Evorobot* User Manual

Stefano Nolfi & Onofrio Gigliotta
Institute of Cognitive Science and Technologies, National Research Council (CNR)
Via S. Martino della Battaglia, 44, 00185, Roma, Italy
stefano.nolfi@istc.cnr.it
http://laral.istc.cnr.it/nolfi/

Index

Index 1
1. Introduction 2
  1.1 Evorobot* features 2
  1.2 Using Evorobot* 3
2. Running the program 4
  2.1 The graphic interface 4
3. Menu and program variables setting 5
  3.1 The File menu 5
  3.2 The Run menu 6
  3.3 The Display menu and the SET command: how to display and modify parameters 7
  3.4 The Help menu 11
  3.5 Running robots in hardware 11
4. Compiling the software 12
5. Overview of the source code and tips on how to modify the program 13
  5.1 Source files 13
  5.2 Program parameters 13
  5.3 Fitness functions 14
  5.4 Sensors and Motors 15
  5.5 The evolutionary algorithm 16
  5.6 Important variables and data structure 16
6. References 17

1. Introduction

This document provide the user manual for the Evorobot* software (http://laral.istc.cnr.it/evorobotstar/), which has been developed at the Laboratory of Artificial Life and Robotics, ISTC-CNR (http://laral.istc.cnr.it) by Stefano Nolfi and Onofrio Gigliotta. Evorobot* will allow you to run experiments on the evolution of collective behavior and communication (for more information about evolutionary robotics see Nolfi & Floreano, 2000). The tool is based on the e-puck robotic platform which has been developed at the Ecole Politechnique Federale de Lausanne (Mondada & Bonani, 2007). Evorobot* allows to evolve neural controllers for this type of robotic platform both in simulation and in hardware.

Evorobot* is an extension of the evorobot software available from http://laral.istc.cnr.it/evorobot/simulator.html. New features include: the possibility to run collective robotics experiments, the possibility to define any type of neural architecture, an enhanced graphic interface, the possibility to be compiled on different operating systems (Windows and Linux).

The Evorobot* software is copyrighted (or "copylefted", to use the term introduced by the Free Software Foundation) under a GNU General Public License. This means that it may be used, copied, modified, or redistributed for free. However, any redistribution (of the original or modified code) should adhere to the General Public Licence terms, and copies should acknowledge both the original author and be subject to the terms of the GNU General Public License. The Evorobot* package (which include the source files, the user manual, a set of examples, and this tutorial) can be freely downloaded from http://laral.istc.cnr.it/evorobotstar/). It is written in C and C++ and can be compiled under Windows and Linux operating systems.

1.1 Evorobot features*

Evorobot* allows you to evolve a robot or of a group of robots for the ability accomplish a certain task in a given environment. The robots have a circular body shape (i.e. corresponding to the characteristics of the e-puck robotic platform) and can have different type of sensors (e.g. infrared, ambient light, ground, vision, signal) and of actuators (e.g. wheel motors, active led lights, signal emitters). The robots are provided with neural controllers including a certain number of sensory neurons (which encode the state of the corresponding sensors), internal neurons, and motor neurons (which encode the state of the corresponding actuators). The environment consists of one or more arenas surrounded by walls which can include objects (e.g. walls or cylinders with different size and color), light bulbs, and landmarks (e.g. floor areas painted in a given color). The task to be accomplished is specified in a fitness function which determines how the performance of the evolving robots will be evaluated.

More precisely, the Evorobot* software include six integrated tools (i.e. program sub-parts playing specific functionalities): (i) an evolutionary algorithm, (ii) a neural network simulator, (iii) a simulator of the robots, of the environment, and of their interaction, (iv) a graphic interface as well as commands for saving and analyzing data, (v) a tool which allow the user to test and/or evolve robots' controller in simulation and in hardware, (vi) an evorobot* firmware to be loaded on each robot which allow each robot to behave on the basis of the neural controllers evolved on a PC which communicate with the robots through a wireless bluetooth connection.

The **evolutionary algorithm** tool allows the user to create an initial generation of genotypes, to evaluate individuals' performance on the basis of a fitness function, and to generate successive generations. Each selected individual is allowed to produce a certain number of offspring which consists of copies of the genotype of the reproducing individual with the addition of variations (i.e. mutations). The user can specify the parameters of the evolutionary process which include the number or reproducing individuals, the number of offspring, the mutation rate, the use of elitism. The user can also specify the number of individual robots which are situated in environment and whether the group of robots is homogeneous or not (i.e. whether the individuals have the same genetic characteristics or not).

The **neural network simulator** tool allows the user to specify the characteristics of the robots neural controller (i.e. the architecture of the neural controller and the number and the type of the neurons) and to compute the activation state of the neurons. The program allows the usage of standard logistic neurons, leaky neurons with genetically encoded time constant parameters, and biased or unbiased neurons. Moreover, the program allows the users to easily specify any possible type of neural architecture. Standard architectures (e.g. feed-forward or recurrent neural controller

can be specified by setting few parameters). Irregular or unconventional architectures can be specified by indicating connectivity blocks formed by a group of neurons receiving connections from another group of neurons. The tool includes a graphic interface which allows the user to easily define the architecture of the robots' neural controller as well as display the architecture and the parameters of a specific individual controller.

The **robot/environmental simulator** tool allows the user to define the characteristics of the robots and of the environment (i.e. the robots' sensors and actuators, the size and the objects contained in the environment), to compute (in simulation) the state of the robots' sensors (on the basis of the current position and orientation of the robots in the environment) and how the position and the orientation of the robots or the characteristics of the environment vary in simulation as a result of the robots' actions.

The **graphic interface** tool allows the user to: (i) run commands from the menu bar, (ii) modify the parameters of the program while the program is running by using a simple command line instructions, (iii) modify the characteristics of the environment, of the positions of the robots, or the characteristics of the neural controllers graphically, (iv) visualize graphically the environment, the robots' behavior, the architecture of the neural controller, the current state of the neurons, the fitness value, etc.

The **evorobot* firmware** is a software which allows to run evolutionary experiments on the robots or to test neural controller evolved in simulation on the real robots. To use the real robots, the user should load the firmware on each robot and establish a bluetooth connections with the robots. A part from that, the user can use the same graphic interface and the same command for running experiments in simulation or in hardware.

