The XCS Library (`xcslib`)

September 2008

Pier Luca Lanzi, `pierluca.lanzi@polimi.it`
Daniele Loiacono, `loiacono@elet.polimi.it`

Abstract

This document is a short introduction to the XCS Library (`xcslib-1.0`). It explains how to compile and install the libary `xcslib` and how to run the examples included in the distribution. The current version (1.0) implements both the XCS classifier system as described in [8, 9, 2], including the recent extensions discussed in [3], and XCSF, the version of XCS with computed prediction [10].

Contact Information

Pier Luca Lanzi & Daniele Loiacono
Politecnico di Milano
Dipartimento di Elettronica e Informazione
`pierluca.lanzi@polimi.it`
voice: +39-02-23993472
fax: +39-02-23993411

Installation

To install `xcslib` first unpack the source code by typing:

```
tar zxvf xcslib-1.0.tar.gz
```

the command will create the directory `xcslib-1.0` containing the main sources and three subdirectories:

`./utility/` contains the scripts and the utilities to produce data for plotting;

`./docs/` contains the code documentation produced by `doxygen`;

`./examples/` contains example problems for XCS taken from [8] and [9] and for XCSF [6].

By default all the executables are copied in the directory `$(HOME)/bin`. This directory can be created by typing:

```
mkdir $(HOME)/bin
```

the directory should also be added to the `PATH` environment variable. If you are using `bash` just type:

```
export PATH=$PATH:$HOME/bin
```

To list all the possible options type `make` that will produce,

```
make
   print this help

make all
  build all the available executables
```

```
make mp
  build the version of XCS to learn Boolean multiplexer

make fa
  build the version of XCSF to approximate functions

make gw
  build the version of XCSF to solve the 2D gridworld

make utility
  build and install the utilities
```

To compile and install all the available versions of XCS and XCSF including the utilities, type "`make all`", which will install everything in the directory `$HOME/bin/`. To specify another installation directory (e.g. `mydir`), type "`make all INSTALL=mydir`". Alternatively, it is possible to compile each single version of XCS. For instance, the command "`make mp`" will compile the version of XCS that can be used to learn Boolean multiplexer (`xcs-mp`).

Running the Examples

After the executables have been generated and installed, it is possible to run the examples that are included in the distribution.

Boolean Multiplexer

As a very first example, we will apply XCS to learn the Boolean multiplexer of 6 bits (or 6-multiplexer) with the same parameter settings used in in [8]. First, move to the directory `xcslib-1.0/examples/6-multiplexer` which contains the configuration file for the experiment, i.e., "`confsys.mp6`". The experiment consists of ten runs each one consisting of 10000 learning problems and 10000 testing problems (as in [8], learning problems and testing problems alternate). To check whether the executable of XCS for the Boolean multiplexer has been installed correctly type "`which xcs-mp`". The command should return the directory where the file "`xcs-mp`" has been installed; if the default settings have used, the command should return `$(HOME)/bin/xcs-mp`, where "`$(HOME)`" is the home directory of the current user. At this point, the experiment can be started by typing:

```
    xcs-mp -f mp6
```

which will print the parameters setting read from the configuration file and it will trace the execution of each run, as follows,

```
1/10 saving the experiment state... ok
1/10 saving the population state... ok

...

10/10 saving the experiment state... ok
10/10 saving the population state... ok
```

To avoid such printing, the experiment can be run in background by typing, `xcs-mp -f mp6 >& /dev/null &`. While the experiment is running, it is possible to monitor the statistics that the system keeps about the current run by typing "`tail -f statistics.mp6-?`". The command will print seven scrolling columns like:

```
...
3 13745 1 1000 48 43.6233 Testing
3 13746 1 1000 48 1.94525 Learning
3 13747 1 1000 48 2.77569 Testing
3 13748 1 1000 48 24.9166 Learning
3 13749 1 1000 48 73.961 Testing
...
```

