Missouri Estimation of Distribution Algorithms Laboratory

Documentation of XCSF-Ellipsoids Java plus Visualization

Patrick O. Stalph & Martin V. Butz

MEDAL Report No. 2008008

July 2008

Abstract

This report gives an overview of the *XCSF-Ellipsoids Java* code. The document explains where to get the code and how to use it. Furthermore, the settings and features are briefly described. *XCSF-Ellipsoids Java* is an XCSF learning classifier system implementation using hyperellipsoidal conditions and recursive least squares predictions for function approximation. The code can be used to evaluate XCSF on several test functions with online visualization support for performance, prediction, and conditions. Other test functions or approximation problems can be easily implemented.

Keywords

Learning classifier systems, LCS, XCS, XCSF, Java, implementation, function approximation, evolutionary computation, visualization.

Missouri Estimation of Distribution Algorithms Laboratory (MEDAL)
Department of Mathematics and Computer Science
University of Missouri–St. Louis
One University Blvd., St. Louis, MO 63121
E-mail: medal@cs.umsl.edu
WWW: http://medal.cs.umsl.edu/

Documentation of XCSF-Ellipsoids Java plus Visualization

Patrick O. Stalph & Martin V. Butz
Dept. of Psychology (Cognitive Psychology III)
University of Würzburg
Röntgenring 11, Würzburg, Germany 97070
stalph@informatik.uni-wuerzburg.de & butz@psychologie.uni-wuerzburg.de

Abstract

This report gives an overview of the *XCSF-Ellipsoids Java* code. The document explains where to get the code and how to use it. Furthermore, the settings and features are briefly described. *XCSF-Ellipsoids Java* is an XCSF learning classifier system implementation using hyperellipsoidal conditions and recursive least squares predictions for function approximation. The code can be used to evaluate XCSF on several test functions with online visualization support for performance, prediction, and conditions. Other test functions or approximation problems can be easily implemented.

Keywords: Learning classifier systems, LCS, XCS, XCSF, Java, implementation, function approximation, evolutionary computation, visualization.

1 Introduction

When someone wants to use Learning Classifier Systems (LCS) for the first time, there is one big hurdle: the correct implementation of such an LCS. The *XCSF-Ellipsoids Java* code described here is an implementation of XCSF (Wilson, 2000, 2001, 2002), which is an XCS (Wilson, 1995, 1998) variant for function approximation. The implementation includes some of the relevant, up-to-date techniques as well as many optional settings. It is possible to implement other function problems and exploit the learned classifier population for knowledge extraction, etc. The programming paradigm is mostly object-oriented but sometimes sloppy for performance reasons.

XCSF-Ellipsoids Java comes with several features, including hyperellipsoidal conditions (Butz, 2005; Butz, Lanzi, & Wilson, 2006, 2008), recursive least squares predictions (Lanzi, Loiacono, Wilson, & Goldberg, 2006; Butz et al., 2008), subsumption, and the greedy compaction algorithm (Butz et al., 2008). Additionally, it is possible to separately define the input for condition and prediction. Moreover, the user can activate three visualization tools to support development and facilitate online performance tracking. The performance visualization shows several graphs (e.g. error, population size). Condition visualization is available for 2D and 3D ellipsoids. The prediction visualization uses *GnuPlot* to show the current prediction surface.

The remainder of this report is organized as follows. Section 2 briefly describes where to get *XCSF-Ellipsoids Java* and how to compile and run the code. Furthermore the features of the XCSF implementation and the settings are explained as well as available functions. The visualization plugins are described in Section 3. In Section 4 the implementation of new functions and listeners is explained.

2 Getting Started

XCSF-Ellipsoids Java can be downloaded from:

`http://medal.cs.umsl.edu/files/XCSF_Ellipsoids_Java.zip`

The package includes the source code (Java 1.5), corresponding JavaDoc, an executable JAR package and three files to specify the XCSF setup. Java Runtime Environment (JRE) 1.5 or higher is necessary to run the code without visualization. To utilize all features (that is, prediction visualization, 3D condition visualization, performance- and prediction plots), *GnuPlot* and *Java3D* is needed.

Java	`http://java.com/`
	`http://java.sun.com/`
Java3D	`https://java3d.dev.java.net/`
GnuPlot	`http://www.gnuplot.info/`

2.1 General XCSF Workflow and Output

When using the executable JAR package or invoking the `XCSF.main(String[])` method, first the following setup steps are executed:

- The initialization files are parsed.
- The functions to be evaluated are loaded.
- Visualization plugins are registered.

