COBOSLAB

Cognitive Bodyspaces: Learning and Behavior

COBOSLAB Report Y2009N001

October 23, 2009

Documentation of *JavaXCSF*

Patrick O. Stalph & Martin V. Butz

COBOSLAB, Psychologie III
University of Würzburg
Röntgenring 11
97070 Würzburg, Germany
http://www.coboslab.psychologie.uni-wuerzburg.de

Documentation of *JavaXCSF*

Patrick O. Stalph[*] Martin V. Butz[†]

Abstract

This report gives an overview of the *JavaXCSF* code and explains, where to get the code. Furthermore, the settings and features are described briefly. The document also explains how to use the code and how to extend it. *JavaXCSF* is an implementation of the XCSF Learning Classifier System that is used for function approximation. The code contains four types of conditions, namely rectangular, ellipsoidal, rotating rectangular, and rotating ellipsoidal conditions. In addition to a simple constant predictor based on the Widrow-Hoff rule, implementations of a linear recursive least squares (RLS) prediction and a quadratic RLS prediction are included. Eleven test functions may serve as example problems. In order to adapt to current state of the art CPU's, which usually have more than one core, the code also optionally supports parallelized matching to exploit the full power of multicore CPU's by using multiple parallel threads.

Keywords: Learning Classifier Systems, XCSF, Java, implementation, function approximation

1 Introduction

Implementing a Learning Classifier System is a non-trivial task and debugging such a complex system can take a large amount of time. The code package described here is a recent implementation of the XCSF Learning Classifier System in Java. It is freely available to the public.

Learning Classifier Systems were introduced by John H. Holland (Holland, 1992) and strongly contribute to the field of *genetics-based machine learning*, that is, machine learning techniques that make use of evolutionary algorithms. The most prominent Learning Classifier System is XCS (Wilson, 1995, 1998). Recent research also emphasized the function approximation variant, that is XCSF (Wilson, 2002; Butz, Lanzi, & Wilson, 2008).

The remainder of this report is organized as follows. Section 2 describes, where the code can be downloaded and the requirements for running or

[*]patrick.stalph@psychologie.uni-wuerzburg.de
[†]butz@psychologie.uni-wuerzburg.de

extending the code. Furthermore, a minimal example shows how to use the code and the available settings are briefly explained. Going further into the details, Section 3 lists the condition and prediction structures delivered with this code package and Section 4 explains the visualization and logging mechanisms that are available. Directions for developers, who wish to extend the code, can be found in Section 5, where the structure of the package is explained, and in Section 6, where details about the common interfaces are given. The topic of parallelization is tackled in Section 7, where the one parallelized class for matching is explained. Final comments conclude this report.

2 Getting Started

The *JavaXCSF* code can be downloaded from:

`http://www.coboslab.psychologie.uni-wuerzburg.de/code/`

The zip file includes the source code, generated JavaDoc, an executable JAR package and three text files to specify the XCSF setup, when running the the main method. To run the JAR package, which starts examplary function approximations, it is sufficient to have a recent version of Java installed. Unpack the content and execute the following command.

```
java -jar xcsf.jar
```

The detailed prerequisites for the complete functionality of *JavaXCSF* are described in the following paragraphs.

Java. In order to run the core funcionality of the code, it is sufficient to have a recent version of Java (at least 1.5) installed. The Java Runtime Environment (JRE) can be downloaded from `http://java.com/`. Developers who want to implement their own learning problems or extend other parts of the code need a recent version (at least 1.5) of the Java Development Kit (JDK), which can be downloaded from `http://java.sun.com/`. Java is sufficient for the core functionality, however, the code base also includes classes for visualization and logging purposes that make use of Java 3D and gnuplot. These classes are optional and the code will run without.

Optional: Java 3D. The visualization class for the condition structure actually contains two visualizations. If the input space is two-dimensional, the visualization is done using Java's internal `java.awt.Graphics2D` package. However, if the input space is three-dimensional, the class checks for Java 3D, which can be downloaded from `https://java3d.dev.java.net/` and then starts the three-dimensional visualization. Users that want to use the three-dimensional condition structure visualization require this additional package—two-dimensional condition visualization runs nicely without.

