Investigating the Learning Classifier Systems XCSR and XCSF

Daniel Sowden

Batchelor of Science in Mathematics and Computing with Honours
The University of Bath
May 2007
This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:
Investigating the Learning Classifier Systems XCSR and XCSF

Submitted by: Daniel Sowden

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:
Abstract

XCS is a Learning Classifier system that produces accurate, maximally general classifiers. There are two extensions to XCS called XCSR and XCSF that can deal with real numbered input values and that can approximate functions respectively. This project aims to implement them both using Java and to run tests on these implementations. Certain design decisions in XCSR will be tested to see performance levels, and the results of this discussed.
Acknowledgements

I would like to thank Dr Alwyn Barry for his continuing support throughout this project, and for all of his guidance and help.

I would also like to thank AJ Bagwell for all of his encouragement, for listening to me and for his opinions throughout the year.

Further thanks are extended to Tom McDowell and Faye Longhurst for moral support.
Contents

Contents ........................................................................................................................1
1 Introduction ...............................................................................................................3

Part I: Background
2 Literature Review .....................................................................................................6
  2.1 Introduction ....................................................................................................6
  2.2 Genetic Algorithms ........................................................................................6
    2.2.1 Background ............................................................................................6
    2.2.2 How Genetic Algorithms work ..............................................................7
    2.2.3 Strength of Genetic Algorithms .............................................................8
  2.3 Learning Classifier Systems ..........................................................................9
    2.3.1 XCS ......................................................................................................9
    2.3.2 XCSR ..................................................................................................12
    2.3.3 XCSF ...................................................................................................13
  2.4 Existing Theory ............................................................................................15
    2.4.1 The Generalisation Hypothesis ............................................................15
    2.4.2 Extensions for Function Approximation ..............................................20

3 Existing Programs ...................................................................................................26
  3.1 The Java Implementation of XCS .................................................................26
    3.1.1 Object Classes ......................................................................................27
    3.1.2 The User Interface ................................................................................28
    3.1.3 The Overall Running of XCS ..............................................................29
  3.2 The Partial Java Implementation of XCSR ....................................................30

Part II: XCSR
4 Design .......................................................................................................................32
  4.1 Requirements Analysis ................................................................................32
  4.2 The Implementation of XCSR .....................................................................33

5 Testing on XCSR .......................................................................................................35
  5.1 Design of Code Modifications ...................................................................35
  5.2 Implementation of Code ............................................................................35
  5.3 Experiments ..................................................................................................35
  5.4 Results and Analysis of Experiment ............................................................36
    5.4.1 Manipulating the Results Data ............................................................36
    5.4.2 Viewing and Analysing the Results Data ............................................36
  5.5 Conclusions ..................................................................................................38
# Part III: XCSF

## 6 Design

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Requirements Specification</td>
<td>40</td>
</tr>
<tr>
<td>6.2 The Implementation of XCSF</td>
<td>40</td>
</tr>
</tbody>
</table>

## 7 Testing

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Approximating the function $y = x^2$</td>
<td>43</td>
</tr>
<tr>
<td>7.2 Approximating the function $y = \sin(x)$</td>
<td>44</td>
</tr>
<tr>
<td>7.3 Analysis of Tests</td>
<td>44</td>
</tr>
</tbody>
</table>

# Part IV: Conclusions

## 8 Conclusions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 XCSR and XCSF</td>
<td>47</td>
</tr>
<tr>
<td>8.2 Further Work</td>
<td>48</td>
</tr>
<tr>
<td>8.3 Concluding Comments</td>
<td>48</td>
</tr>
</tbody>
</table>

## 9 References

# Part V: Appendices

## A XCSR

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mux6Environment Class</td>
<td>53</td>
</tr>
<tr>
<td>Condition Class</td>
<td>55</td>
</tr>
<tr>
<td>Lines 70-75 of original:</td>
<td>55</td>
</tr>
<tr>
<td>Lines 70-76 of the final version:</td>
<td>55</td>
</tr>
</tbody>
</table>

## B XCSR Experiment

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition Class differences</td>
<td>56</td>
</tr>
<tr>
<td>The version of XCSR that uses Boolean wildcards:</td>
<td>57</td>
</tr>
<tr>
<td>The version of XCSR that sets values to the extremes of the interval:</td>
<td>59</td>
</tr>
<tr>
<td>The Perl file</td>
<td>61</td>
</tr>
<tr>
<td>average.pl</td>
<td>61</td>
</tr>
<tr>
<td>The gnuplot file</td>
<td>62</td>
</tr>
<tr>
<td>average.plt</td>
<td>62</td>
</tr>
</tbody>
</table>

## C XCSF

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>WeightVector class</td>
<td>63</td>
</tr>
<tr>
<td>The test of the Weight Vector class</td>
<td>66</td>
</tr>
<tr>
<td>weighttest.java:</td>
<td>66</td>
</tr>
</tbody>
</table>

## D XCSF Tests

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The test Environment</td>
<td>67</td>
</tr>
<tr>
<td>TestEnvironment.java</td>
<td>68</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Learning Classifier Systems are a branch of machine learning that are inspired by processes that occur in the natural world. Learning Classifier Systems attempt to learn from an environment by performing actions, and receiving rewards if the action that they have performed is deemed to be correct or desirable. They build a set of production rules called classifiers which take a condition:action form. The condition part of these rules represents a characteristic which can be matched by inputs from the environment. The action states what action the classifier believes the system should take upon receiving this input from the environment.

Using the reward given by the environment, the system employs a genetic algorithm to attempt to improve the set of conditions. The genetic algorithm works in a way analogous to sexual reproduction in nature; two parents are selected from the population in a ‘survival of the fittest’ manner, whereby more accurate classifiers are more likely to be selected than less accurate ones, then these are ‘mated’, and random mutation may occur.

There is a class of LCS called XCS (Wilson 1995), that was created to attempt to overcome the shortcomings of normal LCSs. In XCS, the fitness of classifiers is judged by their accuracy instead of on the reward itself. It has been shown (Kovacs, 1997) that XCS will develop accurate and maximally general classifiers.

Two further modifications of the XCS have been created. A system that can take real input values instead of the binary inputs traditionally taken by XCS was created called XCSR (Wilson, 2000). This system can be used to work in environments where certain ranges of values lie in one class, and other ranges of values lie in another class. XCSR is particularly useful for working with continuous variables such as temperature, concentration or time values.