Next the experiments are started for each chosen function, that is, for each experiment the number of desired learning steps are executed. Given a function, one experiment of *XCSF-Ellipsoids Java* works roughly as follows (in pseudocode).

```
repeat until termination criterion is met {
  state = function.nextProblemInstance()
  create the match set
  ensure coverage of state
  generate the prediction for state
  evaluate the performance
  update the classifiers in match set
  evolve the classifiers in match set
}
```

When learning has finished, XCSF creates several files depending on two flags in the `xcsf.ini`, namely `doWriteOutput` and `doWritePopulation`. The first one specifies if XCSF should create a performance file, population screenshot, and prediction plot once per experiment. If the latter flag is set, then XCSF writes the classifier population to file after every run.

function-avgPerformance.txt Contains the average (over all experiments) performance for the function. One row contains tab-separated the iteration and other data pairs (mean and standard deviation estimates, respectively).

function-avgPerformance.eps Encapsulated Postscript plot that shows the log-scale average prediction error and number of macro classifiers.

function-avgPerformance.plt *GnuPlot* script to generate the plot.

function-population.png Portable Network Graphics showing the 2D or 3D population. Note that for the 3D case *Java3D* is needed and the screensaver (or blank screen) must be deactivated.

function-prediction.eps Encapsulated Postscript plot that shows the final function approximation.

function-prediction.plt *GnuPlot* script to generate the plot.

function-prediction.dat Contains the data for the plot.

function-pop$\langle i \rangle$.txt After experiment i the final population is written to this file.

2.2 Settings and Initialization Files

There are three initialization files for XCSF. When invoking `XCSF.main(String[])` or the executable JAR package, these files need to be accessible in the working directory. Although the files are well documented, a good XCSF background is necessary to understand some of the parameters.

xcsf.ini contains all settings for XCSF. This includes specific parameters like the number of experiments and simple options like `doWritePopulation`, which specifies if output files should be created after learning.

xcsf_functions.ini specifies the functions to evaluate.

xcsf_visualization.ini specifies the visualization settings and the location of the *GnuPlot* executable can be specified.

The following table explains the parameters and corresponding types contained in `xcsf.ini`.

Parameter	**Type**	**Description**
Experiments and Output		
doWriteOutput	boolean	Specifies, if xcsf writes experimental settings, results and plots to the filesystem.
doWritePopulation	boolean	Specifies, if xcsf writes the population of classifiers to allow for a detailed analysis.
outputFolder	String	Experimental settings, results & plots are stored here; relative or absolute path.
numberOfExperiments	int	Specifies the number of investigated experiments.
averageExploitTrials	int	The number of test instances that should be averaged in the performance evaluation.
initialSeed	long	The initialization of the pseudo random generator. Must be at least one and smaller than 2147483647. Will be used only if "doRandomize" is set to false.
doRandomize	boolean	Specifies if the seed should be randomized (based on the current milliseconds of the computer time).

Parameter	Type	Description
General XCSF Settings		
maxLearningIterations	int	The number of learning iterations in one experiment.
maxPopSize	int	The maximum number of micro classifiers in the population.
alpha	double	The accuracy factor (decrease) in inaccurate classifiers. Default: 1.0
beta	double	The learning rate for updating fitness, prediction error, and set size estimate in xcsf's classifiers. Default: 0.1
eta	double	The learning rate for updating the prediction. Default: 0.1
delta	double	The fraction of the mean fitness of the population below which the fitness of a classifier may be considered in its vote for deletion. Default: 0.1
minConditionStretch	double	The minimum stretch for covering.
coverConditionRange	double	The range of randomization for covering. The maximum stretch is minConditionStretch + coverConditionRange.
Compaction and Matching		
startCompaction	int	The compaction begins at this iteration.
compactionType	int	The compaction type: 0 = condensation and normal matching 1 = condensation and closest classifier matching 2 = condensation, greedy compaction and normal matching 3 = condensation, greedy compaction and closest classifier matching Default: 3
doNumClosestMatch	boolean	Specifies if closest classifier matching is always active. Default: false
numClosestMatch	int	The number of closest classifiers in the match set, if doNumClosestMatch is true. Default: 20
Evolution Parameters		
theta_GA	int	The threshold for the GA application. Default: 50
selectionType	double	Choice of selection type: 0 = proportionate selection]0,1] = tournament selection (set-size proportional) Default: 0.4 (tournament selection)
pM	double	The probability of mutating one allele, often termed mu, in an offspring classifier. Default: 0.05
pX	double	The probability to apply crossover to the offspring, often termed chi. Default: 1.0
theta_del	double	Specifies the threshold over which the fitness of a classifier may be considered in its deletion probability. Default: 20
theta_sub	int	The experience of a classifier required to be a subsumer. Default: 20