The six tools described above are tightly integrated to maximize usability and to reduce the risk of introducing errors while extending or modfying the source code. In particular, the program automatically determines the length of the individuals' genotype on the basis of the characteristics of the robots' neural controllers. Moreover, the program automatically creates sensory and motor neurons on the basis of the sensors and motors selected by the user. Finally, the graphic interface of the program automatically displays all crucial variables over time thus allowing to the user to quickly analyze the obtained results and eventually easily identify problems or bugs in introduced in the code.

*1.2 Using Evorobot**

The use of the tool typically involves three phases in which the user: (i) set-up an experiment, (ii) run the evolutionary process, and (iii) analyze the obtained results. The program allows the user to replicate some of the experiments described in this book, but also to run its own brand new experiments. In some case, new experiments can be set up simply by varying the program parameters. In other case (e.g. when the user wants to use a new fitness function) the user might need to extend the source code of the program and recompiling it before running the new experiment.

Setting up an experiment. During this phase the user can define the characteristics of the robots and of the environment which are fixed by specifying the sensory-motor system of the robot, the architecture of the neural controller and the type of neurons, the characteristic of the environment/s, the characteristics of the robots' lifetime (e.g. number or robots concurrently situated in the environment, number of trials, number of step for each trial, fitness function type), and the characteristics of the evolutionary process (e.g. population size, mutation rate, number of generations, etc.). These parameters can be defined by editing the configuration text files before executing the program or by modifying the parameters through the graphic interface and by saving the modified parameters in the configuration files. In the latter case, once the parameters has been set, the users should exit from the program, and re-execute the program again in order to allow the program to appropriately allocate memory on the basis of the parameters which have been specified. The parameters which do not affect the size of the genome or the size of the architecture of the neural controller (i.e. do not affect memory allocation) instead, can be modified at any time without the need to quit and restart the program.

Running an experiment. To run an evolutionary experiment the user simply has to issue the corresponding command from the menu bar. During an evolutionary process the program automatically display statistical data about the fitness and automatically save the genome of evolving individuals.

Analyzing obtained results. The program allows the user to test the behavior of a single or of a group of robots in simulation and in hardware and to run evolutionary experiments (typically in simulation). To test pre-evolved individual the users should first load the corresponding genotype from

a file. The program also allows the user to easily analyze evolved robots at the level of the robots' behavior but also at the level of the robots' neural controller. In particular the program allows the user to analyze the free parameters of evolved neural controllers, to lesion neurons, to visualize the state of the neurons while the robots interact with their environment etc.

Extending the Evorobot* source code. The user manual of the program provide an overview of the program source code as well as indications for the most common extensions required which typically consist in the need to implement a new fitness function, or the need to implement a new type of sensor or actuator. Evorobot* has been conceived so to simplify these operation as much as possible.

2. Running the program

When you run the program it automatically loads from the current directory a series of files and more specifically: (a) the file "evorobot.cf" that contains most of the parameters; (b) the file evorobot.net, if present, that contains a description the architecture of robots' neural controllers; (c) the file/s world?.env which contains a description of the environment in which the robots are situated (where ? indicate the id of the world files). The number of world files to be loaded is indicated in a parameter included in the evorobot.cf file; (d) the sample files (wall.sam, round.sam, sround.sam, light.sam), placed in /bin/sample_files, that contain samples of the infrared and light sensors of a particular robot. All these files are text file and can be visualized with a text editor. However, it is more convenient to visualize and modify the content of these files through the graphic interface (see below).

Once the program started, the users can: (i) modify the parameters of the experiment so to set up a variation of an existing experiment or a new experiment, (2) run an evolutionary process, (3) test evolved robots. Robots can be tested in simulation (default mode) or in hardware (see section 3.5)

In the next section we describe the graphic interface. In the section 3 we describe the commands that can be executed through the menu bar, the command line window, and the tool bar.

2.1 The graphic interface

The program consists of a standard graphic application with a menu bar, a tool bar, and a status bar. The central part of the graphic windows is used to display graphic and textual data (the behavior of the robots, the architecture of the neural controllers, statistical data, the current value of the parameters, etc. The command line window at the bottom of the graphic application is used to input textual commands (e.g. "set lifetime ntrials 8") that allow modifying the parameters of the program.

When you ran the program it automatically displays the environment and the robots and the architecture of the robots' neural controller). You can re-visualize this data by clicking on the graphic window (unless the commands Run->Evolution or Run->Test are currently running). You can also modify the positions of the robots, the objects located in the environment, and the neurons by using the graphic display. Finally, you can display the free parameter of the current individual (i.e. the value of the time constants and biases of neurons, or the connection weights).

To modify the position of a robot: (1) select the robot by clicking on it with the mouse (selected robots are indicated with a gray circle), (2) drag and drop the robot. To modify the orientation of one robot: (1) select the robot by clicking on it with the mouse, (2) click on the ***rotate selected robot*** icon on the tool bar to rotate the robot 2 degree at a time or use the CTRL-R shortcut. Notice that after the positions of the robots have been manually modified, the Run->Test command preserves the current positions of the robots. This means that you can test the behavior produced by the robot when there are located in specific position of the environment. To temporary disable a robot, you should drag and drop it out of the environmental area.

To modify the position of one object of the environment you drag and drop the object with the mouse (for wall objects you can drag and drop one of the two end-points at a time). To select an object, click on the center of it. To modify the position of a neuron on the graphic display, drag and drop the neuron with the mouse. You can use the parameter **display.grid** (see below how to set parameters) to constraint the final position on a grid. To delete an environmental object, drag it at the very bottom of the graphic display.

You can also modify the neurons to be displayed during the Run->Test individual command by selecting one neuron or a block of neuron (see above) and by clicking on the ***select/deselect neurons to be displayed as graph*** icon on the tool bar.

You can visualize the biases or the time constant of the neurons by clicking on the ***display/un-display neurons delta*** and ***display/un-display neurons biases*** icons of the tool bar. You can

display the incoming and outcoming weights of a neuron by selecting the neuron (i.e. by clicking on the selected neuron with the mouse) and by clicking on the ***display/un-display weights*** icon of the tool bar.

You can "lesion" neuron (i.e. freeze their activation state to 0.0) by selecting one neuron or a block of neurons and by clicking on the ***lesion/restore neurons*** icon of the tool bar.