The second modification of the XCS is a system that has been adapted to learn from an environment where the input is continuous and the payoff is continuous with respect to the input. This new system was called XCSF (Wilson, 2002). XCSF is useful for approximating functions with no prior knowledge of the function to be approximated.
This project aims to create working implementations of both XCSR and XCSF. Before starting this project, a Java implementation of XCS and a partial implementation of XCSR were provided.

The first part of this report gives a background of genetic algorithms, and of the Learning Classifier Systems XCS, XCSR and XCSF, and it covers some of the existing theory about these systems. It also covers the details of the Java implementations provided; it gives a brief overview of the workings of the XCS implementation, and a summary of the changes made in the XCSR implementation.

The second part covers the completion of the implementation of XCSR. Firstly it covers the design and implementation of this system, and then it covers a test of the difference made to system performance by a slight change in the implementation from the original specification by Wilson (2000). The third part of the report covers the design of the implementation of XCSF, and it covers some testing with the system attempting to approximate some common functions.

The final sections of the report discuss the conclusions made from the project, and there is an appendix at the end containing the key pieces of code. All of the code for the project is on the attached CD.
Part I

Background
Chapter 2

Literature Review

2.1 Introduction

Several key areas need to be understood before this project can be properly undertaken. For the purposes of this literature review, these have been broken down into three major categories, with many subcategories.

The first main category looks at Genetic Algorithms, giving some background on them, outlines how they work, and discusses the strength of them.

The second category looks at Learning Classifier Systems and has three subcategories which look at some major classes of Learning Classifier System; XCS, XCSR and XCSF. In each of these subcategories, the key features of the system are outlined, and in the case of the latter two, the differences from the former one are outlined.

The final category looks at the existing theory about Learning Classifier Systems, outlining the theoretical frameworks that have been developed.

2.2 Genetic Algorithms

2.2.1 Background

Genetic Algorithms (GAs) are search algorithms, so called because their function is based on the process of selection and breeding seen in genetics. It can be seen in nature that there is a ‘survival of the fittest’, where stronger creatures prosper, and weaker ones might not. The fittest offspring are often produced by the fittest parents, and if two parents have different characteristics which are seen as favourable, the offspring may inherit these characteristics from both parents and be stronger than both of its parents. Furthermore, in nature, occasionally mutations may occur, and if these mutations are beneficial, they may be passed down through generations, making the population as a whole stronger in the long run.

In nature, characteristics are defined in a creature’s chromosomes. Chromosomes are made up of genes, which maintain different encoding values called alleles. The
position of each allele within the gene is called its locus. In artificial genetic search, there are sets of strings that are made up of features, characters or detectors which have different values. These features are located at different positions in the string.

GAs were first implemented by John Holland, his colleagues, and his students at the University of Michigan (Holland, 1975) with the aim of designing a computer system that could exploit the processes outlined above. Generations are created in time steps, each made up from parts of the best creatures from the previous time step, and very occasionally, some mutation occurs in a creature, so it will possess a different trait to its parents from the previous time step.

2.2.2 How Genetic Algorithms work

GAs typically take binary strings (although it would be possible to use real numbers or other inputs) of finite length and perform operations on them depending on the fitness value of each string. The strings used are encodings, for example, in the black box switching problem (Goldberg, 1989), the binary strings are made up of five binary values representing whether the switch in the corresponding position on a box is switched in the on position or the off position. The initial population in this case is taken randomly by flipping a coin 5 times to define a string and repeating the process a further three times to create 4 strings.

The GA will then take this population, and use some simple operations to construct new populations which will hopefully improve over time. These operations are much as outlined above in nature: reproduction, crossover and mutation.

Reproduction is the process of deciding which members of the population will be used to construct the new populations. The deciding factor in this stage of the GA is the string’s objective function value. To relate this to the biological analogy, this can be referred to as the string’s fitness, although it can also be thought of as a measure of success or quality of the string.

There are a number of ways in which strings can be selected from the population depending upon their fitness value. One of the easiest ways to do this is to create a biased roulette wheel, where each string is given a pocket, with its size depending on the fitness of the string. It’s easy to picture this as a roulette wheel with pockets of various sizes resembling a pie chart of the fitness of the strings.

Selecting strings in this way, the fitter strings are more likely to be selected for reproduction than the less fit strings, meaning that fit strings will have more offspring in the next generation. Next, a new population called a mating pool is created, where exact copies of the selected strings are placed, and the next step of the genetic algorithm is carried out.

In order to perform one-point crossover, first two strings must be randomly selected from the mating pool. Next, the crossing site must be selected at random. The crossing site is an integer value taken uniformly from the range \([1, l-1]\), where \(l\) is the length of the string. Now, the pair of strings undergo a crossing at this position in the strings. For example, if the strings are of length 5, and the crossing position is selected as 3, the strings are cut after the third character, and crossed over, that is;
\[A_1 = a_1a_2a_3|a_4a_5\]

\[B_1 = b_1b_2b_3|b_4b_5\]

This yields the new strings \(a_1a_2a_3b_4b_5\) and \(b_1b_2b_3a_4a_5\) to put into the population of the next generation. Once all the strings from the mating pool have been mated with another string, the main process of the GA is essentially done.

The remaining operation of the GA is mutation. This operation has only a very small role in the evolution of the strings. The mutation step simply randomly switches a value in a string with very small probability, in much the same way that very rarely, an allele might change in a chromosome in nature. Whilst this plays very much a secondary role in the GA, it can be necessary to ensure that important allele values are not prematurely lost.

This section has described a simple GA, using so-called ‘roulette wheel’ selection, single point crossover and point mutation. More complex forms of GA are possible, varying the form of selection, crossover or mutation, or adding in additional operations so as to vary the rate of convergence and/or diversity in the population.

### 2.2.3 Strength of Genetic Algorithms

The processes behind the reproduction and crossover steps in the GA are very simple, consisting of random selection, copying strings, and partially exchanging strings. Despite this simplicity, GAs can be seen to work very well at optimisation (Goldberg, 1989).

Other optimisation methods often have problems with finding and settling on local optima, rather than finding the optimum value. For example, calculus based methods are very good at reaching a peak, but cannot easily discover if they are at the highest peak in a graph. Clearly GAs can only become better optimisational techniques if they differ from other methods in some way. The main ways in which they do this are outlined below.

Many traditional methods of optimisation work on the actual inputs themselves, whereas GAs work with a coding of the set of inputs. This means that they will not encounter some of the problems that these other methods might come across, for example continuity and existence of derivatives problems that may occur in a calculus based method. The coding can also be manipulated in a different way to how the inputs could be manipulated. However, this may not necessarily be an advantage, as each coding scheme has its own constraints.

