
60	0.476	0.530	0.474
80	0.880	0.846	0.869
100	0.784	0.751	0.713
120	0.445	0.479	0.422
130	0.733	0.664	0.526

Table 2 – Testing the "memory" of the network.

We also experimented to see if a controller evolved for a particular FOV could be used with a different FOV. We ran each best controller of the previous runs with each possible FOV. Table 3 shows the results of the 40 runs in a matrix form. As you can see, it is possible to use a controller programmed for a larger FOV, but not the other way round.

Controller / Lens	20	40	60	80	100	120	130
20	**0.549**	0.094	0.062	0.057	0.056	0.055	0.054
40	0.732	**0.591**	0.161	0.072	0.057	0.055	0.058
60	0.519	0.854	**0.476**	0.110	0.069	0.059	0.059
80	0.109	0.706	0.886	**0.880**	0.317	0.194	0.148
100	0.108	0.715	0.871	0.842	**0.784**	0.512	0.377
120	0.214	0.645	0.797	0.879	0.724	**0.445**	0.318
130	0.216	0.559	0.755	0.861	0.864	0.804	**0.733**

Table 3 – Cross running controllers with different FOV values.

Experiment 2: The mutation rate parameter

It is very intuitive that there exists an ideal mutation rate for our problem. If we could represent the fitness value as a function of the genes, we would see a very complex function. That means the fitness function has got many local maximas, but we want to find the absolute maximum. A small mutation rate will imply that we will get "stuck" in local maximas. A large mutation rate will imply that we will randomly "hop" from one point to another in the function's domain.

Again we ran 2 complete runs with 8 different mutation rates:
0.001, 0.071, 0.141, 0.210, 0.280, 0.350, 0.420, 0.500.

We used the default FOV value: 70°

We used the same two seeds for the random number generator:
1083150669, 2083250669

Table 4 shows the results we obtained. It is very hard to conclude what the idea parameter is, because we have too little data (in order to obtain useful data we would need to run the experiment with more mutation rates and a much greater number of times). So besides saying that Figure 3. illustrates clearly what happens when the mutation rate is too high, we can't say much. The ideal mutation rate is probably somewhere near 0.1.

A lower mutation rate can be compensated by a larger population or a larger cross-over rate. By increasing the population size, we are increasing the chances of having a mutation, and hence compensating the lower mutation rate. A larger cross-over rate will also compensate a lower mutation rate as it will add more randomness. See Table 5 our experimental results regarding this.

Mutation	fitness > 0.80	fitness	fitness > 0.85	fitness	best	fitness
FIRST RUN						
0.001	27	0.808	70	0.850	197	0.917
0.071	5	0.862	5	0.862	93	0.913
0.141	9	0.800	11	0.851	97	0.937
0.210	83	0.807			153	0.827
0.280	8	0.869	8	0.869	8	0.869
0.350					145	0.798
0.420					42	0.797
SECOND RUN						
0.001	30	0.813	39	0.855	179	0.937
0.071	10	0.804	21	0.856	166	0.915
0.141	6	0.815	9	0.866	31	0.885
0.210	22	0.864	22	0.864	22	0.864
0.280	9	0.807	61	0.852	76	0.859
0.350					30	0.799
0.420					158	0.775

Table 4 – Summary of mutation dataset.