Finally, you can increase or reduce the size of the environment by dragging the green circle placed on the top-right part of the environment left or right.

In the next sections we describe all the commands, parameters, and files meaning in details. In doing that we will indicate: (i) the command which can be issued by the menu bar with **menu->command** (where menu and command indicates the name of the corresponding menu and command), (ii) the command which can be issued by the command line window with **SET X Y** (where X and Y are parameters of the command), (iii) the variables which can be modified with **folder.variable** (where the folder and variable indicate the class of the corresponding variable and the name of the variable itself).

3. Menu and program variables setting

3.1 The File menu

The File menu contains commands that allow you to load and save data from files.

3.1.1 The **File->Open** command

The **File->Open** command can be used to load:

(a) Configuration files (.cf) containing program parameters (for more details see the Display menu).

(b) Genome files (.gen) containing the genotype of one or more individuals. These files are generated by the program through the command Run->Evolution and might contains the genome of an entire population (as in the case of the file "G99P0S1.gen" that contains the genome of the generation 99 of the replication 1) or the genome of the best individuals (as in the case of the file B1P0S3.gen that contains the genome of the best individual of each generation for the replication n. 3). When you load the best individuals you should verify to have allocated enough space for the genome by eventually setting the parameter evolution.additional_ind. If the file contains more individuals than the maximum allowed, the program will load only part of them (the number of loaded individual is indicated in the status bar). Evorobot assign the filenames GxPySz.gen (where x is the number of generation, y is the number of the population, and z is the seed of the replication) to files that contains the genome of a population and the filename B1PySy.gen (where y is the number of the population, and z is the seed of the replication) to the files that contain the best individuals of each generation).

(c) Fitness files (e.g statS4.fit) that contains the average fitness of the population throughout generations and the fitness of the best individual of each generation. Fitness files are automatically generated by the program during the execution of the Run->Evolution or Run->Test_All commands. The syntax of the filename is statSx.fit and masterSx.fit (where x is the seed of the replication) in the former and latter case, respectively.

(d) Environmental files (e.g. world1.env) that contain a description of the characteristics of the environment.

(e) Neural Architecture files (e.g. evorobot.net) that contains a description of the architecture of the neural controllers.

(f) Phenotype files (e.g. test.phe) that contain pre-elaborated values of free parameters (the parameters that are still left free are indicated with the value 999.0). This command is useful to run incremental evolutionary process (more details in the description of the command run->evolution).

3.1.2 The **File->Save** command can be used to save:

(a) The evolution, individual, lifetime, and display parameters in a file .cf. The file evorobot.cf is loaded at runtime. Therefore, parameters to be loaded automatically should be saved in this file.

(b) The characteristics of the environment in a file world1.env or in a set of file worldX.env where X is the i.d. number of the environment. The number of environmental file (range [1-5]) is defined in the parameter **lifetime.nworlds**. The root of the filename is constrained. That is, you cannot save data on a different file (e.g. on a file "prova1.env" for example).
(c) The characteristic of the neural architecture in a file .net. The file evorobot.net is loaded at runtime. Therefore the network structure to be loaded automatically should be saved in this file.
(d) The free parameters or the current neural controller encoded as floating point value ranging from –**individual.wrange** to +**individual.wrange** of the current individual (**display.dindividual**). The command used with the filename empty.phe allow to save a .phe file that include all don't care (i.e. 999.0) values. This command is useful to run incremental evolutionary process (more details in the description of the command run->evolution).

3.1.5 The **File->Exit** command can be used to exit the program.

3.2 The Run menu

The Run menu includes command that allow to run evolutionary experiments and to test obtained results.

3.2.1 The **Run->Evolution** command

This command runs the evolutionary process or a set of replications of the evolutionary process starting from different randomly generated initial populations. The number of replications that will be run is defined in the parameter **evolution.nreplications**. The first replication will be initialized with the seed defined in the parameter **evolution.seed**. The next replications will be initialized by using the succeeding number as seeds for the random number generators. The seed of the replication is also used to differentiate the name of the files of the different replications. If some of the replications have previously ran or if part of the generations of a given replications has been already been generated, the program will start from the first new replication or will restart from the last generation previously generated. If you want to avoid that, you should first remove the corroding .gen and .fit files contained in the current directory.

While the evolutionary process is running, the program displays in the status bar the id (seed) of the current replication, the current generation, the number of last tested individual, and its fitness. Moreover, as soon as the first generation has been tested, the program starts to display two graphs with the best and the average fitness of the population for each generation.

You can terminate the evolutionary process by clicking the right bottom of the mouse on the graphic window or by executing the command **Run->Stop**.

The **Run->Evolution** command will automatically generate a set of files containing the genotype of the first and last population of each replication and the genotype of the best individuals of each generation (by using the following file names: GxP0Sz.gen and BqP0Sz.gen (where x is the generation number and z is the seed of the replication, and q is the 1 for the best individual, 2 for the second best individual etc.). The number of best individuals that will be saved is defined in the parameter **evolution.savebest**. The command will also generate a statSx.fit file (where x is the seed of the replication) that contains the best and average performance for each generation and a stat.fit file that contains the average results of all replications. Finally, if the **evolution.saveifit** parameter is set to 1, the command will create a fitP0Sx.txt file (where x is the seed of the replication) that contains the fitness gathered by each individual of each generation.

You can use .phe files to specify parameters that you want to fix to certain value and you do not want to subject to the evolutionary process. This can useful, for example, to run incremental evolutionary processes. For example, in order to evolve a controller able to display an obstacle-avoidance behavior and a phototaxis behavior in an environment with obstacles you might evolve the connection weights which connect the sensory neurons to the motors neurons first for the ability to avoid obstacles in an environment without light bulbs, and then the connection weights connecting the light sensors to the motors for the ability to perform a phototaxis behavior. For doing that you might proceed as following: (1) you first evolve the controller from scratch for solving task A in an environment without light-bulbs; (2) you load the best individual and save its phenotype parameters in a file (e.g. preevolv-a.phe); (3) you edit the file preevolv-a.phe by replacing the parameters that you

want to leave free with values 999.0 (i.e. the parameter corresponding to the connection weights between the light-sensors and the motors); (4) you load the file preevolv-a.phe and evolve the controllers for the capacity of solving task B in an environment with a light-bulb.