The first column reports the current run; the second column reports the current problem, the third column reports the number of steps required to solve the problem (note that, since the 6-multiplexer is a one-step problem, the third column is always one in this experiment); the fourth column reports the sum of the rewards the system obtained during the problem; the fifth column reports the number of macroclassifiers in the population; the sixth column reports the absolute difference between the reward predicted by the system and the actual reward received from the environment; finally, the seventh column indicates whether XCS has solved the problem in *learning* mode or in *testing* mode. The columns will continue to scroll until the run ends or the `tail` command is interrupted by a `control-C`. When the execution is completed, the directory will contain three types of files. Experiment files (e.g., `experiment.mp6-0.gz`) contain the state of the each experiment and they can be used to restart an experiment from the point it was stopped. Statistics files (e.g., `statistics.mp6-5.gz`) contain the most relevant data collected during each experiment such as, the number of steps, the reward obtained, the population size, and the prediction error. Population files (e.g., `population.mp6-3.gz`) contain the final population evolved during for each experiment. For instance, we can inspect the final population produced in the fourth run by typing:

```
    zcat population_report.mp6-0 | more
```

which will produce an output like:

```
...
44385 01#0## : 0 1.000e+03 2.274e-13 9.996e-01 2.910e+01 1623 28
44373 10##1# : 0 1.171e-57 1.524e-55 1.000e-00 2.694e+01 585 27
44379 01#1## : 0 6.830e-57 2.989e-54 9.999e-01 2.896e+01 564 27
44376 10##0# : 1 2.540e-54 2.922e-52 9.997e-01 2.827e+01 558 26
44406 001### : 1 1.000e+03 2.274e-13 9.342e-01 2.876e+01 1305 26
44379 01#1## : 1 1.000e+03 2.274e-13 9.657e-01 2.744e+01 1313 26
44337 11###1 : 0 6.309e-45 6.046e-43 9.997e-01 2.508e+01 321 24
44391 11###0 : 0 1.000e+03 1.337e-160 9.991e-01 2.330e+01 1657 24
44370 000### : 0 1.000e+03 2.274e-13 9.995e-01 2.437e+01 1721 24
44340 11###1 : 1 1.000e+03 1.137e-13 1.000e-00 2.459e+01 1441 23
...
```

Where, column 1, is a unique identifier of the classifier; column 2, reports the classifier condition; column 3, reports the classifier action; column 4, reports the prediction; column 5, reports the prediction error; column 6, reports the fitness; column 7, reports the action set size; column 8, reports the experience (the number of updates); and column 9, reports the numerosity.

To analyze the results produced we will use the utility program we installed at the beginning. First to produce the plot for the performance just type:

```
    prepare_rwd.sh -f mp6 -w 100 -i 100
```

this command asks for the results regarding the reward obtained by the system as a moving average over the last 100 problems (command "`-w 100`") and to report only such data every 100 problems (command "`-i 100`"). The command will produce (i) ten files, from `perf.mp6-0` to `perf.mp6-9`, reporting the XCS performance in each run as the moving average of the reward received in the last 50 problems; (ii) the file `AVERAGE.perf.mp6-10` containing the average of the ten performance files. Likewise, we can produce the statistics regarding the number of classifiers in the population, by typing:

```
    prepare_pop.sh -f mp6 -w 100 -i 100
```

The command will produce (i) ten files, from `popul.mp6-0` to `popul.mp6-9`, reporting the XCS performance in each run as the moving average of the reward received in the last 50 problems; (ii) the file `AVERAGE.popul.mp6-10` containing the average of the ten performance files. Finally in the directory there is the file `report.mp6-10` containing the following summary of the CPU time elapsed for the overall computation, for each experiment, and for each problem:

```
TOTAL ELAPSED TIME              9.4
DETAILS FOR EXPERIMENTS

 EXP.          ELAPSED IN EXP.          AVG FOR PROB.
    0                  0.95sec               9e-05sec
    1                  0.95sec             8.6e-05sec
    2                  0.97sec             8.7e-05sec
    3                  0.91sec               8e-05sec
    4                  0.92sec             8.1e-05sec
    5                  0.93sec             8.9e-05sec
    6                  0.95sec             9.3e-05sec
    7                  0.87sec               8e-05sec
    8                  0.97sec             9.2e-05sec
    9                  0.99sec             9.3e-05sec
```