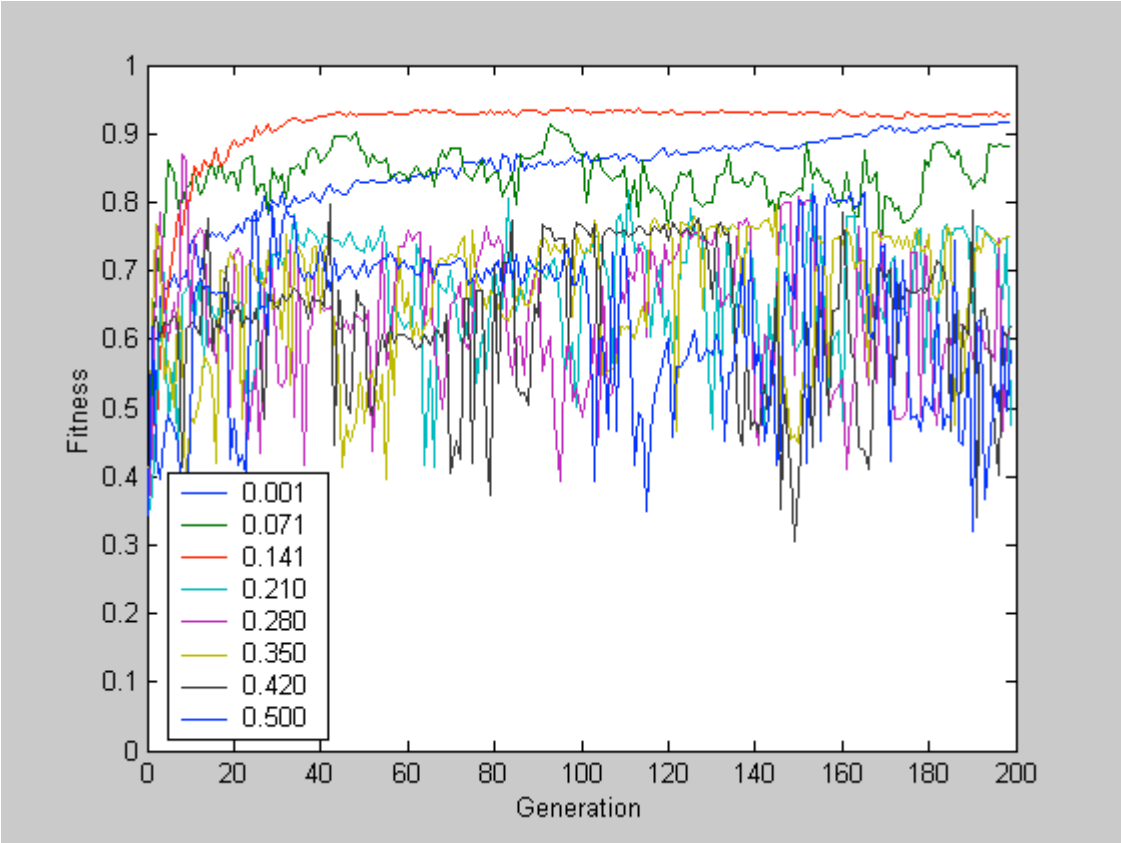


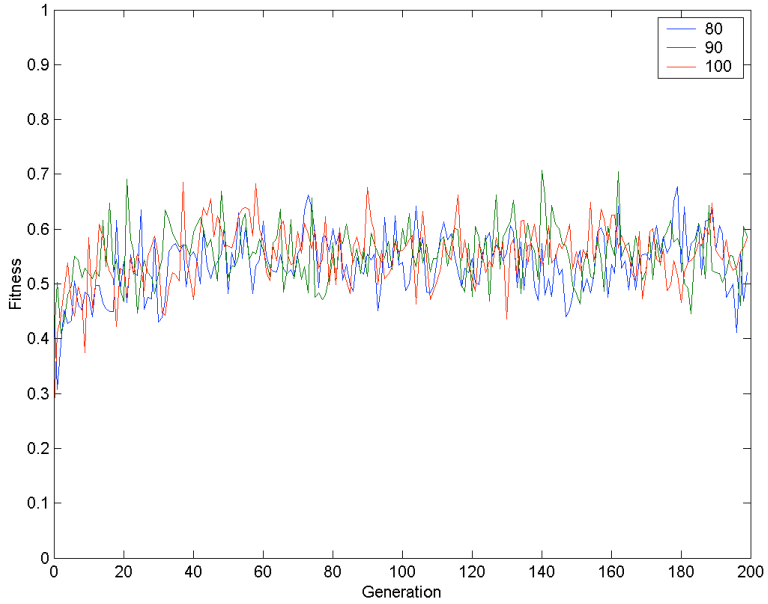
Figure 3 – Fitness evolution for different mutation values.