3.2.2 The **Run->Test** command

This command tests the current individual of the current population. The current individual is set in the parameter **display.dindividual**. To display the individual x of generation y of replication z, you should: (a) load the file GyP0Sz.gen, (b) set the parameter display.dindividual to x, (c) run the command. To display the best individual of generation x of replication y, you should: (a) load the file B1P0Sy.gen, (b) set the parameter display.dindividual to x, (c) run the command. To display the best individual of the last generation of replication y, you do not need to manually set the parameter display.individual since it is automatically set when you load the file containing the best individuals to the id of the last loaded individual.

During the execution of the command (in the simulation mode) the program will display the robot and the environment on the left part of the graphic window and the state of neurons on the right part of the graphic window. The current lifecycle, fitness, and total fitness is displayed in the status bar (notice that the value of total fitness might be normalized at the end of the test, the status bar simply display the total fitness accumulated up to the current lifecycle).

You can stop the command by clicking the right button of the mouse on the graphic window or by running the **Run->Stop** command.

3.2.3 The **Run->Test_Best** command

This command is identical the **Run->Test_Ind.** command. The only difference is that the **display.dindividual** parameter is automatically set by selecting the best individual. To test the best individual of a replication you should: (a) load the file B1P0Sx.gen (where x is the seed of the replication) (b) load the file StatSx.fit or the file MasterSx.fit (where x is the seed of the replication), (c) run the command.

3.2.4 The **Run->All** command

This command tests the best individuals of all generations for a certain number of replications. The command does not show the behavior of individuals graphically but compute individual fitness and save the obtained data in files MasterSx.fit (where x is the seed of the replication). The command also shows these data graphically on the graphic window. The command automatically loads the genome of the best individuals from file B1P0Sx.gen (where x is the seed of the replication). The number of replication tested and the seed of the initial replication are determined by the parameter **evolution.nreplications** and **evolution.seed**, respectively.

3.2.5 The **Run->Create_Lineage** command

This command generate the lineage of the ancestors of the best individuals of the last generation of a previously ran experiment (for all replications of the experiments). In order to do that, you have to set the parameter **evolution.savenbest** parameter equal to **evolution.nreproducing** when you ran the evolutionary experiments (i.e. you have to save the genotype of all individuals which are allowed to reproduce). The genotype of the lineage are saved in files Lineage?.gen where ? correspond to the seed of the corresponding replication.

3.2.6 The **Run->Stop** command

Terminate the execution of the **Run->Evolution**, **Run->Test**, or **Run->Test_all** commands. The same command can be executed by right-clicking on the graphic window.

3.3 The Display menu and the SET command: how to display and modify parameters

This menu contains commands that display the parameters of the program. These parameters are loaded from the files evorobot.cf, evorobot.net and worldx.env (where x is the id of the environmental file). In this section we also describe the parameters that can be modified through the SET command issued from the command line window located at the bottom of the screen and through the graphic interface.

3.3.1 The **Display->Evolution_par** command

This command displays on the graphic window the parameters that regulate the evolutionary process. The command also displays a brief description of the meaning of each parameter (if the parameter **display.verbose** is set to 2). For parameters that can assume restricted values, these values are usually indicated at the beginning of the parameter description, in square brackets. The command can also be executed by writing the command "SET evolution" followed by a carriage return in the command line.

To modify these parameter you should write the command "SET evolution <x> <value>" or "SET evo <x> <value>" followed by a carriage return where <x> is the name of the parameter and <value> is the value to be set. For example to modify the number of generation to 300 you should use the command "SET evolution ngenerations 300". After the execution of the command the program will automatically display the updated parameters.

The current state of the parameters can be saved on a file .cf with the command **File->Save**. Please consider that the parameters saved in the file evorobot.cf are automatically loaded at runtime.

3.3.2 The **Display->Individual_par** command

This command displays on the graphic window the parameters that determine the characteristic of individuals (sensors, motors, internal neurons, etc.). The command also displays a brief description of the meaning of each parameter (if the parameter **display.verbose** is set to 2). For parameters that can assume restricted values, these values are usually indicated at the beginning of the parameter description, in square brackets. The command can also be executed by writing the command "SET evolution" followed by a carriage return in the command line.

To modify these parameter you should write the command "SET individual <x> <value>" or "SET ind <x> <value>" followed by a carriage return where <x> is the name of the parameter and <value> is the value to be set. For example to provide individuals with 8 light sensors you should use the command "SET individual lightsensors 8". After the execution of the command the program will automatically display the updated parameters.

The current state of the parameters can be saved on a file .cf with the command **File->Save**. Please consider that the parameters saved in the file evorobot.cf are automatically loaded at runtime.

3.3.3 The **Display->Lifetime_par** command

This command displays on the graphic window the parameters that determine the parameters that determine the lifetime of individuals. The command also displays a brief description of the meaning of each parameter (if the parameter **display.verbose** is set to 2). For parameters that can assume restricted values, these values are usually indicated at the beginning of the parameter description, in square brackets. The command can also be executed by writing the command "SET lifetime" followed by a carriage return in the command line.

To modify these parameter you should write the command "SET lifetime <x> <value>" or "SET lif <x> <value>" followed by a carriage return where <x> is the name of the parameter and <value> is the value to be set. For example to set the number of trials to 50 you should use the command "SET lifetime ntrials 500". After the execution of the command the program will automatically display the updated parameters.

The current state of the parameters can be saved on a file .cf with the command **File->Save**. Please consider that the parameters saved in the file evorobot.cf are automatically loaded at runtime.

3.3.4 The **Display->Display_par** command

This command displays on the graphic window the parameters that determine what is displayed on the graphic window. The command also displays a brief description of the meaning of each parameter (if the parameter **display.verbose** is set to 2). For parameters that can assume restricted values, these values are usually indicated at the beginning of the parameter description, in square brackets.

The command can also be executed by writing the command “SET display” followed by a carriage return in the command line.

To modify these parameter you should write the command “SET display <x> <value>” or “SET dis <x> <value>” followed by a carriage return where <x> is the name of the parameter and <value> is the value to be set. For example to not show the trace of the robots while they move in the environment you should use the command “SET display drawtrace 0”. After the execution of the command the program will automatically display the updated parameters.