Restarting an experiment

Suppose now, we wish to continue the previous experiment for other 10000 problems. To do this we have just to edit the `confsys.mp6` file and to replace the line:

```
    first problem = 0
```

with the following line:

```
    first problem = 10000
```

which specifies that additional 10000 learning problems (and 10000 testing problems) should be solved by starting from problem 10000. Now we can restart the experiment by typing:

```
xcs-mp -f mp6 >& /dev/null &
```

Note that as the new series of experiments begins the files about the experiments statistics will be uncompressed and the information about the new problems will be added.

```
..............................
.QQF..QQF..OQF..QQG..OQG..OQF.
.OOO..QOO..OQO..OOQ..QQO..QQQ.
.OOQ..OQQ..OQQ..QQO..OOO..QQO.
..............................
..............................
.QOF..QOG..QOF..OOF..OOG..QOG.
.QQO..QOO..OOO..OQO..QQO..QOO.
.QQQ..OOO..OQO..QOQ..QOQ..OQO.
..............................
..............................
.QOG..QOF..OOG..OQF..OOG..OOF.
.OOQ..OQQ..QQO..OQQ..QQO..OQQ.
.QQO..OOO..OQO..OOQ..OQQ..QQQ.
..............................
```

Figure 1: The `Woods2` Environment.

Woods Environments

In the second example, we apply XCS to control an artificial animal (or animat [7]) that must learn how to survive in an artificial environment. `Woods2` (see Figure 1) is a grid with two types of obstacles (represented by “`O`” and “`Q`” symbols), food positions (represented by “`F`” and “`G`” symbols), and empty positions (represented by “`.`” symbols). `Woods2` is a torus: its left and right edges are connected as well as its top and bottom edges. The animat, controlled by XCS, can occupy any of the empty positions and it can move to any adjacent position that is empty. The animat has eight sensors, one for each adjacent position, that are encoded by three bits coding features of the object. Thus, the animats sensory input is a string of 24 bits (3 bits $\times$ 8 positions). There are eight possible actions, one for each possible adjacent position. The animat must learn how to reach food positions starting from any empty position. When it reaches a food position (`F` or `G`) the problem ends, and the animat receives a constant reward equal to 1000; otherwise it receives zero.

To apply XCS to `Woods2`, we first have to move into the directory `xcslib-1.0/examples/woods2` which contains the configuration file for the experiment (`confsys.woods2`), the map of the environment (`woods2.map`), and a `gnuplot` file to generate the plots for performance and population size. The experiment consists of twenty runs of 5000 learning problems and 5000 testing problems each (as in [8, 9], learning problems and testing problems alternate). To check whether the executable of XCS for the woods environment has been installed correctly type “`which xcs-woods2`”. The command should return the directory where the file “`xcs-woods2`” has been installed; if the default settings have used, the command should return `$(HOME)/bin/xcs-woods`, where "`$(HOME)`" is the home directory of the current user. At this point, the experiment can be started by typing:

```
xcs-woods2 -f woods2
```

which will again print the parameters setting read from the configuration file and it will trace the execution of each run as follows,

```
1/20 saving the experiment state... ok
1/20 saving the population state... ok
...
20/20 saving the experiment state... ok
20/20 saving the population state... ok
```

The execution will produce several files containing information about the state of each experiment (whose name starts with `experiment.woods2`), the final populations (whose name starts with `population.woods2`), and the statistics collected for each problem (whose name starts with `statistics.woods2`).