Another difference is that GAs search through a population of strings, instead of working with a single point. As mentioned before, working with one point means that it would be very easy to find a local maximum that isn’t necessarily a global maximum. Working with many points means that many peaks can be observed at the same time, which will lower the probability of settling on a false peak. Any other population based method may also provide a similar advantage, however.

GAs also use fitness or payoff information when searching for optima, instead of working with derivatives or other auxiliary information. In some situations, this
simplicity can be an advantage, but when going head to head with methods designed for a specific problem, this may place an unnecessary limit on its effectiveness.

The final major difference is that GAs use probabilistic transition rules instead of a deterministic method. This can add an element of uncertainty that is exhibited in natural situations.

There is much more to be said on GAs, and there has been very much work done on the subject. There are thousands of publications, and the interested reader is directed to see (Mitchell, 1998) for more detail. In this project the GA will be used as one search component in a larger system, and therefore further detail is unnecessary at this stage.

2.3 Learning Classifier Systems

Learning Classifier Systems (LCSs) are machine learning systems that gather information from an environment, and evolve a set of condition-action rules called classifiers. These classifiers take input from the environment, and take actions depending on them. The fitness of classifiers is generally based on their predicted payoff, and they are operated on by a Genetic Algorithm with the aim of improving the system’s overall performance.

2.3.1 XCS

There is a class of classifier system called eXtended Classifier System (XCS) (Wilson, 1995), which has its fitness measure based on the accuracy of its predictions, rather than their strength.

Problems with Basing fitness on Strength

It seems intuitive to base the fitness of classifiers on the strength of the classifier, but there are several problems with this. Some of these problems are detailed below.

Different niches usually have different payoff values, so a single niche with high payoff could possibly take over the population. The solution to this, called ‘fitness sharing’ is that the payoff is shared amongst active classifiers, rather than giving each one the full value.

Although the takeover effect detailed above is now controlled, sharing the payoff will mean that the strength is no longer representative of the payoff; it is the total shared strength of classifiers with the same action that predicts the payoff. This is an issue because a classifier is often part of many different matching sets, which means that the meaning of the strength value will become unclear.

When sharing the payoff, there is still a bias towards profitable niches. This will mean that some earlier classifiers that may normally ‘set up’ later ones may be prematurely lost, meaning that these chains cannot exist. This can be fixed if the GA is run on the niches rather than on the population as a whole. Hence differences between fitness
values of niches will not affect a classifier’s selection chances, and competition will only exist between classifier within a given niche.

There is no way of telling the difference between an overly general classifier and an accurate classifier with only moderate payoff. This means that overly general classifiers may be overly encouraged. Also, there is no reason why payoff based fitness will encourage maximally general classifiers. It seems logical, therefore, to base fitness on accuracy since payoff doesn’t show the difference between accuracy and strength (Kovacs, 2000).

The simple classifier systems tried to get the best rule in niche without knowing about payoff consequences of every action. This raises a problem since incomplete exploration of these actions may lead to a suboptimal solution. It was decided to investigate systems where three new values replace the strength parameter. These parameters are: prediction, the average of the payoff received; prediction error, an average of the error in the prediction parameter; and fitness, an inverse function of the prediction error. The first two would be used in the performance component, and the latter in the genetic algorithm.

Description of XCS

XCS initially seems very similar to traditional LCSs. There are detectors that interact with an environment, producing inputs in the form of a binary encoded message. This is then matched against a population of classifiers, which take a ternary encoded condition:action form, with the string from the alphabet \{0,1,#\}, where # represents a ‘don’t care’ (either 1 or 0). The input from the environment matches the classifier if the condition string from the classifier matches the encoded input value. The strings match if the condition is exactly the same as the input, or if it is the same with one or more characters replaced by a ‘don’t care’.

The classifiers which match the input messages are used to create a Match Set [M]. [M] is a the set of Action Sets that each classifier identifies. It includes an action, the classifiers that have been matched which advocate that action, and the predicted payoff \(P(a_i)\) that will be received when that action is performed, which is the weighted sum of the payoff prediction of each classifier in the action set. An action set \([A]\) is chosen from \([M]\) to perform an action, chosen arbitrarily if exploring to enhance the classifier representation, or chosen by the highest predicted payoff action set if exploiting the learnt classifier representation.

The action given by \([A]\) is sent to effectors and performed in the environment. A reward \(r_{\text{imm}}\) may or may not then be received from the environment. In multi-step environments, the prediction \(p\), the prediction error \(\varepsilon\) and the fitness \(F\) in the action set in the previous time step \([A]_{-1}\) are updated using the Q-Learning (Watkins, 1989) technique. In single step (direct reward) environments, the update is performed from the current action set \([A]\). First, the maximum value \(P(a_i)\) of the prediction array is adjusted by multiplying it by a discounting factor \(\gamma \in [0,1]\), and any value \(r_{\text{imm}}\) is added to it. This result is called \(P\), and it is used to adjust the parameters of the classifiers from the last action set, denoted \(cl_i \in A^{-1}\) in the following sequence;
\[
p_i \leftarrow p_i + \beta (P - p_i) \\
\varepsilon_i \leftarrow \varepsilon_i + \beta (|P - p_i| - \varepsilon_i) \\
\kappa_i = \begin{cases} 
\alpha \left( \frac{\varepsilon_0}{\varepsilon_i} \right)^\nu & \text{if } \varepsilon_i > \varepsilon_0 \\
1 & \text{otherwise}
\end{cases} \\
f_i \leftarrow f_i + \beta \left( \frac{\kappa_i \text{num}_i}{\sum_{j \in A_i \setminus \kappa_i \text{num}_j}} \right)
\]

where \( \beta \in (0,1) \) is the learning rate parameter, \( \kappa_i \) is a temporary classifier accuracy value, \( a \in (0,1) \) is an accuracy parameter, \( \nu \) is a fitness power, \( \varepsilon_0 \) is an error threshold, and \( \text{num}_i \) is the numerosity value, which will be explained later.

Initially, XCS has an empty population. When an input is received that matches no classifiers, XCS performs a process called covering. In covering, a new classifier is created to match this message. In Butz and Wilson (2001), this was done by first checking for every match set if the number of different actions promoted is smaller than the number of possible actions. If this is true, then one new classifier is created for each missing action and inserted into the population. These new classifiers have conditions matching the current input conditions, with some '#'s introduced with fixed probability to allow for generalisation. This will ensure that there is always at least one classifier for each condition:action combination, which will allow a complete payoff map to be created.