To determine the neurons whose activation state should be displayed through time while robots are tested through the command Run->Test or Run->Test_best, you can use the mouse and the graphic interface. The neurons that will nor or will be displayed have their label displayed in black and red, respectively, when they network is displayed in right part of the graphic window. This display property can be changed by selecting one block of neurons and by using the ***select/deselect neuron to be displayed as a graph*** icons on the tool bar. To select a block of neuron you should (1) click in an empty place of the graphic window (if you want to eliminate the current selection), (2) click on the first neuron of the block (the neuron become black), (3) click on the last neuron of the block (all the neurons of the block become black). Neurons are ordered from sensors, to internal, to motors are a graphically displayed from left to right and from the lower to the higher layer.

The current state of the parameters can be saved on a file .cf with the command **File->Save**. Please consider that the parameters saved in the file evorobot.cf are automatically loaded at runtime.

3.3.4 The **Display->Environ_par** command

This command displays the characteristic of the current environment. The id of the current environment and the number of environments are defined in the parameters **lifetime.nworld** and **environment.cworld**, respectively. The command can also be executed by writing the command “SET environment” or “SET env” followed by a carriage return in the command line.

To modify these parameter you should write the command “SET environment <object> <id> <values>” or “SET env <object> <id> <value>” followed by a carriage return where <object> is the type of the object, <id> is the id number of the objects, and <values> are the parameters of the objects. You can: (1) delete an existing object by issuing the command “set env <object> <id>”; (2) modify an existing object by issuing the command “set env <object> <id> <new_values>” where new_values are the parameters of the object; (3) you can add a new object by issuing the command “set env <object> <id> <new_values>” where <id> is a number greater than all existing objects of that type.

For example to delete the second wall object you can used the command “set env wall 1”, assuming that at least two wall objects exist in the current (environment.cworld) environment. To set the coordinate of the first wall to [100, 100, 300, 100] you can issue the command “set env wall 0 100 100 300 100”. To add a new wall with coordinate [100, 100, 100, 400] you should issue the command “set env wall 99 100 100 100 400” (notice that if the current number of existing wall object was, for example, 4, the new id of the new wall will be 4, i.e. the first available id number).

The type of objects that can be created, deleted, or modified through these commands are:

(1) **wall** (i.e. wall objects) that have 4 parameters (the x and y coordinate of the initial and final point in mm).
(2) **round** (i.e. large round objects) that have 2 parameters (the x and y coordinate of the object central point in mm).
(3) **sround** (i.e. small round objects) that have 2 parameters (the x and y coordinate of the object central point in mm).
(4) **light** (i.e. a light bulb) that have 2 parameters (the x and y coordinate of the object central point in mm).
(5) **t_area** (i.e. a circular portion of the ground painted in black) that have 3 parameters (the x and y coordinate of the center of the circular area and the radius of the area in mm).
(6) **start** (i.e. the xy coordinate and direction in degrees of robots initial position). In the case of this last command the value should be set through the graphic interface by dragging and dropping the robots. If the position of the robots has not been modified, the command deletes the corresponding entry.

The current state of the parameters can be saved on a file worldx.env (where x is the id of the current environment) with the command **File->Save** (the command save all define environment independently from the id number used in the file name). Please consider that the parameters saved in the file world?.env (where ? range from 1 to **lifetime.nworlds**) are automatically loaded at runtime.

3.3.4 The **Display->Net_par** command and the other commands that define the characteristics of robots' neural controllers.

This command displays the characteristic of the robots' neural controllers. The architecture of the network is defined by the number of neurons, the meaning of neurons (i.e. whether neurons are sensory, internal, or motor neurons, and what they specifically encode in the case of sensory and motor neurons) the properties of neurons (i.e. whether they have biases or time constant parameters). Some of these properties are defined in the individual parameters and can be modified through the command "set individual <parameter> <value>" (see section 3.4.2). These parameters include the type of sensors and sensory neurons included in the neural controller, the type of motors and motor neurons included in the neural controller, and the number of internal neurons. These parameters are visualized graphically on the right part of the graphic window. More precisely, each neuron has a label that indicates whether what type of sensory or motor neuron it is or whether it is an internal neuron.

The properties of the neurons (i.e. whether they have a bias weight and/or a time constant) are also visualized graphically on the right part of the graphic window (biases are visualized with a black circle around the neuron and time constant are indicated by the pattern with which the neuron is filled that might be full or partially filled in the case of neurons provided or not provided with time constant parameters). The properties of the neurons can be changes by selecting one block of neurons and by using the ***add_remove_neuron_bias*** or ***add_remove_neuron_delta*** icons on the tool bar. To select a block of neuron you should (1) click in an empty place of the graphic window (if you want to eliminate the current selection), (2) click on the first neuron of the block (the neuron become black), (3) click on the last neuron of the block (all the neurons of the block become black). Neurons are ordered from sensors, to internal, to motors and are a graphically displayed from left to right and from bottom to up. You can also change the position of a neuron on the graphic display through the mouse by drag and drop. If a file .net have not been created yet, internal and motor neurons are provided with biases, and sensory, internal, and motor neurons are provided with time constant accordingly to the parameters **individual.delta_inputs**, **individual.delta_hiddens individual.delta_outputs**.

The way in which neurons are connected between themselves and the order with which neurons activation state is updated are defined in the parameters that can be visualized through the Display_Net_par. These parameters consist on an ordered list (with id number starting from 0) of connection and update blocks that indicate, respectively, blocks of neurons receiving connections from another block of neurons and blocks to neurons to be updated. Connection blocks include first a description of the block of neurons that receive connections and then a description of the block of neurons that send connections (4 parameters). Update block include only the block of the neurons to be updated (2 parameters). Each block is indicated with the id number of the first neuron of the block and the number of neurons forming the block. The order of the list is significant since the function that update the neural controller updates neurons' netinput and neurons activation state by following the order of the blocks. As we said above, neurons are ordered from sensors, to internal, to motors are a graphically displayed from left to right and from the lower to the higher layer.

You can delete a block (either a connection or update block) by using the command "set net <id>", where id is the id number of the block. You can modify an existing block by replacing it with an update block by: (1) selecting a block of neurons by clicking on the first and on the last neuron of the block with the mouse (all neurons forming the block will become black), and (2) by issuing the command "set net <id>", were <id> is the number of the block to be replaced. You can modify an existing block by replacing it with a connection block by: (1) selecting a block of receiving neurons by clicking on the first and on the last neuron of the block with the mouse (all neurons forming the block will become black), (2) selecting a block of sending neurons by clicking on the first and on the last neuron of the block with the mouse (all neurons forming the second block will become blue), and (3) by issuing the command "set net <id>", were <id> is the number of the block to be replaced. You can create a new block after the existing blocks by following the same procedure describing for replacing existing block but by using as <id> a number greater than the number of existing block. You can delete all blocks with the command "set net erase".