Parameter	Type	Description
doGASubsumption	boolean	Specifies if GA subsumption should be executed. Default: true
Classifier Error and Fitness		
nu	double	Specifies the exponent in the power function for the fitness evaluation. Default: 5
epsilon_0	double	The error threshold under which the accuracy of a classifier is set to one. Default: 0.01
predictionErrorReduction	double	The factor (reduction) of the prediction error when generating an offspring classifier. Default: 1.0 (no reduction)
fitnessReduction	double	The factor (reduction) of the fitness when generating an offspring classifier. Default: 0.1
predictionErrorIni	double	The initial prediction error value when generating a new classifier (covering). Default: 0.0
fitnessIni	double	The initial fitness value when generating a new classifier (covering). Default: 0.01
Recursive Least Squares Prediction		
rlsInitScaleFactor	double	The initial diagonal values of the gain matrix. Default: 1000
lambdaRLS	double	Forget rate for RLS. Danger: small values may lead to instabilities! Default: 1.0
resetRLSPredictions-AfterSteps	String	If set, then after the specified number of iterations, all gain matrices are reset according to the initial scale factor. Either an integer is specified or the tag 'startCompaction'. Default: startCompaction
predictionOffsetValue	double	The offset factor that is multiplied with the first coefficient (actually that is the offset) of the prediction. Default: 1

2.3 Available Functions

XCSF-Ellipsoids Java comes with six test functions (see table below). The function input space is limited to the $[0, 1]$ interval for all dimensions. Figure 1 shows plots of the function surfaces using modifier $m = 2$. The CrossedRidge and the SineInSine functions are only available for two dimensional input.

Name	Formula ($n \hat{=}$ dimension, $m \hat{=}$ modifier)
Sine	$\sin(m \cdot \pi \cdot \sum_{i=1}^{n} x_i)$
Sine2	$\sum_{i=1}^{n} \sin(m \cdot \pi \cdot x_i)$
Radial	$\exp(-m \cdot \sum_{i=1}^{n} (x_i - 0.5)^2)$
RadialSine	$\exp(-16 \cdot \sum_{i=1}^{n} (x_i - 0.5)^2) \cdot \cos(m \cdot 2\pi \cdot \sum_{i=1}^{n} (x_i - 0.5)^2)$
SineInSine	$\sin(m \cdot \pi \cdot (x + \sin(\pi \cdot y)))$
CrossedRidge	$\max[\exp(-10(2x-1)^2), \exp(-50(2y-1)^2),$ $1.25 * \exp(-5((2x-1)^2 + (2y-1)^2))]$