	fitness > 0.80	fitness > 0.85	max
mutation 0.001, population 60, cross over 0.1	10	23	0.932
mutation 0.001, population 240, cross over 0.1	3	10	0.896
mutation 0.001, population 60, cross over 0.4	2	6	0.936

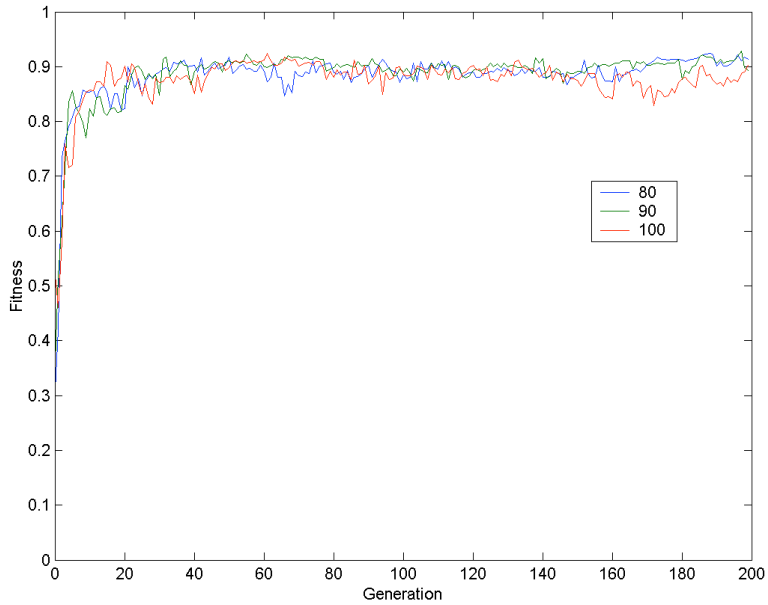
Table 5 – Compensating low mutation values.

Experiment 3 : Various preprocessing of vision (and no preprocessing)

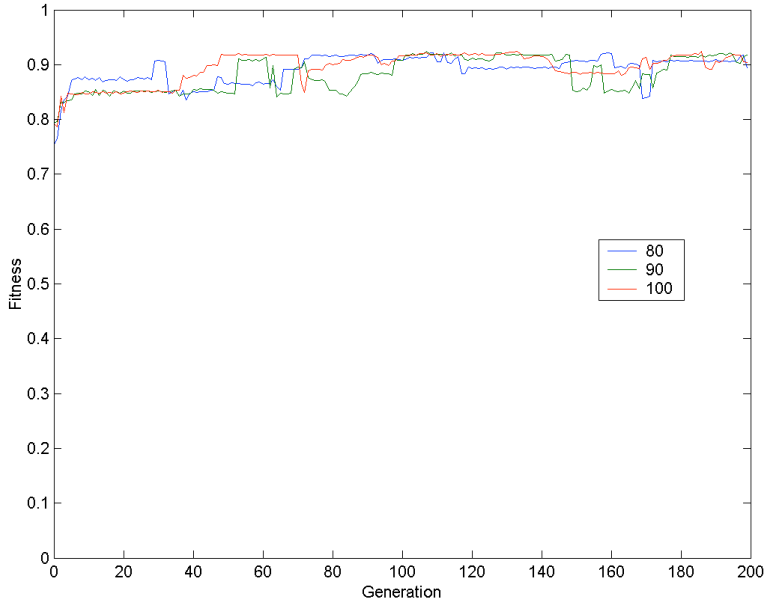
Evolution of robot with raw vision:



Evolution of robot with spatial difference:



Evolution of robot with mean spatial difference:

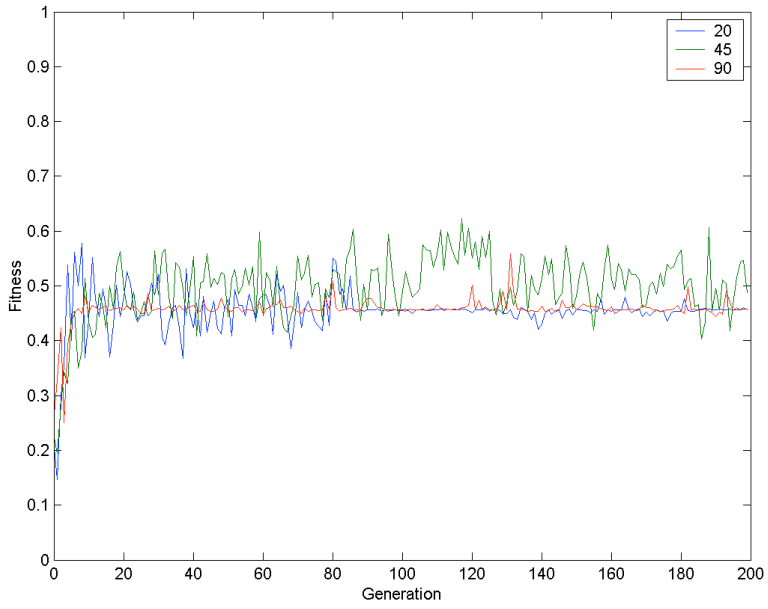


We tested the following individuals:
-Individual 140 with a FOV 90 of degrees with **raw** vision.
-Individual 197 with a FOV 90 of degrees with **spatial difference**
-Individual 107 with a FOV 90 of degrees with **mean spatial difference**.

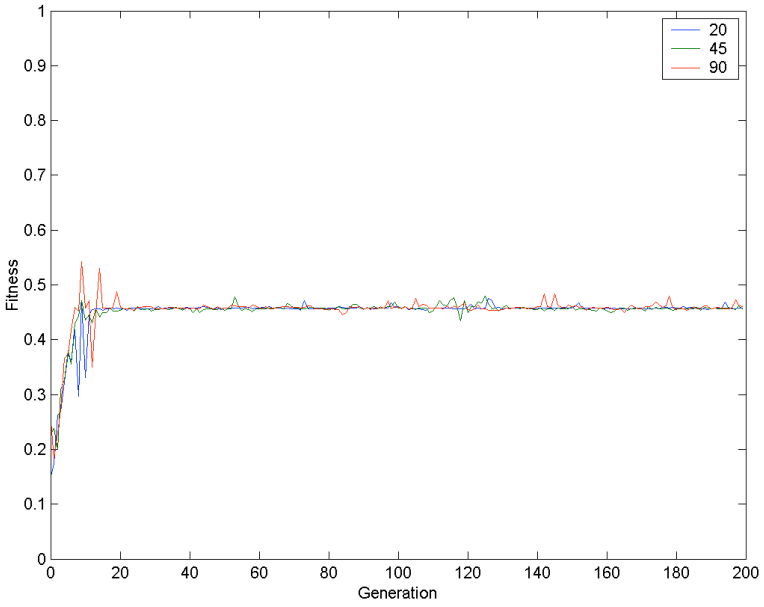
The raw vision version seems to avoid walls but performs badly (lower fitness value). The spatial difference version performs well, but it avoids the walls with small jerks. The mean spatial difference runs smoother; it really seems to follow the walls as opposed to just avoiding them.