Woods environments are sequential (or multistep) problems and to analyze the performance of XCS we need to generate the data of the average number of steps that the animat needs to reach goal position. This can be achieved by using the scripts provided in the distribution and typing,

```
prepare_steps.sh -f woods2 -w 50 -i 50
```

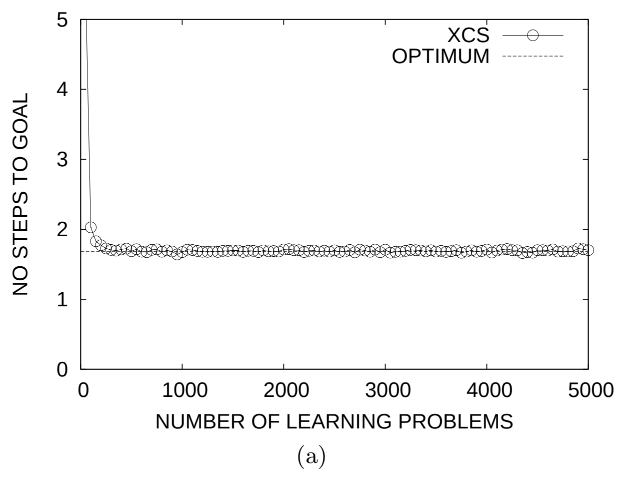
which generates the results regarding the number of steps required to reach goal by the system as a moving average over the last 50 test problems (command "`-w 50`") and to produce only one data point every 50 problems (command "`-i 50`"). The command will produce (i) twenty files, from `steps.woods-0` to `steps.woods2-19`, reporting the XCS performance in each run as the moving average of the reward received in the last 50 problems; (ii) the file `AVERAGE.perf.woods2-20` containing the average of the twenty performance files. Similarly, we can produce the statistics regarding the number of classifiers in the population, by typing:

```
prepare_pop.sh -f woods2 -w 100 -i 100
```

The two average files can be used to produce the typical performance and population plots [8, 9] as the ones showed in Figure 2 by executing `gnuplot` on the configuration file `plot_woods2.gpi` that is provided in this distribution.

XCSF for Function Approximation

In the third example, we will apply XCS with computed prediction (briefly XCSF) to approximate a simple real-valued function, i.e., a sinusoidal function with period 1 and defined between 0 and 1. The experiment is similar to the one discussed in [10] and it uses using piece-wise linear approximators updated using Normalized Least Mean. The configuration file for this experiment, `confsys.sin`, can be found in the directory `xcslib-1.0/examples/functions` where we have move to begin the experiment. To run XCSF on this simple function approximation problem just type "`xcsf-fa -f sin`"; as before, the parameter settings read from the configuration file will be printed and then the execution of ten runs will be traced as follows,



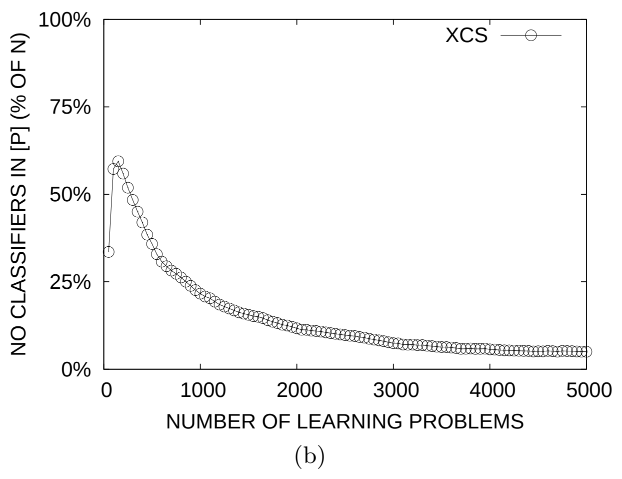


Figure 2: XCS applied to `Woods2`: (a) performance; (b) number of classifiers in the population.

```
1/10 saving the experiment state... ok
1/10 saving the population state... ok
...
10/10 saving the experiment state... ok
10/10 saving the population state... ok
```

As in the previous examples, the execution will generate the usual files of the final populations, the files of the experiment states which can be used to restart the experiments, and the files of the statistics. When applied to function approximation problems, the performance of XCSF is measured as the approximation error obtained [10] and the learning process is analyzed by plotting how the approximation error changes as the learning proceeds. While generalization is again evaluated by tracing how the number of classifiers changes during evolution. To generate the data regarding the evolution of the approximation error type,

```
prepare_se.sh -f sin -w 100 -i 100
```