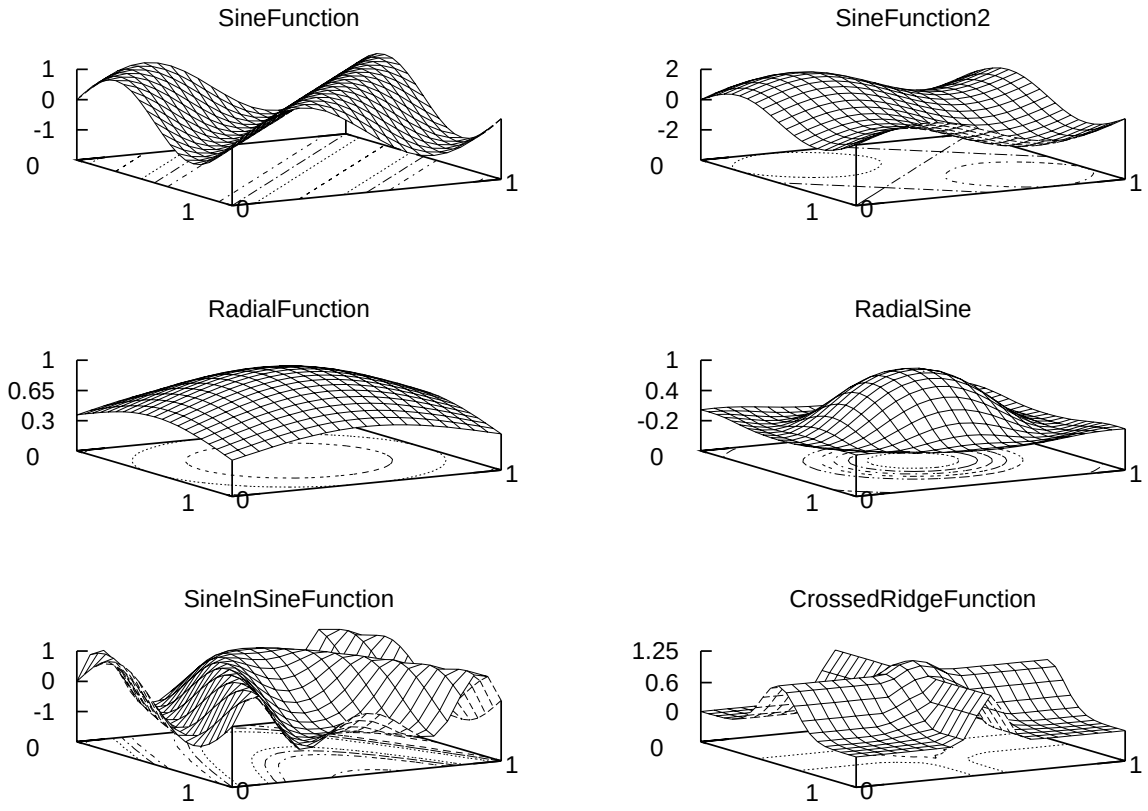

Figure 1: Available functions (plotted with modifier $m = 2$). Note that the function input is limited to [0,1] for every dimension.

3 Visualization

XCSF-Ellipsoids Java comes with three visualization plugins, which can be registered before XCSF is started. If the `doVisualization` flag is set, all three plugins are activated, if possible. XCSF calls their listeners once per iteration. Note that *Java3D* is needed for 3D condition visualization and *GnuPlot* is needed for prediction visualization. Figure 2 shows the plugins at work.

Performance This tool shows several graphs (e.g. average error) concerning performance of the current XCSF run. It is possible to select and deselect the plotted data. Note that the vertical axis is $\log_{10}$ scaled.

Conditions Shows the current ellipsoidal classifier population (2D and 3D only). The population is confined to the $[0; 1]$ interval for each dimension. There are up to four sliders to specify the shown ellipsoidal style:

- *steps* - the number of iterations to evaluate between each visualization frame.
- *delay* - the delay in ms after the frame is painted.
- *size* - the relative size in percent of the shown ellipsoids.
- *transparency* - the transparency of the ellipsoids (not for 3D).

In normal mode all classifier conditions are painted in purple, where darker color indicates higher fitness. Additionally, the *Show Matchset* button can be toggled. If activated, every iteration is visualized and the match set is painted in green (saturated green for high activity). Note that the GUI is sometimes unresponsive, because the main XCSF thread uses as much CPU as possible.

Prediction This plugin shows the current prediction (2D only) if *GnuPlot* is installed. To do this, 21×21 condition inputs are generated, i.e. 5% steps for X and Y axis. A match set is generated for each input and the average prediction is calculated. Now 441 data triples (input x and y values, prediction is the z value) are sent to *GnuPlot* and plotted.