Optional: gnuplot. Gnuplot is a famous scientific plotting package for two- and three-dimensional visualization of data. The free package can be downloaded from `http://www.gnuplot.info/` and the following four classes make use of gnuplot.

- `PredictionPlot.java` plots the currently predicted function.
- `PredictionErrorPlot.java` plots the average prediction error.
- `PerformanceWriter.java` writes XCSF's performance to a file (works without gnuplot) and then plots the average prediction error, generality and population size to a file.
- `OutputWriter2D.java` creates a snapshot of the two-dimensional condition structure and plots the predicted function; both images are written to the filesystem.

Calling an external program is a system-dependet task—the code was tested on Windows as well as Linux and default executables are implemented for these systems. On Windows, the gnuplot executable is expected to be found at:

```
C:\<localized program files>\gnuplot\bin\pgnuplot.exe
```

where the exact name of the *program files* directory depends on the language set for your OS. For Linux systems, the code expects the path to gnuplot beeing set properly, such that calling `gnuplot` from anywhere suffices.

For other operating systems and differing installation directories, the executable can be set in a public, static field before running XCSF.

```
xcsf.XCSFUtils.Gnuplot.executable = "foo/bar/executable";
```

2.1 Basic Workflow

For a detailed description of XCSF the interested reader is referred to (Wilson, 2002; Butz et al., 2008). The following pseudocode illustrates the basic workflow of this implementation.

```
for n iterations do
  x,y = function.nextSample()
  ms = createMatchset(x)
  if matchset is empty then
    covering(x)
  end if
  p = ms.getWeightedPrediction(x)
  ms.updateClassifiers(p, y)
  evolution(ms)
end for
```

Supose, XCSF is approximating a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$. In each iteration XCSF processes one sample, which consists of an n-dimensional input vector $\vec{x}$ and the corresponding m-dimensional function value $\vec{y} = f(\vec{x})$. First, the matchset for the current input $\vec{x}$ is created, that is, the set of all classifiers, whose condition is satisfied by $\vec{x}$. If no classifiers match, the covering mechanism kicks in and creates a default classifier that matches the input. During initial iterations this is often the case, because XCSF starts with an empty population. The matchset is then used to predict the function value at $\vec{x}$ and classifiers in the matchset are updated using the actual function value $\vec{y}$ and the generated prediction $\vec{p}$. Finally, the steady state genetic algorithm reproduces and modifies two classifiers of the matchset.

2.2 General Use Case

The general use case for *JavaXCSF* consists of five steps:

1. Instantiate the function to be learned,
2. load the desired settings from a properties file,
3. instantiate the XCSF object,
4. optionally add listeners, and finally
5. start the experiment(s).

Using the sine function as an example (see the `xcsf.examples` package), the corresponding java source code looks like this:

```java
Function f = new Sine(1, 4, 0, 2);
XCSFConstants.load("xcsf.ini");
XCSF xcsf = new XCSF(f);
xcsf.addListener(new PerformanceGUI(true));
xcsf.runExperiments();
```

The listener concept in *JavaXCSF* realizes the observer design pattern and thus clearly distinguishes between core functionality and visualization or logging. This is extreemly useful when running the code on remote machines, where no graphics are available at all. Listeners are informed about the current progress at the end of every iteration and access to the population and matchset is permitted.

2.3 The Settings File

Although optional, it is highly recommended to specify problem-dependent settings for every function. The settings in *JavaXCSF* are usually loaded from a properties file. The `xcsf.ini` file that comes with the code package is such a properties file and contains a complete list of options—all of them well-documented. Alternatively, the `XCSFConstants.java` class may be accessed directly, since all options are public and static. The following list briefly describes the most important parameters.

maxLearningIterations specifies the number of iterations as an integer.

maxPopSize specifies the maximal population size as an integer.

epsilon_0 specifies the target error as a double.

conditionType specifies the condition class as a fully qualified binary name.

predictionType specifies the prediction class as a fully qualified binary name.

A “fully qualified binary class name” specifies the complete package hierarchy and the full class name. Consequently a ClassLoader can load the class and new implementations don’t have to change the classifier class itself. As an example, the rotating ellipsoidal condition class is located in the `xcsf.classifier` package and the name is `ConditionRotatingEllipsoid`. The correct binary class name is

```
xcsf.classifier.ConditionRotatingEllipsoid
```

The following section contains the correct binary class names for all implementations delivered with this code package.