Whenever classifiers are created, XCS checks the population to see whether the new classifier has the same condition and action as any existing classifier. If this is the case, the new classifier is not added to the population, but a numerosity field in the existing classifier is incremented. If there is no classifier that matches the new one, then it is simply added to the population, and its numerosity value is initialised as one. These classifiers are called macroclassifiers. To ensure that the system still behaves as though it has \( N \) classifiers, all functions are written to take the numerosities into account where relevant.

When the GA is used, two classifiers are selected from the action set as outlined earlier, with ‘roulette wheel’ selection, point crossover and point mutation. As all classifiers in each action set promote the same action, the new classifiers will also promote this action. The new classifiers are then introduced into the population. If doing this would cause the population to become larger than some threshold value, other classifiers are deleted to create space for the new ones. These classifiers are selected again using a roulette wheel selection, although this time it is not restricted to the one action set.

In a revised paper (Wilson, 1998), two subsumption deletion procedures were introduced to improve the generalisation capabilities of XCS. The first of these is GA subsumption, whereby offspring are checked to see if their conditions are logically subsumed (that is, if it is the same, but with less ‘#’s) by the condition of an accurate and sufficiently experienced parent. If this is the case, the offspring is not inserted into the population, and its parent’s numerosity is increased. The second procedure is
action set subsumption, where an action set is searched for the most general classifier that is both accurate and sufficiently experienced. All of the classifiers in the action set are tested against this general one to see whether they are subsumed by it, and the subsumed classifiers are then eliminated from the population.

2.3.2 XCSR

In some situations continuous variables such as a length, weight, time, or temperature may be decisive in a classification, with one range of variables in one class, and another in another class. A problem with LCSs, and indeed XCS, is that variables take a binary representation, and whilst these could be interpreted as a threshold value, there has been no real adaption for this. A version of XCS was constructed, called ‘XCSR’ (Wilson, 2000), that is XCS for real valued inputs.

Modifications to XCS

XCSR is much the same as XCS, and it is only different in the input interface and the methods of mutation and covering. In XCSR, the conditions of all classifiers are changed from a string in \{0,1,#\} to a union of ‘interval predicates’, \( \text{int}_i = (c_i, s_i) \), where \( c_i \) and \( s_i \) are real numbers. This condition then matches an input \( x \) if and only if \( c_i - s_i \leq x_i < c_i + s_i \) for all \( x_i \). In this way, it may be easier to view \( c_i \) to be the centre in \( \text{int}_i \), and \( s_i \), termed ‘spread’, as a delta value with respect to \( c_i \). A result of using intervals in this way is that there are now twice as many alleles in the condition as components in \( x \). An alternative form of interval exists, which implements lower and upper bounds for conditions (Stone & Bull, 2003b). This technique is also used in XCSF, as seen later.

Despite these changes to the classifiers, crossover works in much the same way as in XCS. A notable effect of this is that crossover can occur at any position in the classifier, that is, it can occur within an interval as well as between predicates. However, mutation does not take a similar form to in XCS. In mutation, an allele is scaled by a random increment \( \pm r \) taken uniformly from the range \( r \in [0, m] \), with \( m \) small (around 0.1) and the sign of the increment chosen randomly.

As in XCS, covering occurs to create a classifier if there is not one which matches \( x \). The new classifier in XCSR has components \( (c_0, s_0, c_1, s_1, \ldots, c_5, s_5) \), where \( c_i = x_i \), and \( s_i = \text{rand}(s_0) \), where \( s_0 \) is a constant such as 0.5. As in XCS, covering occurs for every possible action, and deletion is handled in the same way.

Initial Test

XCSR was initially tested where the input vector \( x = (x_0, x_1, \ldots, x_n) \) had each \( x_i \) in the range \( x_i \in [0,1) \), although there would be no loss of generality if XCSR is implemented over a different input range. Also, the initial problem was tested where the vector was classified into only two cases; 1 and 0. It was constructed so that the class value depended on whether the input values fell into certain ranges of values.
Given these assumptions, the test was performed on the Boolean 6-multiplexer problem. In XCS, the Boolean 6-multiplexer takes a 6 bit condition string and returns 0 or 1. In this condition, the first two bits represent an address of a bit in the remaining bits, for example in 100011, the first two bits, ‘10’, represent the 2nd bit out of the remaining bits (starting at the 0th bit), which is 1. This becomes the function value.

In the real 6-multiplexer, a real vector \( x = (x_0, \ldots, x_5) \) is taken, with each \( x_i \) as above. Using threshold values \( \theta_i \) such that \( \theta_i \in [0,1) \), \( x_i \) is interpreted as 0 if \( x_i < \theta_i \), and 1 otherwise.

When tested (Wilson, 2000), the real 6-multiplexer reached its optimum performance of about 98% in roughly ten times the number of problems that XCS takes with the Boolean 6-multiplexer. There are two obvious sources for this disparity; the classifiers in XCSR have 12 alleles in their conditions compared to just 6 in XCS, and there are in theory infinitely many allele values in XCSR for each dimension, where there are only two in XCS. Despite these intuitive notions, there is no proper analysis into this difference of search time.

### 2.3.3 XCSF

Much as there are situations where variables take real numbers, there may be situations where the desired output takes a continuous quantity, rather than one discrete value. To model a situation like this, it should be possible to adapt XCS to produce real valued outputs and also use it for function approximation. XCSF (Wilson, 2002) is such an adaption, approximating functions of the form \( y = f(x) \), where \( y \) is real, and \( x \) is a vector of integer values \( (x_0, x_1, \ldots, x_n) \).

#### Modifications to XCS

XCSF is also much like XCS but with a few minor changes. The first of these is to allow integer input vectors, rather than binary ones. To do this, the classifier condition was changed again to a union of ‘interval predicates’, \( \text{int}_i = (l_i, u_i) \), where \( l_i \) and \( u_i \) are integers. A classifier matches an input vector \( x \) if and only if \( l_i \leq x_i \leq u_i \) for all \( x_i \). In this way, \( l_i \) can be viewed as a lower bound and \( u_i \) as an upper bound for \( x_i \). Another minor change to XCS was that the program’s payoff predictions were made directly accessible at the output, and the system was restricted to a single (dummy) action.