The current state of the parameters can be save on a file .net with the command **File->Save**. Please consider that the parameters saved in the file evorobot.net, are automatically loaded at runtime. If this file does not exist, the program creates a network architecture on the basis of the parameters contained in the individual folder.

3.3.5 The **Display->Fit_par** command

This command displays the id number of the available fitness function. The current fitness function is defined in the parameter **individual.ffitness**.

3.3.6 The **Display->Statistics** command

This command displays in the graphic window the fitness of the best individuals of each generation for all replications. Moreover, it display in the status bar the list of the seeds of the replications ranked on the basis of the achieved performance. Data are automatically loaded from the files statSx.fit (where x is the seed of the corresponding replication).

If the command is issued while an evolutionary process is running, it only shows the best and average fitness through out generations for the current running experiment.

3.3.7 The **Display->Sample_Data** command

This command displays in the graphic window the sample data of robot's infrared sensors collected for wall, round, and sround objects and the state of the light sensors collected for a light bulb object. Each rectangular area encodes the activation of one of the eight sensors for different angles (x axis, from -180^{o} to $+180^{o}$) and distances (y axis). The color (from black to white) indicates the intensity of the activation.

3.4 The Help menu

The Help menu includes a single command (**Help->About**) that gives general information about Evorobot*.

3.5 Running robots in hardware

As a default, the program evolves or tests the robots in simulation. To run the robots in hardware you should connect a sufficient number of robots to the program trough a Bluetooth wireless connection. Then you can use the standard commands such us Run->Test or Run->Evolution. So far, the connection with real robots has been developed and well tested only within Windows Xp operating system with service pack 2. A Linux version is under testing and will be available as soon as possible.

To connect the robots to Evorobot* you should:

(1) Click on the **enable real robot connection** icon.

(2) Type the command "**set rcon ID PORT**" for each robot to be connected indicating as ID the number of the robot (from 0 on) and indicating as PORT the number of the corresponding port (see below).

Before using the robots for the first time, however, you should configure the Bluetooth wireless connection and you should upload the Evosercom firmware on each robot (Evosercom firmware is a modified version of the sercom firmware developed by EPFL which has been adapted to interact with Evorobot*).

To configure your Bluetooth wireless connection you should:

a) Install and configure the Bluetooth software in order to have a serial port for each robot that you plan to use at the same time (see the manual of your Bluetooth dongle e.g. Bluesoleil, Widcomm, etc.).

b) Identify the PIN of your robots using your Bluetooth software. E-puck robots will appear in your folder or list with their own labels "e-puck_????" where ???? is a four digit number which represents the PIN of the corresponding robot. The PIN number is also written in the body of each E-puck robot just below the speaker.

c) Pair robots with your PC through the pairing procedure provided by your Bluetooth software by indicating the PIN of the robots.

d) Using your Bluetooth software open a serial connection with your E-puck robot/robots, the software will assign a **PORT** number for each robot. You have to indicate this PORT number when you issue the command "set rcon ID PORT" described above.

To upload the Evorobot* firmware (i.e. the file evosercon.hex) on the E-puck robots from your PC you can use the Tiny Bootloader Program freely downloadable on the website http://www.etc.ugal.ro/cchiculita/software/picbootloader.htm.

Due to the limitation of standard Bluetooth, you can connect up to 7 robots, only (i.e. the ID parameter can be between 0 and 6).

4. Compiling the software

To compile Evorobot* you need to install the QT (version 4) library (http://www.trolltech.com/qt/).

The software can be compiled on Windows or Linux. The variables EVOWINDOWS and EVOREALWIN included in the file mode.h should be defined or undefined when the program is compiled on Windows or Linux, respectively.

To compile the program without the graphic interface, the variable EVOGRAPHICS included in the file mode.h should be undefined.

Evorobot* package include a directory "src" which contains: (i) a generic project file evorobot.pro that contains the list of the source files and the flag QT that instruct the compiler to link the QT library, (ii) all the required source files, (iii) a makeProject.bat file which allow you to create the project files for different types of compilers.

.

4.1. Compiling Evorobot* on Windows

To compile evorobot* with opensource technologies you can use the opensource version of the QT and an opensource compiler such us MinGW (and eventually an IDE such us Eclipse). Qt/Windows Open Source Edition is available as a self-extracting installer at http://trolltech.com/developer/downloads/qt/windows. The package provided will also download and install the MinGW compiler, if needed. To build and compile the application: (i) open Qt command prompt, (ii) go in evorobotstar/src directory, (iii) issue the command "`qmake evorobot.pro`" which create the Makefile (or dis-comment the corresponding line in the file makeProject.bat and issue the command makeProject), (iv) issue the command mingw32-make (or the corresponding command in your favourite IDE).

To compile the program with Microsoft Visual Studio and the Commercial Edition of the QT you can create a project file by issuing the command MakeProject.bat after dis-commenting the command line which corresponds to your compiler.

For Microsoft Visual Studio 6 you can create a project file by issuing the command "`qmake -tp vc -o evorobotstar.dsp evorobotstar.pro`" For Microsoft Visual Studio 2005 you can create a project file by issuing the command "`qmake -tp vc -o evorobotstar.vcproj evorobotstar.pro`". Even in these two cases you can create the project file issuing the command MakeProject.bat after dis-commenting the corresponding line. You should then open the project file evorobotstar.vcproj from Visual Studio with the command File->Open_Workspace.

For command-line compiling, set Visual Studio's environmental variables by executing `"VSPATH\Common7\Tools\vsvars32.bat"` where VSPATH is the install direcotory of Visual Studio"). Then, from the working directory create the makefiles by using the command "`qmake evorobotstar.pro`" and then compile by using "`nmake`" ("`nmake -f Makefile.Release`" or "`nmake -f Makefile.Debug`" if you want to select the release or debug mode).

Please notice that the source is indented by using tabs which correspond to 8 characters. To set this parameter properly in Windows Visual Studio, set correctly the parameter in Tool->Option->Text_Editor->All_Languages_Tabs.