3 Available Condition and Prediction Classes

Various condition and prediction types are discussed in the literature and some of them are contained in the code base. Two n-dimensional geometric shapes are implemented, namely hyperrectangles and hyperellipsoids, of which also rotating versions are available (Butz et al., 2008; Stalph, Butz, Goldberg, & Llorà, 2009). For the predictions, the simplest implementation is a constant prediction using the Widrow-Hoff rule for updates (Wilson, 2002). The linear predictor implements the recursive least squares algorithm, which was found to yield good approximations fast (Drugowitsch & Barry, 2008; Lanzi, Loiacono, Wilson, & Goldberg, 2006). Furthermore, a quadratic recursive least squares implementation with mixed terms is included. Thus, oblique quadratic subspaces can be approximated accurately. However, the prediction complexity also grows quadratically in the number of dimensions for this predictor (Lanzi, Loiacono, Wilson, & Goldberg, 2005). Note that all condition classes implement the `Condition.java` interface, while prediction classes implement the `Prediction.java` interface.

In order to use different condition or prediction types, their fully qualified binary class name has to be specified in XCSF's settings file or in the `XCSFConstants.java` class. The following table contains the binary class names required to use these conditions and predictions. However, if the code is put in a different package, the names have to be adapted accordingly.

condition type	**fully qualified binary class name**
rectangles	`xcsf.classifier.ConditionRectangle`
ellipsoids	`xcsf.classifier.ConditionEllipsoid`
rotating rectangles	`xcsf.classifier.ConditionRotatingRectangle`
rotating ellipsoids	`xcsf.classifier.ConditionRotatingEllipsoid`

prediction type	**fully qualified binary class name**
constant	`xcsf.classifier.PredictionConstant`
linear RLS	`xcsf.classifier.PredictionLinearRLS`
quadratic RLS	`xcsf.classifier.PredictionQuadraticRLS`

4 Available Listener Implementations

As mentioned before, the listener concept realizes the observer design pattern to clearly distinguish between data processing and visualization or logging. All listener classes implement the `XCSFListener.java` interface and can be found in the `xcsf.listener` package. These classes can be registered with the XCSF instance in order to get informations about XCSF's progress and current state after each iteration. Furthermore, they are notified about new experiments, if more than one experiment is beeing run.

Various implementations are provided, including visualizations for the condition structure in two- and three-dimensional input spaces and a visualization for the predicted function surface for two-dimensional input spaces. Furthermore, the package contains classes that store information about the experiments to the filesystem, including XCSF's performance and resulting populations in a flat text format. The following sections summarize the functionality of the included classes.

4.1 Visualizations

PerformanceGUI shows XCSF's performance in several graphs. The user can select the values that should be visualized using the button at the bottom right. The class extends `javax.swing.JPanel` and thus can be embedded into other graphical user interfaces. However, the constructor can also be instructed to create an individual window for this visualization.

ConditionsGUI2D3D realizes the visualization of two- and three-dimensional condition structures. Conditions are colored according to their fitness, where darker color indicates higher fitness. If the matchset is shown, the matching conditions are colored differently and a red cross indicates the current input.

PredictionPlot uses gnuplot to visualize the predicted function surface on two-dimensional input spaces.

PredictionErrorPlot uses gnuplot to visualize the average prediction error of the classifiers on two-dimensional input spaces.

ProgressGUI is a small `javax.swing.JPanel` that shows a progress bar, indicating when XCSF will finish the current run. This is especially handy, when running XCSF from other graphical user interfaces and the progress cannot be monitored on a console.

4.2 Logging and Storage

PerformanceWriter writes the XCSF's performance (mean and sample standard deviation over the experiments) to a tab-separated file. Furthermore, if gnuplot is installed, a default plot including prediction error, population size, and generality is created. Only the top bar for the sample standard deviation is shown due to the log-scaling.

PopulationWriter writes XCSF's population to a flat text file. These files can be parsed by the `Population.java` class. This also works with new condition or prediction implementations, if the required constructor is specified. For details, see Section 6.