Part II - Navigating in corridors

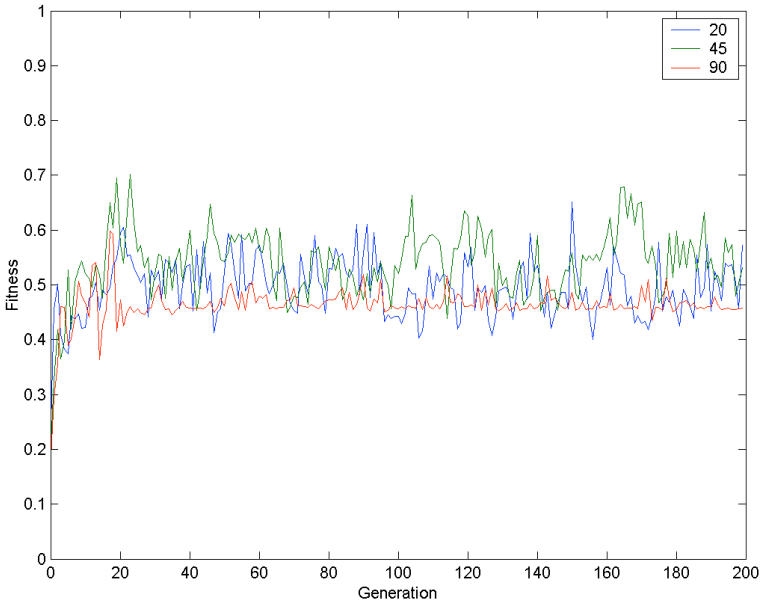
The goal of this experiment is to try different configurations in order to evolve a controller that moves in a corridor without getting blocks by the walls.



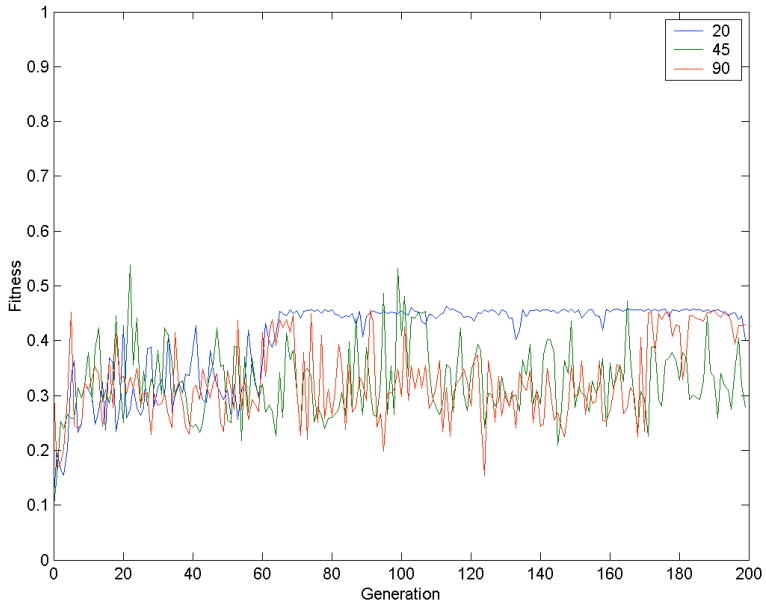
Configuration 1:
Life time = 40s
Mutation rate = 0.05
Crossover rate = 0.1
Number of epochs = 2
Spatial difference
Resolution of linear camera = 25
Both motors must be forward



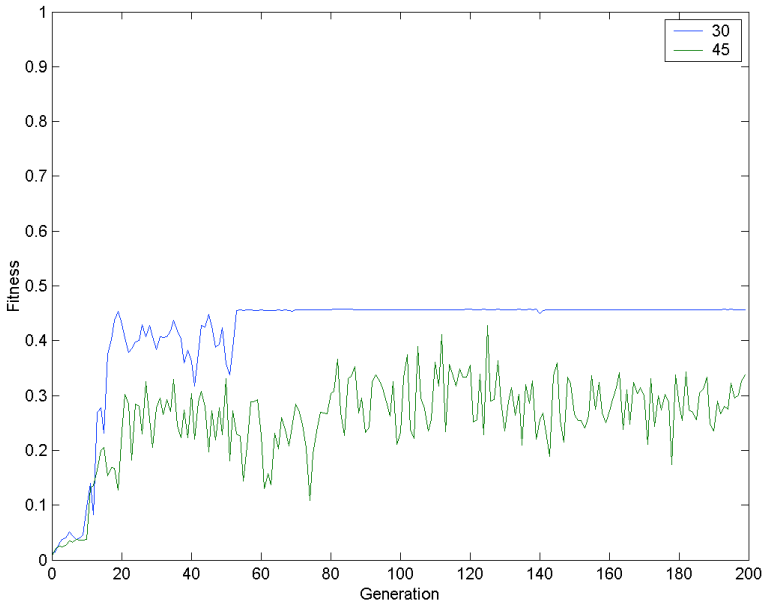
Configuration 2:
Life time = 80s
Mutation rate = 0.05
Crossover rate = 0.1
Number of epochs = 2
Spatial difference
Resolution of linear camera = 25
Both motors must be forward