As in XCSR, crossover in XCSF is very similar to as in XCS. Crossover can take place between any two alleles, or at the ends of the condition (no crossover takes place). Also as in XCSR, mutation occurs by scaling an allele by a random increment \( \pm r \) taken uniformly from the range \( r \in [0, m) \), with \( m \) small and the sign randomly chosen. If after mutation the new lower bound \( l_i \) is less than the minimum input value, which is 0 in the default case, then the new value is set to that value. Also, if the new value is greater than the upper bound \( u_i \), then it is set equal to \( u_i \). A similar rule is implemented for an upper bound upon mutation.

When covering a classifier, the condition of the new classifier has components \( l_0, u_0, \ldots, l_n, u_n \). Each \( l_i \) is set to be \( x_i \) minus a random increment \( r_{0i} \), limited by the minimum
input value, and each $u_i$ is set to $x_i$ plus a random increment $r_0$, limited by the maximum input value. The random value $r_0$ is taken uniformly from the range $r_0 \in [0, m_0]$, with $m_0$ a fixed integer.

Subsumption deletion also occurs in a very similar way to as in XCS. One classifier is subsumed by another if every interval predicate in it is subsumed by the corresponding interval predicate in the other. To say an interval predicate is subsumed by another means that its $l_i$ is greater than or equal to that of the other, and similarly its $u_i$ is less than or equal to that of the other. For action set subsumption, a classifier is more general than another if the sum of the widths $u_i - l_i + 1$ of the interval predicates divided by the maximum possible value of this sum, called its generality, is greater than the generality of the other.

Now to simply approximate a function of the form $y = f(x)$ in XCSF, $x$ can be defined as the input, and $y$ the payoff. So after sufficient time, given an $x$, the system should produce a relatively accurate estimate of $y$. Experimentally, this yields a piecewise constant (step-like) approximation within the given error criterion.

However, it would be better if a piecewise linear approximation was yielded. In order to do this, $x$ is again defined as the input, but now the output is defined, for a one-dimensional function $f(x)$, such as the function $h(x) = w_0 + w_1x_1$. This would yield a straight line approximation for the function. For an n-dimensional function, the output can be defined as $h(x) = w \cdot x'$, where $w$ is a weight vector $(w_0, w_1, ..., w_n)$, and $x'$ is the input vector $(x_0, x_1, ..., x_n)$. This would yield a hyperplane approximation. Now for any given value $x$, the system could calculate the prediction by simply computing $h(x)$.

As the classifiers will now have weight vectors, these will need to be adapted for the function. This is done by using a modified version of the delta rule (Mitchell, 1997). The delta rule is given by $\Delta w_i = \eta(t - o)x_i$, where $o$ is the output, or classifier prediction; $t$ is the target, or the value of $y = f(x)$; and $\eta$ is a correction rate constant. For the case of XCSF, it was decided to use a modified version, given by $\Delta w_i = \left(\eta/|x|^2\right)(t - o)x_i$.

With the exception of the addition of weight vectors, the calculation of predictions and application of the modified delta rule at every time step, no further changes to XCSF are required for a piecewise linear approximation. For classifiers created through covering, the weight vector is initialised with weights taken randomly from the range $[-1,1]$, and in the GA, the offspring created inherit their parents weight vectors.

With these modifications, a function is now approximated accurately, although the number of classifiers required for this approximation is much greater than is needed for a one-dimensional function.
2.4 Existing Theory

2.4.1 The Generalisation Hypothesis

The principles behind the evolution of classifiers in XCS were first outlined in the generalisation hypothesis (Wilson, 1995), which stated that classifiers would be evolved that were maximally general (that is, have as many ‘don’t care’ characters as possible) without losing any accuracy. Theoretical support to this hypothesis was given by Butz et al. (2004). This broke down into two parts; analysing the different evolutionary pressures in XCS, and analysing the tendency towards accurate classifiers in an accuracy-based fitness system.

Evolutionary Pressures

The evolutionary pressures in XCS and the interaction between them were analysed first. Five different evolutionary pressures were analysed; the set pressure, the mutation pressure, the deletion pressure, the subsumption pressure, and the fitness pressure.

Set Pressure

The set pressure in XCS is due to the GA being performed on action sets [A] and deletion is performed on the population [P] as a whole. The set pressure was discussed in the generalisation hypothesis (Wilson, 1995), which said that general classifiers appear more often in action sets, and so are more likely to be proliferated by the GA. This was later extended in Kovacs’ optimality hypothesis (1997), which stated that because of set pressure, XCS can develop a complete, accurate and maximally compact solution for a given problem.

To formalise this pressure, the expected specificity of the classifiers in an action set, denoted s([A]), with respect to that of the population, s([P]) is calculated. The expected specificity measures the average proportion of characters in the conditions of classifiers that aren’t ‘don’t care’ characters.

At the beginning of an experiment, the proportion of ‘don’t care’ characters is dictated by the ‘don’t care’ probability $P_\#$, so the specificity of the population is initially $s([P]) = 1 - P_\#$. To calculate the specificity in the action set [A], the specificity is assumed to be binomially distributed in the population. Now the probability that a given classifier $cl \in [P]$ has specificity $k/L$ is:

$$P(s(cl) = k/L) = \binom{L}{k} s([P])^k (1 - s([P]))^{L-k}$$

where $k$ is the specified (non-‘don’t care’) characters, and $L$ is the length of the classifier conditions. The classifier $cl$ matches an input if all of the specified bits match it, so this has probability $(0.5)^k$. This means that the proportion of matching classifiers with specificity $k/L$ is;
\[ P((s(cl) = k/L) \land (cl \text{ matches})) = P(s(cl) = k/L) \cdot 0.5^k \]
\[ = \cdots = \left( \frac{L}{k} \right) \left( \frac{s([P])}{2} \right)^k \left( 1 - s([P]) \right)^{L-k} \]

In order to find the specificity of a match set \( s([M]) \), the proportion of \( cl \) in \([M]\) with specificity \( k/L \) given \( s([P]) \) must be calculated, that is;

\[ P((s([M]) = k/L) \vert s([P])) = \frac{P((s(cl) = k/L) \land (cl \text{ matches}))}{\sum_{i=0}^{L} P((s(cl) = i/L) \land (cl \text{ matches}))} \]
\[ = \cdots = \left( \frac{L}{k} \right) \left( \frac{s([P])}{2 - s([P])} \right)^k \left( 1 - \frac{s([P])}{2 - s([P])} \right)^{L-k} \]

To compute \( s([M]) \), the actual specificity values \( k/L \), are simply multiplied by the proportions \( P((s([M]) = k/L) \vert s([P])) \), and then summed up. The action set has on average the same specificity as the match set, so \( s([A]) = s([M]) \). So,

\[
s([A]) \approx s([M]) \]
\[ = \sum_{k=0}^{L} \frac{k}{L} P((s([M]) = k/L) \vert s([P])) \]
\[ = \cdots = \frac{s([P])}{2 - s([P])} \]

This equation determines the average specificity \( s([A]) \) in an action set \([A]\) assuming a binomially distributed specificity with mean \( s([P]) \) in the population. This assumption always holds true at the beginning of an environment with a randomly generated population with ‘don’t care’ probability \( P_h \). This equation shows that the specificity of the action set \([A]\) is always less than the specificity of the population \([P]\), with the exception of the end points where it is the same. Since the selection takes place in the action set and deletion is performed on the population as a whole, the pressure is for the specificity of the population to decrease. That is, there is a tendency for the generality of the population to increase.