OutputWriter2D writes the final condition structure and predicted function surface to the filesystem, if the input space is two-dimensional.

5 Structure of the Source Code

JavaXCSF comes with four packages, of which two packages are optional. The `xcsf.listener` package contains mostly visualization and performance logging classes and `xcsf.examples` contains various examplary function implementations and a sophisticated main method to run experiments on these functions. In the following, a brief description of all packages and short descriptions of the most important classes are given. Detailed descriptions of individual classes can be found in the generated JavaDoc API.

xcsf The core package contains the functionality of the Learning Classifier System.

- `XCSF.java` represents the Learning Classifier System.
- `XCSFConstants.java` contains the settings and methods to load settings from a file.
- `Function.java` is the general interface for all learning problems.
- `StateDescriptor.java` is used to abstract from input and output of functions.
- `XCSFListener.java` is the general interface that realizes the observer design pattern. This is especially useful for visualization and logging purposes without changing the core functionality.
- `Population.java` is an efficient, specialized ArrayList-like implementation to hold the set of classifiers.

xcsf.classifier This mandatory package contains all classes linked to one specific classifier, also including condition and prediction functionality.

- `Classifier.java` represents one classifier of XCSF. A classifier consists of a prediction and a condition.
- `Condition.java` is the general interface for conditions. Some implementations of this interface are also contained in the package.
- `Prediction.java` is the interface for prediction classes. Some implementations are containted in the package.

xcsf.listener This optional package contains various implementations of the `XCSFListener` interface, which is used for visualization and logging purposes.

xcsf.examples The optional package depends on the `xcsf.listener` package and includes eleven test functions and a sophisticated main method to run experiments on these functions. Furthermore, an example for distributed online learning with parallel threads is included.

6 Extending the Code

The structure of the source code allows for easy extensions with respect to new functions, condition and prediction structures, and listeners—four interfaces allow for a straight-forward implementation with minimal functionality requirements. The following sections explain some of the details that implementations have to take care of.

6.1 Implementing Functions

`Function.java` is the general interface for XCSF that works in conjunction with the `StateDescriptor.java` class. The most important method, that is `Function.nextProblemInstance()`, produces the function samples for the Learning Classifier System. In each iteration of XCSF, this is the first method call that either samples a random input and computes the corresponding output, or just loads a sample from a file. The resulting input vector $\vec{x}$ and output vector $\vec{y} = f(\vec{x})$ are stored in a `StateDescriptor` object and handed over to XCSF. The `xcsf.examples` package contains eleven implementations of this method. There is one important thing to notice: the default conditions delivered with *JavaXCSF* expect the input space to be confined to $[0, 1]^n$, because the center of the conditions cannot lie outside this space. Furthermore, the restricted search space simplifies calculations, because the search space volume is always one.

Convenience Classes. Besides the standard interface, there are two abstract convenience classes. Functions with one-dimensional output can extend the `SimpleFunction.java` class, which implements a uniformly sampled n-dimensional function with gaussian noise. When XCSF shall be used in a true online environment, where samples are produced by a different thread, the communication has to be synchronized.

The `BlockingFunction.java` class realizes a simple sychronized producer consumer sheme, where the producing thread adds samples to the function object, while the XCSF thread tries to pull samples from the function. If no samples are available, the XCSF thread is put to sleep—if a new sample is added, the thread is notified. If `Integer.MAX_VALUE` samples are in the current queue, the producing thread is blocked, when adding another sample, until XCSF processes samples.

Separated Inputs. The `StateDescriptor.java` class enables the separation of condition and prediction inputs. Usually, the input for the condition is also used for the predictions. However, when the conditions are used to structure the context of the function to be learned, not the input space itself, one state consists of three vectors: the condition input, the prediction input, and the function value. For example, this technique is used to learn the kinematic forward model of a multi-joint arm, where the conditions structure the joint-angle space, while the predicted function maps joint-angle changes to hand location changes (Butz & Herbort, 2008; Butz, Pedersen, & Stalph, 2009; Stalph, Butz, & Pedersen, 2009).