4.2 Compiling Evorobot* on Linux

To create a Linux Makefile project file you should: (i) enter in the "src" directory, (ii) issue the command: "`qmake -o Makefile evorobotstar.pro`", (iii) compile the program by issuing the command "`make`". To run the program from an experiment path run "`../bin/evorobotstar`"; there are also two pre-compiled version of the program (evorobotstar-32bit and evorobotstar-64bit) that should work on the different linux distributions.

Please notice that if you want to compile the windows distribution package in linux, you should comment the definition of the flags "EVOWINDOWS" and "EVOREALWIN" in the file "mode.h" and convert the text files from linux by using an utility like dos2unix.

4.3 Compiling the Evosercom firmware

To compile the Evosercom firmware (evosercom.hex) you should download, install, and configure the development tools freely available from the website www.e-puck.org. The Evorobot* firmware package include the workspace file evosercom.mcw, for the MPLAB as well as all the required library files.

5. Overview of the source code and tips on how to modify the program

5.1 Evorobot source files*

The program consists of the following source files:

1) mainwindown.cpp: that contains the functions that handle the main windows, the menu bar, the tool bar, the status bar, etc.
2) renderarea.cpp: that contains the functions that handle the display on the graphic window
3) parameters.cpp: that contains the functions that load, save, and modify the program parameters.
4) main.cpp: that contains the functions that initialize the program and handle robots' lifetime.
5) evolution.cpp: that contains the functions that handle the evolutionary algorithm and the mapping between the selection of individuals forming team of interacting robots.
6) network.cpp: that contains the functions that handle the neural controllers.
7) simulation.cpp: that contains the simulate the robots and the robots/environmental interactions.
8) globals.cpp: that contains the global variables.
9) io.cpp: that contains the functions that load and save the genotype on files.
10) special.cpp: that contains low level functions.
11) defs.h: that contains a declaration of all global variables and program function (except graphic functions contained in mainwindow.cpp and renderarea.cpp.
12) epuckSerComm.cpp: that contains functions that handle the Bluetooth wireless communication with robots (under Windows Xp only)
13) epuck.h: that contains structures useful for the serial communication (only under Windows Xp)

5.2 Program parameters

Program parameters consist of global variable that are defined and initialized in the file globals.cpp and declared in the file defs.h. Since the latter file is included in all source files, global variables can be used in any file without the need to be declared as external. The individual parameters instead (i.e. the parameters that can be visualized through the command Display->Individual) are declared in the structure **ipar** and **ind** the file defs.h and are initialized in the function **init_parameters()** and **init_phenotype**.

Parameters are divided into six categories: evolution, individual, lifetime, display, environment, network. The functions that allow to load and save parameters, visualize parameters on the graphic window, and modify parameters through the command line are: **parse_?_parameters()** and **display_?_parameters**, where ? is evolution, ind, lifetime, display, env, and net depending on the category of the parameter.

To add a new integer evolution, lifetime, or display parameter you should:

(1) create and initialize a global variable in the file globals.cpp,
(2) define the variable as external in the file defs.h,
(3) add the following lines in the function: parse_?_parameters():

```
if (strcmp(word, "parameter") == 0)
{
   parameter = atoi(st);
   done = 1;
}
```

(4) add the following lines in the function: display_?_parameters():

```
sprintf(message,"parameter %d / parameter description ", parameter);
display_message(message);
```

To add a new floating point individual parameter you should:
(1) create a new variable in the structure ipar defined in the file defs.h,
(2) initialize the new parameter in the function init_parameters();
(3) add the following lines in the function: parse_individual_parameters():

```
if (strcmp(word, "parameter") == 0)
{
   pipar->parameter = atof(st);
   done = 1;
}
```

(4) add the following lines in the function: display_individual_parameters():

```
sprintf(message,"parameter %.2f / parameter description", pipar->parameter);
display_message(message);
```

5.3 Fitness functions

The fitness of individuals during the evolutionary process and during the test process is computed within the function **loop()** that indeed: (a) initialize a trial, (b) for every lifecycle of the trial, set the sensors of the robots sensors, update the activation state of the neurons, move the robots, and (c) compute the fitness.

To add a new fitness function you should: (1) identify an available id number (use the Display->Fit_par command to visualize the existing fitness functions), (2) add a case of code similar to the examples below (top part), (3) add four lines or code to describe the meaning of the fitness function within the function: **display_fit_parameters()** as in the example below (bottom part).

```
// Fitness 2 - stay on target
case 2:
  if (read_ground(pind) > 0)
     ind->fitness += 1.0f;
  fitmode = 2;
break;

sprintf(message,"Fitness 2: Discrim (1.0 when a robot is in a target area)");
display_message(message);
if (f == 2)
  strcpy(ffitness_descp, message);
```

In the case of this example the fitness of team of individual is computed after every cycle after each individual robot move. The function read_ground(pind) return the number of the target area in which the robot pind is currently located or 0 if it is not located on a target area. Therefore, the fitness of the first individual of the team (ind->fitness) is increased of 1.0 every time one robot stay on a target. The reason why the fitness of the first robot is increased rather the fitness of the current robot is that the team of robots is selected for producing cooperative behaviors. Therefore the fitness of all robots is

summed in the fitness variable of the first robot, the one that is used to determine whether the corresponding team should be selected or not.

There are three **switch(pipar->ffitness)** instructions in which the fitness can be computed that correspond to the following situations: (1) every lifecycle after each single individual moves, (2) every lifecycle after all individuals of the team move, (3) at the end of the trial. The three location are indicated with the comments “compute fitness 1”, “compute fitness 2”, and “compute fitness 3” respectively. In the first case, the pointer **pind** point to the current robot of the team. In the second and in the third case, if the team consist of just a single robot, the pointer **ind** (that always point to the first robot of the team) can be used. If the team consists of more than one robot, the **pind** pointer should be initialized to **pind** and incremented to verify the state of all robots of the team as in the following example (in which the fitness is increase of 1.0 only when all the robots of the team are on the target):

```
// Fitness 7 – all robots of the team should stay on target
case 7:
  nrobots=0;
  for(team=0,pind=ind; team<nteam; team++, pind++)
   {
      if (read_ground(pind) > 0)
              nrobots++;
   }
  if (nrobots == nteam)
    ind->fitness += 1.0f;
  fitmode = 2;
break;
```

The variable **fitmode** can assume the value 1, 2, or 3, and determine whether the total fitness gathered is divided for: (1) the number of lifecycle multiplied by the number of trials; (2) the number of trails, (3) or 1.0 (i.e. whether it is not normalized). This normalization is computed within the **eval_team()** function.