**Mutation Pressure**

Despite usually having a very small probability of occurring, there is still an evolutionary pressure applied by mutation. The average change in specificity between that of a parent classifier \( s(cl(t)) \) and that of its mutated offspring \( s(cl(t+1)) \) for mutation as described in this report can be written as,

\[
\Delta_mf = s(cl(t+1)) - s(cl(t)) \]
\[ = s(cl(t))(1 - \mu/2) + (1 - s(cl(t)))\mu - s(cl(t)) \]
\[ = 0.5\mu(2 - 3s(cl(t))) \]

Hence, mutation pushes the population towards a specificity of 0.67, and the intensity of this pressure is dependent on the mutation probability \( \mu \), and how regularly the GA is applied.
This change in specificity is expressed as $\Delta_{mf}$ because the mutation discussed in this report is called free mutation, but there is also a method of mutation called niche mutation, which is outlined in Butz & Wilson (2001).

**Deletion Pressure**

Since deletion occurs with probabilities proportional to the size of the action sets, it tends to have an equal distribution between classifiers in different niches. This means that during the deletion process, the selection of classifiers will not result in any set pressure due to this equal distribution, so the specificity of the deleted classifiers will on average be equal to the specificity of the population.

**Subsumption Pressure**

The subsumption deletion mechanism adds an evolutionary pressure to accurate and sufficiently experienced classifiers. GA subsumption deletion blocks offspring that have accurate, experienced, more general parents from being inserted into the population, and this pushes towards maximal generality. Action set subsumption exerts an even stronger pressure than GA subsumption because it eliminates all classifiers from a set that are more specific than an accurate general classifier.

**Fitness Pressure**

The fitness pressure is very much dependent on the problem settings, and as such is difficult to formalise. Generally, it causes pressure pushing the population away from over-general classifiers, towards accurate ones. Broadly speaking, in pushing toward accuracy, it opposes the set pressure pushing towards generality, resulting in more accurate, maximally general classifiers.

**Interaction Between Pressures**

There is an important relationship between the set pressure, the mutation pressure and the deletion pressure. They all influence the average specificity in the population, although they’re all dependent on the fitness pressure, which cannot be formulated. In order to consider the relationship between these pressures, the fitness of all classifiers must be considered constant. When an offspring is created, a classifier with an average specificity $s([A]) + \Delta_{mf}$ is generated. Using this and results gained previously, the average specificity of the population after one time step is:

$$s([P(t+1)]) = s([P(t)]) + f_{ga} \cdot \frac{2(s([A]) + \Delta_{mf} - s([P(t)]))}{N}$$

where $f_{ga}$ is the frequency of a GA application per time step, which depends on $s([P(t)])$ and the specificity distribution within the population. This equation is called the specificity equation.

The tendency in XCS to develop accurate, maximally general classifiers does not depend on the subsumption deletion mechanism, so it has been left out of the theoretical work by Butz et al. (2004), and so it is not used further in this section of the report.

Tests have been carried out (Butz et al., 2004) to validate the influences of the evolutionary pressures outlined above. These tests showed that the specificity...
equation adequately represents the pressures, and the validity of the model was confirmed.

**Evolving Accurate Classifiers**

Whilst fitness influences were revealed in the testing carried out, the fitness cannot be considered explicitly. However, the general conditions that guarantee fitness pressure exists in XCS have been investigated. Whilst there is no formal model for the fitness pressure, the conditions which favour existence of fitness pressure have been analysed.

The first condition that must be satisfied to guarantee fitness pressure is that the classifiers’ fitness must be computable. That is, the parameters must be such that the GA can run, and the classifiers stay in the population long enough that their fitness will be adequately evaluated. The second condition is that when the GA is applied, it must result in adequate pressure towards high fitness classifiers. These conditions are called the covering challenge, and the schema challenge respectively.

**Covering Challenge**

The covering challenge deals with two issues; setting the parameters in XCS so that the GA is applied on fitness values that are actually meaningful and applying the classifiers enough times that the fitness values are actually an adequate measure of fitness.

Covering generally only occurs at the beginning of a run in XCS, but under some circumstances, XCS may enter a loop where covering will continue indefinitely, such a situation is called a cover-delete loop. On receiving an input, XCS builds the match set \( [M] \). If the specificity of the population, which is set at the beginning of the run and determined by \( P_s \), is too high there will be many over-specific classifiers, meaning the input space not well covered. Covering is then applied, and more classifiers will be inserted into the population. At such an early stage, the fitness values have no real meaning, meaning excess classifiers are effectively deleted at random. In this way, classifiers covering other inputs may be deleted to make room for the new covering classifiers. In a similar way, the GA will then also effectively select classifiers at random due to the lack of experience of classifiers. Overall, this causes little fitness pressure among the classifiers in the population. This problem can be fixed by setting \( P_s \) and \( N \) in such a way that the input values have a reasonable probability of being matched.

To formalise this situation, the probability that an input is covered by at least one classifier in a random population, \( P(\text{cover}) \), is determined. This is calculated as a function of specificity in the population, \( s([P]) \);

\[
P(\text{cover}) = 1 - P(\text{no match in } [P]) = 1 - (1 - P(\text{match}))^N = 1 - \left(1 - \left(\frac{2 - s([P])}{2}\right)^L\right)^N
\]

18
where $L$ is the length of the input string, $N$ is the size of the population, $P(\text{match})$ is the probability that a random classifier matches an input, and $P(\text{no match in } [P])$ is the probability that no classifier matches an input.