6.2 Novel Conditions

The general interface for conditions is `Condition.java`. There is no explicit representation of the shape of a condition—two methods specify the shape indirectly: `doesMatch(double[])` and `getActivity(double[])`. In the current code, conditions are required to have a center and a volume, although this might hinder the implementation of more abstract conditions, such as gene expression programming conditions. Furthermore, the generality of conditions has to be comparable in order to allow for subsumption. Three methods are dedicated to the evolutionary algorithm for reproduction, crossover, and mutation. For new implementations, it is a good idea to compare the code with the available implementations. Finally, the binary class name specified in XCSF's settings is the only thing that has to be changed in order to use the new implementation.

Constructors. There is no way in Java to force a specific constructor signature with an interface. However, in order to be compatible, implementations have to specify one constructor for covering with an explicitly given signature. During covering, a constructor that takes a double array as an argument is required. If this constructor is not specified, the `Classifier.java` class will throw an exception during covering.

For storing and parsing populations, another constructor is required. This constructor takes a string array as its only argument, that is the decomposed string that represents the condition. For implementation details refer to the methods

- `Condition.write(PrintStream, CharSequence)` and
- `Classifier.parse(String, String, String)`.

The second constructor is only necessary, if parsing of populations is used.

6.3 New Predictions

`Prediction.java` is the interface for any predictor implementation. Analogously to the conditions, there is no explicit representation of the function. The `predict(double[])` method receives the current input vector and returns the predicted function value(s). Predictions are trained by means of the `update(double[], double[])` method, which takes the prediction input vector and the actual function value as arguments. Two methods are dedicated to evolution, namely for reproduction and crossover. However, the crossover method does not refer to standard crossover operators, but notifies the prediction about the new context. Analogous to the condition interface, implementations have to specify at least one constructor for covering, where the following signature is required: `(int, double[])`. Optionally, a constructor with the signature `(String[])` provides functionality for parsing predictions from a file.

6.4 Writing Listeners

Implementations of the `XCSFListener.java` interface can be registered with the XCSF instance using the `XCSF.addListener(XCSFListener)` method. Registered classes are informed about the current state after each iteration, including the current iteration, function sample, population, matchset, and performance. This is done via the `stateChanged(...)` method. Some listeners may be interested in the index of the current experiment or the name of the function to be learned. This information is provided by the `nextExperiment(int, String)` method, which is called once before each experiment. Various implementations of the interface can be found in the `xcsf.listener` package.

7 Support for Multicore Architectures

The most time consuming part of XCSF (or XCS in general) is the matching procedure, since the whole population has to be scanned for matching classifiers. More elaborate conditions may realize complex geometric shapes, further increasing the computational time needed to compute the matchset. In order to adapt to the trend of multicore CPU's, we implemented a parallelized adaptive matching procedure that creates n threads, where

```
n = Runtime.getRuntime().availableProcessors()
```

is the number of processors available to the Java virtual machine. Since there is a considerable overhead for synchonization in parallelized computations, the parallel matching is slower for small population sizes, but for infinitely large populations the speedup by means of parallelization converges to n. The question is, when to start using parallel matching. There is no simple answer to this question, since it depends on (1) the current architecture, (2) current work load, (3) number of available processors, (4) condition complexity, and (5) population size. However, a simple adaptation mechanism, which produces almost no overhead, can be used to estimate a good threshold for each run anew. This way, the usual serial matching is used for small population sizes and the parallel matching for large populations, consequently sqeezing the approximately best performance out of the current system. Furthermore, the technique assures that the speedup is never significantly smaller than one. The following paragraph describes the adaptation sheme.

Automatic Adaptation to Arbitrary Architectures Given an unknown problem on an unknown architecture, the only way to find out when parallelization pays off is to actually measure the times for serial and parallel code. However, comparing the matching time whenever the population size changes results in a huge overhead, until the appropriate threshold is found. In the present implementation, the measurements are not compared directly on the same population size, but on consecutive iterations—thus each matchset is computed only once. Furthermore, the measurement is done maximally 25 times, while the population size increases, and not whenever the population size changed.