Configuration 3:
Life time = 40s
Mutation rate = 0.05
Crossover rate = 0.1
Number of epochs = 2
Spatial difference
Resolution of linear camera = 13
Both motors must be forward



Configuration 4:
Life time = 40s
Mutation rate = 0.05
Crossover rate = 0.1
Number of epochs = 2
No filter on linear camera
Resolution of linear camera = 13
Both motors must be forward



Configuration 5:
Life time = 40s
Mutation rate = 0.05
Crossover rate = 0.1
Number of epochs = 20
Mean spatial difference
Resolution of linear camera = 25
Motion must be forward and trajectory straight

In order to solve the corridors problem, it is very important to use a large epoch value during the evolution phase. We also choose a small FOV, since we want our robot to move towards distant walls, and a narrow FOV will help the robot detect these walls (since further walls change from black to white in higher frequencies).

The first configurations we tried were Configurations 1 through 4, although achieving a quite a high fitness, the controller evolved would only run in small circles.

The last configuration we tried was Configuration 5, the version with a FOV of 30 degrees creates a controller with a fitness around 0.425 and has an oscillating behavior.
The version with a FOV of 45 degrees spits out a robot which is able to navigate in the corridors. It has a lower fitness than the 30 degree version, but behaves in a better way.

So to conclude this section on corridor navigation, we think the important factors are: fitness function; if we don't define it the right way, evolution can find an alternate, unpredicted solution that maximizes the fitness in a way that doesn't suite us. The FOV and epoch values are also very critical.

Conclusions

This project gave us an understanding of how evolutionary robotics work and of what are the advantages and disadvantages of working with such algorithms.

The main point that we found very frustrating is the post-simulation-data-analysis phase. The simulator generates a lot of data, and it is very difficult to understand if a specific behavior is due to evolution or is just a temporary random occurrence due to the nature of genetic algorithms. Running multiple runs with different parameters requires a lot of time, and we have no exact understanding and control over the neural network.

Applying genetic algorithms to real world problems is perhaps not a great idea. It is difficult / impossible to predict the outcome of the system when faced with a new situation. Sometimes genetic algorithms can adapt themselves to different situations (as was illustrated in the lectures and the experiment with the different FOV), but sometimes the evolved robot will just not behave the right way.

Another important point is that the simulated environment is over simplified by having textures on the walls which helps a lot the robot. It is difficult to have simple controllers work in "unmodified" environments. This can also be noticed at the biological level, where ants leave trails of pheromone (and thus modify their environment). The outcome is that the evolved controller is highly tailored to the environment it was evolved in, and taking it in another environment produces poor results.

A solution to the "texture problem" is to have more powerful and intelligent sensors, for instance a sensor that can detect the direction of a gap in the wall, at which distance a wall is etc. But this raises the problems of how to design smart sensors.

Using evolution robotics for goal oriented tasks can be difficult, as by definition a task is either completed successfully or it is not, which brings us to the problem of defining proper fitness functions. There is also a problem related to how to feed information about a task (e.g. coordinates of a destination) into the neural network and still obtain results.

Finally, evolved behavior is dependent on the sensor configuration.