1 Lab 5: Particle Swarm Optimization

This laboratory requires the following: (The development tools are installed in GR B0 01 already):

- C development tools (gcc, make, etc.)
- Webots simulation software
- Webots User Guide
- Webots Reference Manual

1.1 Office hours

Additional assistance outside the lab period (office hours) can be requested using the dis-ta@groupes.epfl.ch mailing list.

1.2 Information

In the following text you will find several exercises and questions.

- The notation $S_x$ means that the question can be solved using only additional simulation.
- The notation $Q_x$ means that the question can be answered theoretically, without any simulation; if you decide to write a report, your answers to these questions should be submitted in your report. The length of answers should be approximately two sentences unless otherwise noted.
- The notation $I_x$ means that the problem has to be solved by implementing a piece of code and performing a simulation.
- The notation $B_x$ means that the question is optional and should be answered if you have enough time at your disposal.

To prepare yourself for the graded homework sessions, we show an indicative number of points for each exercise between parentheses. The combined total number of points for this laboratory exercise is 100.

1.3 Optimization

In many instances, we want to find either the minimum or maximum value of some function. If the function in question is complex or even unknown, it is impossible to accomplish this task analytically. Furthermore, if the function parameter space is very large, a systematic search of this space is often too computationally expensive. Therefore, a number of techniques have been developed to find near-optimal solutions (i.e. local minima and maxima), such as genetic algorithms and particle swarm optimization.

1.4 The Particle Swarm

To find minima/maxima, repeated evaluations of the function in question must be done at different points. A good optimization algorithm will use the information obtained from these evaluations to choose the locations of future evaluations, and eventually to decide where the minimum/maximum is.
In Particle Swarm Optimization, a set of “particles” is initialized with a random position for each particle. Each particle is also assigned a random “velocity” in the search space. Particles are then “flown” through the search space, moving at their respective velocities.

Optimization is achieved by each particle remembering at what position they achieved the best evaluation of the function, or the particle’s “personal best”. Particles also remember the best achieving position of their “neighborhood”. The neighborhood for some particle $A$ is a group of particles to which $A$ belongs. This group can be topological (whatever particles are closest) or fixed throughout the algorithm. It can consist of either a subset of the particle swarm (local neighborhood), or the entire swarm (global neighborhood). The “neighborhood best” of a neighborhood is the position that yielded the best evaluation by any particle in the neighborhood.

At each iteration of the algorithm, the velocity of each particle is updated, using a randomized attraction to both the particle best and the neighborhood best. This allows particles to move towards areas of the search space that have yielded good results, and in doing so, discover nearby better results. In this way, the particles will eventually converge on possibly global optima. The idea for this type of behavior took inspiration from the ways birds act while flying in a flock.

2 Lab: Using PSO

2.1 PSO on the Sphere Function

The Sphere function is defined as follows:

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2$$

We want to find the set of $x_i$ that give the value of $f$ closest to 0 (given by $x_i = 0$ for all $i$). The values for $x_i$ are initially randomly distributed between -10.24 and 10.24. This function is often used as a simple test for optimization functions; although it may seem trivial to a human observer, it can be difficult for an optimization algorithm to figure out that the best solution is all zeroes. We will use PSO with a swarm size of 30 and the neighborhood of a particle given by the three nearest particles on each side (e.g. particle 8 has neighborhood \{5, 6, 7, 8, 9, 10, 11\}) with particle 29 being next to particle 0. We will optimize the Sphere function with $n = 10$, and, therefore, ten components in the vector representing each particle. In other words, a particle is a set of 10 numbers, and the closer to 0 all the numbers become, the better the performance.

Download lab_hwk5.tar.gz from Moodle and unarchive it:

```bash
$ tar xvfz lab5.tar.gz
```

This will create two directories: `pso` and `webots`. Change directories into `pso` and compile the program:

```bash
$ cd pso
$ make
```
You can execute the program with

```
$ ./main <number of iterations>
```

S: Run the program with 10 iterations, 100 iterations, 1000 iterations, and 10000 iterations. At the end of each trial, both the achieved value and the found solution for 10 runs will be printed along with the average value. Lower values mean better the solutions.

Q1 (10): What do you observe? How many iterations do you think will be necessary to get the exact answer? Why?

S: The variable `vmax` in `main.c` controls the maximum velocity that particles can achieve in the simulation. Using 100 iterations, try varying `vmax` from 0.5 to 4.0. You can execute the program with

```
$ ./main <number of iterations> <vmax>
```

Q2 (10): What do you observe?

S: Try varying `vmax` from 0.05 to 0.2.

Q3 (10): What happens now? Why?

It can be very inconvenient to have to tune `vmax` to give the optimal performance for a simulation. A feature that was developed for PSO soon after its invention was an “inertia” coefficient. This was a coefficient typically less than 1 which is multiplied with the velocity at each iteration, in order to naturally slow particles down without imposing a hard velocity threshold.

S: Run the program again with the following command, setting `vmax` to 4.0 and the `inertia` coefficient to 0.6:

```
./main <number of iterations> 4.0 0.6
```

Q4 (10): Try increasing the number of iterations. What do you observe now?