The covering challenge is overcome if this value $P(\text{cover})$ is large enough that the cover-delete cycle is avoided. Experiments (Butz et al., 2004) have shown that the more complex a problem is, the larger $P_#$ needs to be to avoid this cycle. The challenge can clearly be solved by setting $P_#$ very high, but as it is increased the population will fill with over-general classifiers which cannot yield much information about an optimal solution. In this case, XCS may not develop the pressure needed to obtain accurate classifiers.

**Schema Challenge**

In the schema challenge, analysis of the ways in which the fitness pressure developed by the GA guides the evolution towards accurate classifiers is carried out. In particular, the conditions that can encourage the development of effective fitness pressure are analysed.

The population of classifiers must contain enough specified characters that it is possible for the GA to assemble classifiers that are specific enough to be accurate. It is possible to guarantee that an accurate solution can be reached by setting $P_#$ low enough that adequate accurate classifiers for most niches exist due to covering.

The probability that a niche is represented by at least one classifier in $[P]$ is calculated. In order to do this, a niche is thought of as a schema (Goldberg, 1989) of order $o$ combined with an action. This probability, $P(\text{representative})$ can be calculated as;

$$P(\text{representative}) = 1 - (1 - P(\text{one representative}))^N$$

$$= 1 - \left(1 - \frac{1}{n} \left(\frac{s([P])}{2}\right)^o\right)^N$$

where $P(\text{one representative})$ is the probability that one classifier correctly represents a niche, $n$ represents the number of possible actions, and $o$ is the schema order of the niche.

Using this equation, it can be demonstrated (Butz et al., 2004) that small values of $P_#$ correspond to high values of $P(\text{representative})$, that is, the schema challenge can indeed be met by setting $P_#$ small enough. So that for small $P_#$, XCS can develop an accurate, maximally general solution since accurate classifiers exist in $[P]$.

It is clear that the covering challenge and the schema challenge produce conflicting results. Whilst the covering challenge can be solved by setting $P#$ to be sufficiently high, the schema challenge can be solved by setting $P#$ sufficiently low. As such, the interaction of the two challenges set some boundaries that must be considered when XCS is applied. In Butz et al. (2004), the effect of these challenges was tested on several large Boolean multiplexer problems, and it was shown that the solution exists in situations where the value of $P#$ simultaneously solved both problems.
2.4.2 Extensions for Function Approximation

In XCSF, initially only a piecewise constant approximation of a function was possible, but soon it was possible to approximate first, second and third order polynomials (Wilson, 2002). In addition, the algorithms for function approximation were extended from the initially used Least Mean Square algorithm to the Normalised Least Mean Square algorithm and the Recursive Least Squares algorithm (Drugowitsch and Barry, 2006). This was done in an ad hoc manner, rather than implementing the new algorithms from first principles. As a result of this, Drugowitsch and Barry (2006) described a formalisation for the structure that allowed the optimality of the approximation to be defined, and allowed analysis and extension of the current algorithmic approaches.

The Framework

What follows is a summary of the framework that LCSs use to approximate functions with their classifiers.

The function that is to be approximated is \( V: S \rightarrow \mathbb{R} \), where \( S \) is the state space. This function is sampled or observed with a sampling distribution, \( \pi : S \rightarrow [0,1] \), meaning that the probability of sampling state \( i \) is \( \pi(i) \). This function is applied in discrete time steps \( t=0,1,\ldots \), which gives the sequence of states \( \{i_0, i_1, \ldots\} \), the sequence of function values \( \{V(i_0), V(i_1), \ldots\} \), and the sequence of approximations \( \{\tilde{V}_0, \tilde{V}_1, \ldots\} \), where \( \tilde{V}_i: S \rightarrow \mathbb{R} \) is the approximation when sampling \( V(i_0) \). The approximator also observes a set of features of the current state, \( \{\phi_l: S \rightarrow \mathbb{R}\}_{l=1,\ldots,L} \), where \( \phi_l(i) \) is the \( l \)th feature of state \( i \). The feature vector \( \phi: S \rightarrow \mathbb{R}^L \) returns the features of state \( i \) as a column vector \( \phi(i) = (\phi_1(i), \ldots, \phi_L(i))' \). The approximation \( \tilde{V}_k(i) \) for any matched state \( i \) by classifier \( k \) is given by the dot product of the classifier’s weight vector \( w_k \in \mathbb{R}^L \) and the feature vector \( \phi(i) \) of that state. That is, \( \tilde{V}_k(i) = w_k' \phi(i) \). In XCSF as described in this report, the feature vector \( \phi(i) \) corresponds to the input vector \( x' \).

The function approximator’s aim is to minimise the mean squared error (MSE), defined by 
\[
\int_S \pi(i) \left| V(i) - \tilde{V}(i) \right|^2 \, di.
\]

The squared error is weighted by the sampling distribution \( \pi \), so as to emphasise the error of the states which are visited most frequently and because it naturally represents the approximation target of most standard approximation algorithms.

The aim for every classifier \( k \) is to minimise the \( f_k: \mathbb{R} \rightarrow \mathbb{R}^L \) over its matched states set \( S_k \subseteq S \), the set of states that \( k \) matches. \( f_k \) is given by the equation

\[
f_k(w_k) = \frac{1}{\int_{S_k} \pi(i) \, di} \int_{S_k} \pi(i) \left| V(i) - \tilde{V}_k(i) \right|^2 \, di = E_k \left( \left| V - \tilde{V}_k \right|^2 \right)
\]
where \( E \) is the expectation operator. This term is scaled by the inverse of the probability of observing any of the matching states, which makes the errors comparable between different classifiers. Since it often isn’t possible to observe a function fully, an estimate of the error \( f_k \) is calculated, given by;

\[
e_{k,t} = \frac{1}{c_{k,t}} - L \sum_{m=0}^{t} I_{S_k}(i_m)(V(i_m) - w_{k,t} \varphi(i_m))^2
\]

where the indicator function \( I_{S_k} : S \to \{0,1\} \) is defined as

\[
I_{S_k} = \begin{cases} 
1 & \text{if } i \in S_k \\
0 & \text{otherwise}
\end{cases}
\]

so \( I_{S_k}(i) \) returns 1 if classifier \( k \) matches state \( i \), and 0 otherwise; \( c_{k,t} \) is the experience of \( k \) at time \( t \), or \( c_{k,t} = \sum_{m=0}^{t} I_{S_k}(i_m) \).