The algorithm starts with serial matching, when the population is empty. If the population size increased more than $N/25$, where N is the maximum population size, the time of the serial matching method is measured using `System.nanoTime()`. At the following iteration, the parallel matching method is used and its CPU time is measured. Note that the population size might be slightly higher, but the measurement is not very accurate, anyways. Given the CPU time spent for serial and parallel matching, the times are compared. If the parallelized version is faster, the current population size serves as the fixed threshold for parallelization and the adaptation mechanism is turned off. Whenever the population size succeeds the threshold, the parallelized matching is used and the serial version otherwise. For the next experiment on a possibly different function, the adaptation starts anew. Due to infrequent and noisy measurements the threshold is an approximation. However, the differences in computational time close to the threshold are marginal.

Reproducable Results It is important to notice that the results are not exactly reproducable with parallelized matching, even if a fixed seed for the random number generator is used. This is due to the concurrent entry of classifiers into the matchset: the ordering of classifiers in the matchset cannot be guaranteed to be the same as for serial matching. Although the same classifiers participate in the matchset as for serial matching, the following selection mechanism of the steady state genetic algorithm selects classifiers by their array index. Consequently different classifiers may be selected, when parallel matching is used.

If 100% reproducable results are required (e.g. for debugging), there are two workarounds for this. Either the multi-threading is turned off by setting the corresponding flag in the settings to `false` or the matchset is sorted after matching.

8 Final Comments

The code is distributed for academic purposes with absolutely no warranty of any kind, either expressed or implied, to the extent permitted by applicable state law. We are not responsible for any damage from its proper or improper use.

Feel free to use, modify and distribute the code with an appropriate acknowledgment of the source, but in all resulting publications please include the following citation:

> P.O. Stalph & M.V. Butz (2009), *Documentation of JavaXCSF* (COBOSLAB Report Y2009N001). Retrieved from University of Würzburg, Cognitive Bodyspaces: Learning and Behavior website: `http://www.uni-wuerzburg.de/fileadmin/ext00209/user_upload/Publications/2009/Stalph09JavaXCSF.pdf`

Please report any bugs or other inconsistencies in the source code to one of the authors.

References

Butz, M. V., & Herbort, O. (2008). Context-dependent predictions and cognitive arm control with XCSF. In *Gecco '08: Proceedings of the 10th annual conference on genetic and evolutionary computation* (pp. 1357–1364). New York, NY, USA: ACM.

Butz, M. V., Lanzi, P. L., & Wilson, S. W. (2008). Function approximation with XCS: Hyperellipsoidal conditions, recursive least squares, and compaction. *IEEE Transactions on Evolutionary Computation*, *12*, 355-376.

Butz, M. V., Pedersen, G. K., & Stalph, P. O. (2009). Learning sensorimotor control structures with XCSF: Redundancy exploitation and dynamic control. In *Gecco '09: Proceedings of the 11th annual conference on genetic and evolutionary computation* (pp. 1171–1178).

Drugowitsch, J., & Barry, A. (2008). A formal framework and extensions for function approximation in learning classifier systems. *Machine Learning*, *70*, 45-88.

Holland, J. H. (1992). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* Cambridge, Massachusetts: The MIT Press.

Lanzi, P. L., Loiacono, D., Wilson, S. W., & Goldberg, D. E. (2005). Extending XCSF beyond linear approximation. In *Gecco '05: Proceedings of the 2005 conference on genetic and evolutionary computation* (p. 1827-1834).

Lanzi, P. L., Loiacono, D., Wilson, S. W., & Goldberg, D. E. (2006). 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* (pp. 1505–1512). New York, NY, USA: ACM.

Stalph, P. O., Butz, M. V., Goldberg, D. E., & Llorà, X. (2009). On the scalability of XCS(F). In *Gecco '09: Proceedings of the 11th annual conference on genetic and evolutionary computation* (pp. 1315–1322). New York, NY, USA: ACM.

Stalph, P. O., Butz, M. V., & Pedersen, G. K. (2009). Controlling a four degree of freedom arm in 3D using the XCSF learning classifier system. In *Proceedings of the 32nd german conference on artificial intelligence (ki-2009).*

Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, *3*(2), 149-175.

Wilson, S. W. (1998). Generalization in the XCS classifier system. *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 665–674.

Wilson, S. W. (2002). Classifiers that approximate functions. *Natural Computing*, *1*, 211-234.