This command generates the data file of the approximation error as a moving average over a window of 100 test problems (command "`-w 100`") and it reports only one data point every 100 test problems (command "`-i 100`"). As usual, the data of the population size can be obtained as in the previous examples by typing,

```
prepare_pop.sh -f sin -w 100 -i 100
```

Figure 3 reports the plots of the approximation error and population size for this example that have been generated using the `gnuplot` configuration file provided in the distribution. To generate the plots just type "`gnuplot plot_sin.gpi`." As can be noted in Figure 3a the approximation error rapidly goes below the error threshold *epsilon zero* (or ε_0) that has been specified in the configuration file (`confsys.sin`). The same directory also provides other configuration files to perform experiments on other well-known functions [4, 5, 6].

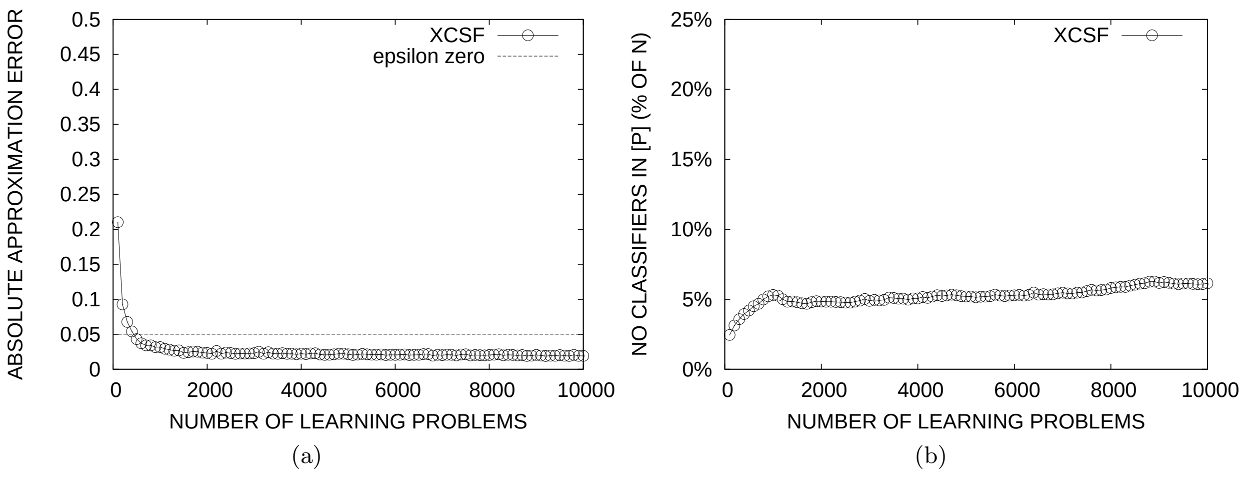


Figure 3: XCSF applied to approximate a sinusoidal function with period 1 and defined between 0 and 1: (a) approximation error; (b) number of classifiers in the population.

XCSF for the Two-Dimensional Gridworld

In the fourth example, we will apply XCSF to the 2D gridworld (firstly introduced in [1]) in which the current state is defined by a pair of real coordinates $\langle x, y\rangle$ in $[0,1]^2$, the goal is in position $\langle 1,1\rangle$, and there are four possible actions (left, right, up, and down); each action corresponds to a step of size s in the corresponding direction; in this experiment we set s as 0.05; actions that would take the system outside the domain $[0,1]^2$ take the system to the nearest position of the grid border. The system can start *anywhere* but in the goal position and it reaches the goal position when *both* coordinates are equal or greater than one. When the system reaches the goal it receives 0, otherwise it receives -0.5.