Different fitness functions might need to make operations on different variables. In general terms, often relevant information is contained into the structure **individual ind** and **environment env[ ]** (see the definition in the file defs.h).

5.4 Sensors and Motors

The state of the sensors is updated within the function set_input() that compute the value of the vector **input[ ]** that contains the state of all existing sensors. Robots’ current sensors are indicated in the parameters of the class individual that correspond to the structure **parameters ipar**.

To add a new sensor you should: (1) add a new individual parameter (see section 5.2), (2) update the number or sensory neurons of the neural controllers (pind-> inputs see the example below), (3) add a piece of code within the set_input() function (see the example below), (4) add the label/labels for the sensory neurons corresponding to the new sensor in the function **create_neurons_labels()**.

```
if (pipar->groundsensor   == 1)
       pind->ninputs += 1;
```

The two lines of code above are included in the **compute_parameter()** function. This function computes the number of sensory neurons of the neural controllers by updating the variable **pind->ninputs** that encode the number of sensory neurons of the controllers. The size of the genotype of each individual (that correspond to the number of free parameters) is computed by the function pseudo_activate_net().

```
// ground sensor
if (pipar->groundsensor == 1)
{
       if (read_ground(pind) > 0)
              pind->input[nsensor] = (float) 1.0;
       else
              pind->input[nsensor] = (float) 0.0;
```

```
            nsensor += 1;
        }
```

The example above indicate how a the state of a ground sensor is stored in the pind->input[ ] vector. The initial line checks whether the sensor currently exists. In that case, the input vector of the current individual (pind) is set to 1.0 or to 0.0 depending on whether the robot is located on a target area or not. The local variable **nsensor** is increase of 1 unit so that the value of other forthcoming existing sensors will be stored correctly in the pind->input vector.

To add a new motor or actuator you should: (1) add a new individual parameter (see section 5.2), (2) update the number or motor neurons of the neural controllers (pind->outputs see the example below), (3) use the state of the corresponding motor neuron/neurons to modify the robot/environmental relation (this can be done, for example, by calling a new function just after **setpos()**, the function that move the robot on the basis of the state of the first two motors neurons that control the speed of the two wheels), (4) add the label/labels for the sensory neurons corresponding to the new motor in the function **create_neurons_labels()**.

5.5 The evolutionary algorithm

The function that handle the evolutionary algorithm are **runevolution()** and **evolution()** where the former simply call the latter for a certain number of replications after performing some initialization activity. The **evolution()** function call: (a) the function **eval_team()** that set the free parameters of the team of individual to be tested which then call the function **loop()** that make the robots interact with the environment, and (b) the function **reproduce()** that create a new generation of genotype by selecting the best individuals and allowing them to replicate by adding mutations during the reproduction process.

The function **loop()** that test a team of robots for a trial, call the function **set_input()** to update the state of the sensors, **update_net()** to update the state of internal and motor neurons, **setpos()** to move the robot on the basis of the state of the motor neurons.

The function that handle the test of one individual or all best individuals are **test_one_individual()** and **test_all_individuals().** Both functional call **eval_team()**, i.e. the same function used for testing individuals or team of individuals during the evolutionary process (see above).

5.6 Important variables and data structure

Beside the program parameters, other important data structures deserve some attention. The genome of the population, that is initially created randomly, is store on the matrix **genome[ ][ ]**. Each line of this matrix contains the genotype of a corresponding individual (or team of individuals) and each element of a line contains 8 bits (encoded in an integer value). Each integer corresponding to 8 bits encodes a free parameter of a corresponding phenotype (e.g. a connection weight, a bias, or a time constant). The genome matrix is also used to store the genotype of the best individuals loaded from a file. The program allocates enough space for storing in memory the genotype of the individuals of the population + the genotype of additional individuals (**evolution.additional_ind**).

During the reproduction process the selected individuals are moved to the matrix **bestgenome[ ][ ]**. Then, the mutated version of these reproducing individuals are copied back into the **genome[ ][ ]** matrix (i.e. only the current generation is stored in memory).

Before an individual or team of individuals is tested, one of the genotype is copied from one line of the **genome[ ][ ]** matrix to the vector or to the vectors **pind->freep**, where **pind** is a structure individual containing several variable that encode the state of the individual.

Since the individual structure **pind** are used to test different genotype sequentially (i.e. one after the other), the program do not create a number of individual structure equal to the number of individuals in the population. For example, a typical experiment might involve a population with 100 genotype stored in the **genome[ ][ ]** matrix, a single structure **parameter ipar** that encode the variable of all individuals (i.e. the type of sensors and motors, their actual position and orientation in the environment etc.), and four structures **individual pind** that encode the phenotype of four identical robots constructed on the basis of the same genotype.

Finally the vector of structures environment **env[ ]** contains the parameters of 1 to 5 different environments.

5.7 Interface between Evorobot and the Evosercom firmware*

When Evorobot* is connected to the real robots, to program do not need to simulate the state of the robots' sensors or the modification of the robot position and orientation as a result of the robots' motor action. In this case, in fact, the program can read the state of the sensors directly from the robots.

The function which allows Evorobot* to acquire the state of the robots' sensors through the Bluetooth connection is **queryRobotsSensors().** The function which is used to update the sensory neurons of the robots' neural controller is **set_input_real()**. The function which allows Evorobot* to set the desired state of the motors in hardware is **set_pos_real()**.

6. References

Mondada F., Bonani M. (2007). The e-puck education robot. http://www.e-puck.org/.

Nolfi S. & Floreano D. (2000). Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines. Cambridge, MA: MIT Press/Bradford Books.

Acknowledgments

The authors thank Giuseppe Morlino who structured the download package and prepared the compiling instructions and the other members of the Laboratory for Autonomous Robotics and Artificial Life at ISTC-CNR (laral.istc.cnr.it) for useful comments. The development of Evorobot* has been supported by the ECAGENTS project funded by the Future and Emerging Technologies programme (IST-FET) of the European Community under EU R&D contract IST-1940.