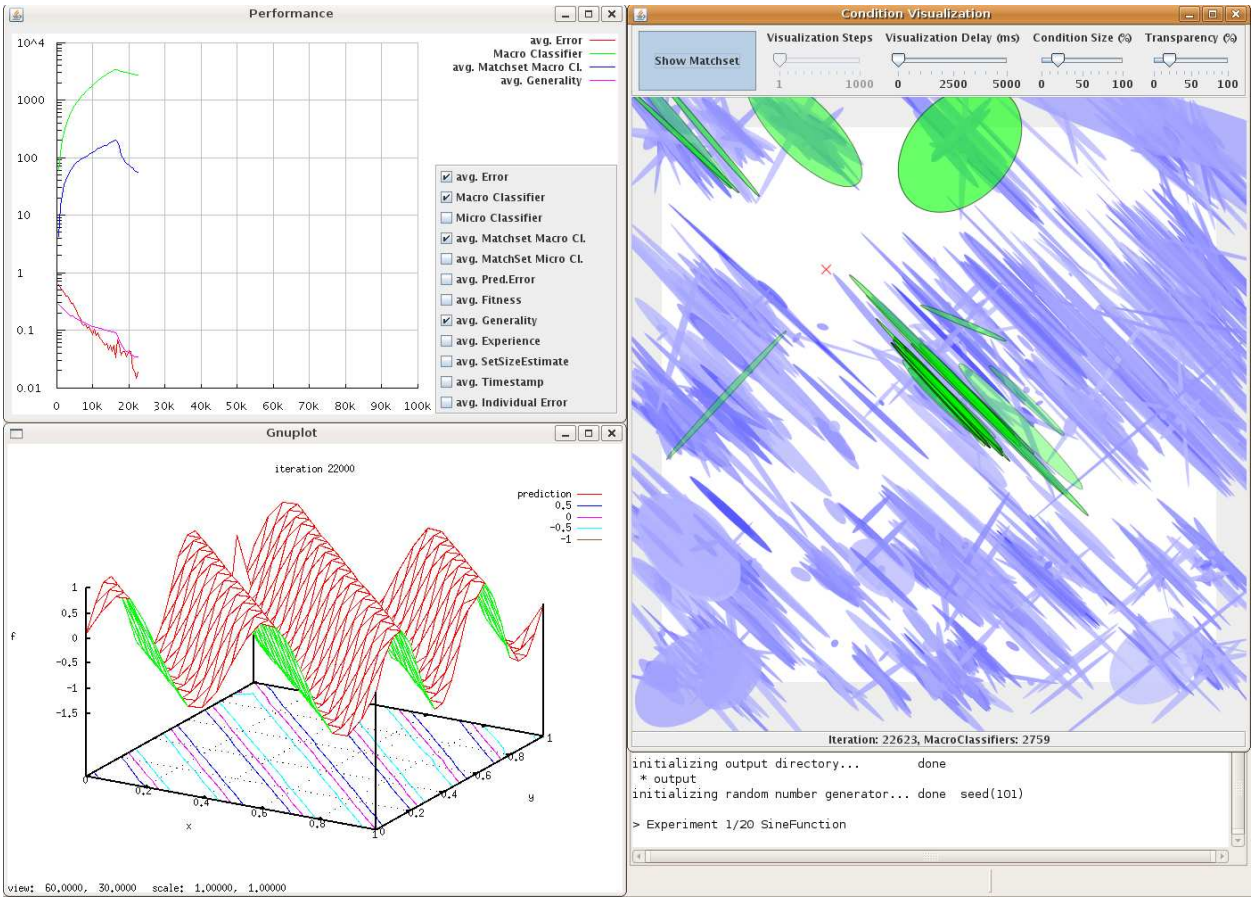

Figure 2: Available visualizations at work: performance (top left), prediction (bottom left), conditions (right).

4 Development

It is possible to implement additional functions or listeners and easily integrate them. For these purposes, the project is divided into two packages: `functions` and `xcsf`, where the latter contains the core classes of *XCSF-Ellipsoids Java* plus two subpackages: (`xcsf.classifier` and `xcsf.visualization`). In Figure 3 shows a simplified UML class diagram.