To perform this experiment, first we have to move to `xcslib-1.0/examples/2d-gridworlds` where the configuration file (`confsys.2dgrid`) is found. Then, to start the experiment, we must type,

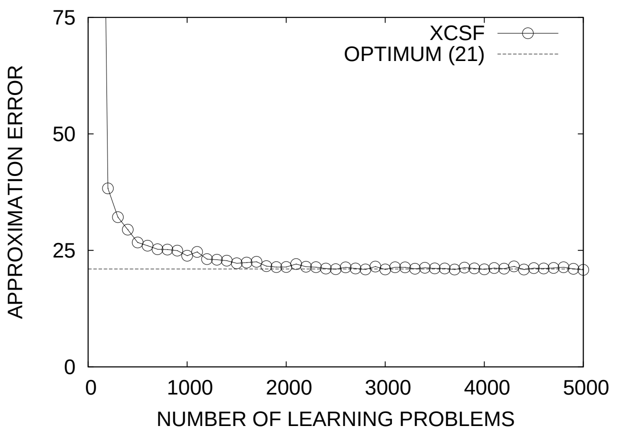
```
xcsf-gw -f 2dgrid
```

the parameter settings read from the configuration file will be printed and then the execution of ten runs will be traced as follows,

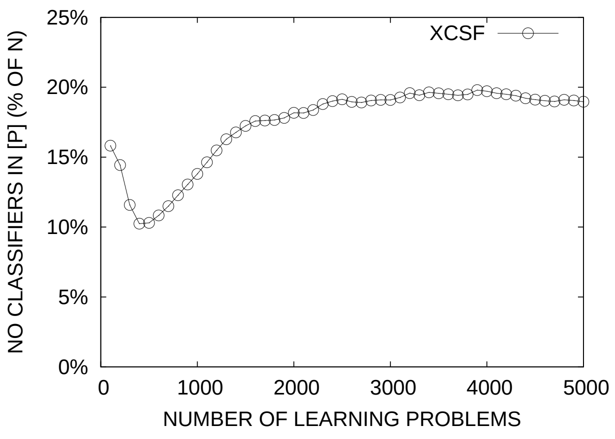
```
1/10 saving the experiment state... ok
1/10 saving the population state... ok
...
10/10 saving the experiment state... ok
10/10 saving the population state... ok
```

When the execution stops, we can generate the data files for the performance (computed as the average number of steps to the goal position) and the number of classifiers in the population by typing,

```
prepare_steps.sh -f 2dgrid -w 100 -i 100
```



(a)



(b)

Figure 4: XCSF applied to the two-dimensional gridworld [1]: (a) performance; (b) number of classifiers in the population.

and,

```
prepare_pop.sh -f 2dgrid -w 100 -i 100
```

The two data files generated by these commands (`AVERAGE.steps.2dgrid-10` and `AVERAGE.popul.2dgrid-10`) can be used to generate the plots for the performance and population size reported in Figure 4 using the gnuplot script (`plot_2dgrid.gpi`) that has been included in the directory.

Other Examples

The distribution include other examples involving several problems that has been used in the literature including woods1 [8], the hidden parity [], the count ones [], and the chain environment []. All the executables to run these examples can be compiled using the file `makefile.others`. To check all the other versions of XCS available in the distribution just type `make -f makefile.others`. For instance, the version of XCS that solves the chain problem can be generated by typing `make -f makefile.others chain`. The configuration files are available in the subdirectories of the the directory `examples`.

Using XCSLib as a Library

XCSLib not only implements XCS [8, 9] and XCSF [10] but it also can be used to build programs based on these models using the classes provided in the distribution.

What Classifiers Match?

For instance, we can use the classes provided to write a program that takes as input a population evolved by XCS (using binary inputs, binary actions and ternary conditions), a sensory input (possibly also an action), and it prints all the classifiers in the population that match the current input and advocate the specified action (if provided). The source code of the program is reported as Listing 1.