2.2 PSO for Evaluative Adaptation

We’ve established that PSO can do a good job of optimizing simple mathematical functions. The next step is to test it in a more demanding environment, in particular in the presence of noisy functions.

In “Evolution of Homing Navigation in a Real Mobile Robot” (Floreano and Mondada, 1996), a genetic algorithm was used to train a robot to move through a maze and avoid obstacles. We will use Webots to shape robotic controllers for the mobile robots, comparing the performance of GA and PSO. For the experiments in this lab, we use the e-puck robot instead of the Khepera that was used in the paper. The behavior we wish to achieve on the robots is obstacle avoidance, the same as in Floreano and Mondada’s paper.
We have implemented both the GA and the PSO algorithm in Webots for you. **Before loading the Webots worlds, go into the webots directory, and compile all controllers:**

```bash
$ cd webots/controllers
$ cd ga_obs_sup           # or pso_obs_sup, obs_con
$ make
```

The worlds we are using here are simple walled arenas with no obstacles and twenty e-puck robots. We use a two neuron, single-layer, neural networks to control each robot, with proximity sensors and the relative position center of mass as inputs and the motor speed as outputs. There are recursive and lateral connections from the outputs of both neurons back into each other. The controller we adapt is the set of weights for these neurons (8 proximity sensors + 2 recursive/lateral connections + 1 threshold = 11 weights for each neuron => 22 weights total). The fitness of a controller is measured with the equation:

\[
F = V(1 - \sqrt{\Delta v})(1 - i), \quad V, \Delta v, i \in [0,1]
\]

where \( V \) is a measure of the average rotation speed of the two wheels over the trial, \( \Delta v \) is a measure of the average difference in wheel speed of the two wheels over the trial, and \( i \) is the average activation value of the proximity sensor with the highest activity. Each of these terms in the fitness function encourage the robot to go fast, go straight, and not stay close to walls, respectively. The default time for a single evaluation of fitness is \( \sim 30 \) s.

The genetic algorithm we're using is similar to that used in 1996 by Floreano and Mondada. It selects the best performing half of the population as potential parents. Out of these, parents are selected using a Roulette Wheel scheme. Crossover occurs with probability 0.2 and mutation occurs with probability 0.15 to create children to replenish the missing half of the population.

### 2.2.1 Fast Heterogeneous Adaptation

Using PSO or GA on a single robot, it can take a long time to automatically design high-performing controllers. This is because the robot will have to test every member of the population/swarm one after another for each iteration of the algorithm. For example, doing 100 iterations of PSO with 20 particles with each evaluation taking 1 minute means we need 2000 minutes or over 33 hours to complete!

One way to speed up this process is to use multiple robots using a public individual heterogeneous adaptation strategy. In heterogeneous adaptation, each robot is using a different controller than the others. By using multiple robots, we can distribute the particles among the robots and evaluate their performance in parallel, thus saving a lot of time. Going back to our previous example, if we have 20 robots each evaluating a different particle, it will only take 1 minute to evaluate 20 particles, and the total adaptation time goes down to about 1.7 hours.

We will now test shaping of obstacle avoidance behavior using public individual heterogeneous adaptation with GA and PSO. In the follow questions, you will be asked to run simulations that will take about 3 hours of simulated time, and several
minutes of real time to complete. In order to save time, you may want to look ahead to the next question to see if it is something which you can work on while you wait.

**S:** Load the *ga_obs* world. Run the world to simulate the evolution. You will need to use fast mode to simulate quickly enough, but you may want to observe how the robots are behaving occasionally (pay attention to whether they are near the beginning or end of an evolutionary run). The simulation will do 10 separate evolutionary runs and print the final fitness for each, as well as the average fitness at the end. When the evolution is finished, observe the performance of the robots (they will be running the best found controller).

**Q₅ (10):** What kind of fitness values do you get?

**S:** Now load the *pso_obs* world, and run the simulations again.

**Q₆ (10):** How does the performance compare?

We now switch to using the modified version of our genetic algorithm specifically designed for noisy fitness function evaluation (see Zhang et al., 2008). At each iteration, the performance of the potential parent set is reevaluated, and the new fitness value is combined with previous values to get a combined estimate using some sort of aggregation function (e.g., minimum, average).

**Q₇ (10):** Why might this modification help to improve performance?

**S:** In the *controllers/ga_obs_sup/ga.c* file, set the NOISY definition from 0 to 1. This enables the modification. Run the simulations again.

**Q₈ (10):** Record the final fitness values. How do the new values compare?

The modification for GA for handling noisy fitness function evaluations (originally proposed by Zhang et al. in 2003) was adapted for PSO in “Particle Swarm Optimization for Unsupervised Robotic Learning” (Pugh et al., 2005). In this algorithm, at each iteration, the previous best positions of the particles are re-evaluated, and the new fitness value combined with previous ones.

**S:** In the *controllers/pso_obs_sup/pso.c* file, change NOISY from 0 to 1. This enables the modification. Run the simulations again.

**Q₉ (10):** Record the final fitness values. How do the new fitness values compare?

**Q₁₀ (10):** In both modifications, the number of iterations run by the algorithm is divided by two. Why does this make sense?
3 References