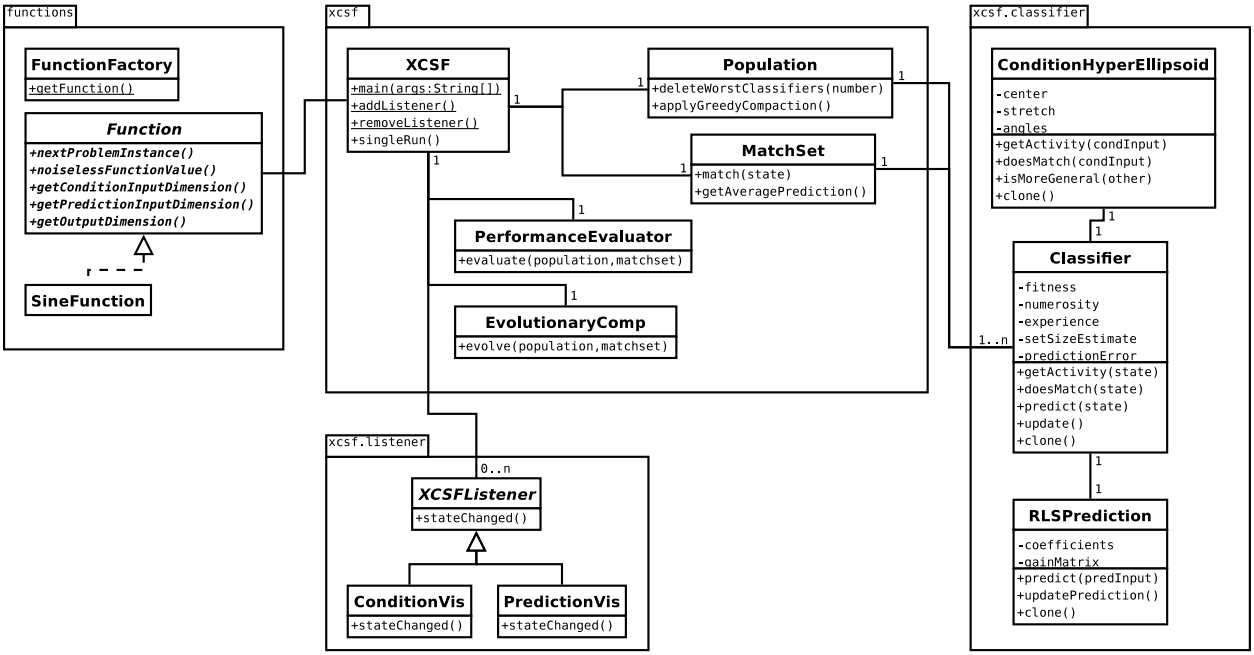

Figure 3: *XCSF-Ellipsoids Java* Package Overview.

4.1 Implementing new Functions

The `functions.Function` interface is used to evaluate a function. To implement additional function problems, the interface of the respective abstract convenience class `functions.Function.SimpleFunction` (n-dimensional functions with 1-dimensional output, support for scaling, modifier, noise) has to be implemented. A very simple way to run XCSF with a new function is described in the `example` package. To access a new function via the `xcsf_functions.ini`), the corresponding name has to be added to the `functions.FunctionFactory` class. The function interface supports the definition of different inputs for conditions (that is, matching) and predictions. Five methods need to be implemented.

```
public StateDescriptor nextProblemInstance();
public double[] noiselessFunctionValue();
public int getConditionInputDimension();
public int getPredictionInputDimension();
public int getOutputDimension();
```

The first one is used to get a function input/output tuple. Usually, a random input is generated (important: the input is confined to the $[0, 1]$ interval) and the corresponding output is calculated. The result is put into the `StateDescriptor` class, which is just a flexible handler for the tuple.

The second method is less important and a return value `null` won't influence the learning process. The value is used for calculation of the prediction error - if `null` is returned, the possibly noisy function value is used instead. The method should be implemented, though, to be able to determine exact prediction errors when noisy test functions are used.

The other three methods determine the length of the condition input, prediction input (if different), and the function output.

4.2 Implementing new Listeners

The `xcsf.XCSFListener` interface is used for visualization purposes only. A listener is registered in the `XCSF` class before the actual XCSF run is started. It is then invoked once per iteration. There is only one method to be implemented.

```
public void stateChanged(int iteration, Classifier[] population,
    Classifier[] matchSet, StateDescriptor state,
    double[][] performance);
```

This method is called once per iteration and contains relevant fields such as the current iteration and the population.

5 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 (2008), *Documentation of XCSF-Ellipsoids Java plus Visualization.* MEDAL Report No. 2008008, Missouri Estimation of Distribution Algorithms Laboratory, University of Missouri in St. Louis, MO.

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

References

Butz, M. V. (2005). Kernel-based, ellipsoidal conditions in the real-valued XCS classifier system. *Genetic and Evolutionary Computation Conference, GECCO 2005*, 1835-1842.

Butz, M. V., Lanzi, P. L., & Wilson, S. W. (2006). Hyper-ellipsoidal conditions in XCS: Rotation, linear approximation, and solution structure. *Genetic and Evolutionary Computation Conference, GECCO 2006*, 1457-1464.

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.

Lanzi, P. L., Loiacono, D., Wilson, S. W., & Goldberg, D. E. (2006). Prediction update algorithms for XCSF: RLS, Kalman filter and gain adaptation. *Genetic and Evolutionary Computation Conference, GECCO 2006*, 1505-1512.

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. (2000). Get real! XCS with continuous-valued inputs. In P. L. Lanzi, W. Stolzmann, & S. W. Wilson (Eds.), *Learning classifier systems: From foundations to applications (lnai 1813)* (pp. 209–219). Berlin Heidelberg: Springer-Verlag.

Wilson, S. W. (2001). Function approximation with a classifier system. *Genetic and Evolutionary Computation Conference, GECCO 2001*, 974-981.

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