To read a classifier population, the program must first include all the basic classes (binary inputs, binary actions, ternary conditions and classifiers) and the class `xcs_config_mgr2` that is used to read configuration files (lines 1-3 in Listing 1)). Then, the parameters specified by the users are read from the command line (lines 11-25), the configuration file specified by the user is read (line 34) and the classes for binary actions and ternary conditions are initialized using the configuration parameters read from the file (lines 36-40). After the classes are initialized, the population file is opened (line 49) and the classifiers can be read from the population file one by one (line 51); classifiers are matched against the input provided by the user (line 53) and (if specified) also against an action (lines 55 and 57); and only the matching ones are printed (line 59). To compile, move into the directory containing the source files and type,

```
make -f make/xcs.make MATCH VERSION=tb
```

which will produce the executable `match-tb` (i.e., match for ternary conditions and binary actions). The compiled program can be applied to print all the classifiers in a population that matche a certain input. For instance, the command,

```
match mp6 population.mp6-0 100001
```

reads the configuration file `confsys.mp6` and it prints all the classifiers in the population file `population.mp6-0` that match the input `100001`.

Listing 1: Match Utility

```
#include "xcs_definitions.hpp"
#include "xcs_classifier.hpp"
#include "xcs_config_mgr2.hpp"

int
main(int argc, char *argv[])
{
  unsigned long act;
  bool    select_action = false;

  if ( (argc!=4) && (argc!=5) )
  {
    cout << endl << endl;
    cout << argv[0] << "<prefix>␣<population>␣<input>␣[<action>]␣" << endl;
    cout << "\t<suffix>␣specify␣the␣configuration␣file" << endl;
    cout << "\t<population>␣file␣containing␣classifier␣population" << endl;
    cout << "\t<input>␣sensory␣input␣" << endl;
    cout << "\t<action>␣classifier␣action␣to␣be␣selected" << endl;
    cout << endl << endl;
    exit(-1);
  }

  string    suffix = string(argv[1]);
  string    population = string(argv[2]);
  string    inputs = string(argv[3]);

  if (argc==5)
  {
    act = atoi(argv[4]);
    select_action = true;
  }

  xcs_config_mgr2 xcs_config(suffix);     //! init the configuration manager

  t_action    dummy_action(xcs_config); //! init the action class

  t_condition   dummy_condition(xcs_config);  //! init the condition class

  string    string_condition;
  t_condition condition;
  xcs_classifier  classifier;

  unsigned long no_actions = dummy_action.size();

  vector<xcs_classifier>  set;    //! population is a vector of classifiers
  ifstream  INPUT(population.c_str());

  while (INPUT>>classifier)
  {
    if (classifier.match(t_state(inputs)))
    {
      if (select_action)
      {
        if (classifier.action==t_action(act))
        {
          cout << classifier << endl;
        }
      }
      else
        cout << classifier << endl;
    }
  }
}
```

References

[1] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 369–376, Cambridge, MA, 1995. The MIT Press.

[2] Martin Butz and Stewart W. Wilson. An algorithmic description of XCS. *Soft Comput.*, 6(3-4):144–153, 2002.

[3] Martin V. Butz, David E. Goldberg, and Pier Luca Lanzi. Gradient descent methods in learning classifier systems: Improving XCS performance in multistep problems. *IEEE Transaction on Evolutionary Computation*, 9(5):452–473, October 2005.

[4] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Extending XCSF beyond linear approximation. In *Genetic and Evolutionary Computation – GECCO-2005*, pages 1859–1866, Washington DC, USA, 2005. ACM Press.

[5] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Prediction update algorithms for XCSF: RLS, kalman filter, and gain adaptation. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1505–1512, New York, NY, USA, 2006. ACM Press.

[6] Pier Luca Lanzi, Daniele Loiacono, Stewart W. Wilson, and David E. Goldberg. Generalization in the XCSF classifier system: Analysis, improvement, and extension. *Evolutionary Computation Journal*, 15:133–168, 2007.

[7] Stewart W. Wilson. Classifier systems and the animat problem. *Machine Learning*, 2(3):199–228, 1987.

[8] Stewart W. Wilson. Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2):149–175, 1995. http://prediction-dynamics.com/.

[9] Stewart W. Wilson. Generalization in the XCS classifier system. pages 665–674. Morgan Kaufmann: San Francisco, CA, 1998. http://prediction-dynamics.com/.

[10] Stewart W. Wilson. Classifiers that approximate functions. *Natural Computing*, 1(2-3):211–234, 2002.