**Optimality**

If the function to be approximated \( V \) is fully known, then the error in approximation of a classifier \( k \) is minimal when the features are orthogonal to the approximation error \( f_k \). This occurs when the dot product of the features and the approximation error is zero, that is

\[
\int_{S_k} \pi(i) \varphi(i)(V(i) - w_{k,t} \varphi(i)) di = 0
\]

In this case, the mean squared error of \( k \) is given by

\[
\frac{1}{\int_{S_k} \pi(i) di} \left( \int_{S_k} \pi(i)V(i)^2 di - \int_{S_k} \pi(i) \tilde{V}_k(i)^2 di \right)
\]

If however the function \( V \) is not known, it is approximated using observed values at different time steps, \( \{V(i_0), V(i_1), \ldots\} \). In this case, the approximation estimate \( e_{k,t} \) for the classifier \( k \) must be minimised to find the optimal solution. This occurs when the sequence of feature vectors \( (\varphi(i_0), \varphi(i_1), \ldots) \) is orthogonal to the sequence of approximation errors for the classifier’s matching states \( i_m \in S_k \). That is, when their dot product is zero, so at time \( t \),

\[
\sum_{m=0}^{t} I_{S_k}(i_m) \varphi(i_m)(V(i_m) - \tilde{V}_{k,t}(i_m)) = 0
\]

where \( \tilde{V}_{k,t}(i_m) = w_{k,t} \varphi(i_m) \), and \( w_{k,t} \) is the classifier’s weight vector at time \( t \). In this case, the minimum approximation error of \( k \) at time \( t \) is given by

\[
e_{k,t} = \frac{1}{c_{k,t}} - L \sum_{m=0}^{t} I_{S_k}(i_m)(V(i_m)^2 - \tilde{V}_{k,t}(i_m)^2)
\]
This is called the Principle of Orthogonality.

**Gradient Based Approaches**

There are several gradient based algorithmic implementations of function approximation, which follow the gradient of the error function. These methods are all computationally very easy, and as such are currently the favoured methods for use in machine learning.

**Steepest Gradient Descent**

The steepest gradient descent method performs steps along the steepest gradient towards a local minimum in order to minimise a function. When considering the error function $f_k$, the steepest gradient descent algorithm is defined by $w_{k,t+1} = w_{k,t} - \delta_{k,t}$, where $\delta_{k,t} = \alpha_t \nabla_{w_t} f_k(w_{k,t})$, with $\alpha_t > 0$ the step size, and $\nabla_{w_t} f_k(w_{k,t})$ is the gradient of $f_k$ with respect to $w_{k,t}$.

This algorithm starts at an arbitrary $w_{k,0}$ and reduces the MSE $f_k$ with each step. Moving in this way will lead to the optimal approximation under some assumptions.

There are some criteria for the use of the steepest gradient descent method which are discussed further in Drugowitsch and Barry (2006). The biggest problem with using the steepest gradient descent for function approximation is that it requires knowledge of the gradient $\nabla_{w_t} f_k(w_{k,t})$ for each step. This would mean that full knowledge of the MSE is needed, which in turn requires full knowledge of the function $V$ at all steps. It would be possible to approximate $V$ over a finite number of steps, but there is no guarantee of the accuracy of this approximation.

**Least Mean Square Algorithm**

The least mean square (LMS) algorithm is similar to the steepest gradient descent method, except that it performs a gradient descent on a current local approximation of the gradient rather than on the full gradient of the function. In this way, the LMS applies the same update equation as in the steepest gradient descent, except it takes the gradient of the error at time $t$ instead of the overall gradient. This means that the LMS update is given by the equation $w_{k,t+1} = w_{k,t} + \alpha_t I_{i}(i_t) \phi(i_t)(V(i_t) - w_{k,t} \phi(i_t))$.

The ratio between the difference between $f_k$ and the minimum estimation error of the LMS algorithm, called the excess mean squared estimation error, and the minimum $f_k$ is called the misadjustment. This is a measure of how far the convergence area of LMS is from the optimal approximation of the function. It has been shown (Drugowitsch and Barry, 2006) that the misadjustment is proportional to the step size $\alpha$, and so reducing $\alpha$ will reduce the misadjustment. Conversely, the time constant is inversely proportional to the step size, and so a compromise between a low estimation error and a high convergence rate must be made.

**Normalised LMS Algorithm**

It can be noted in the equation for the LMS update that the magnitude of the update $w_{k,t+1}$ is proportional to the feature vector $\phi(i_t)$. As such, if large values exist in the feature vector the update will be amplified and fluctuations occur. This problem can
by overcome by weighting the update by the square of the Euclidean norm of the feature vector, so it becomes

\[ w_{k,t+1} = w_{k,t} + \alpha J_{S_k}(i_t) \frac{\varphi(i_t)}{\|\varphi(i_t)\|^2} \left( V(i_t) - w'_{k,t} \varphi(i_t) \right) \]

which is the update equation used in XCSF as outlined earlier and in Wilson (2002). The step size \( \alpha \) is also weighted by the inverted squared norm of the feature vector \( \varphi(i_t) \).

The normalised LMS algorithm exhibits a rate of convergence higher than that of the standard LMS algorithm (Douglas, 1994). One drawback of this method however, is that \( \|\varphi(i_t)\|^2 \) must be checked not to be zero in order to avoid dividing by zero. In the case that it is equal to zero, no update occurs as this would mean that the vector \( \varphi(i_t) \) is zero in all elements.

The Kalman Filter and Recursive Least Squares Algorithm

System Model

The Kalman-Bucy system model (Kalman and Bucy, 1961) as outlined in Drugowitsch and Barry (2006) “describes how a noisy process modifies the state of a system, and how this affects the noisy observation of the system”. The process and observation are both assumed linear, and all noise is Gaussian noise with a zero mean, that is the amplitude of the noise is of a normal distribution with mean zero. By assuming that the process is stationary this model can be applied to single classifiers. This gives the model \( V(i_t) = W'_{k,t} \varphi(i_t) + \xi_{k,t} \), where at time \( t \), \( V(i_t) \) is the measurement, \( W_{k,t} \) is the current model of the system state, \( \varphi(i_t) \) is the measurement vector that determines the relation between \( V(i_t) \) and \( W_{k,t} \), and \( \xi_{k,t} \) is the measurement noise which is independent and identically distributed with variance \( \Xi_{k,t} \).

The system state at a time \( t \), \( W_{k,t} \), is modelled by a multivariate normal distribution, with mean \( w_{k,t} \in \mathbb{R}^L \) and covariance matrix \( \Sigma_{k,t} \), that is \( W_{k,t} \sim N\left( w_{k,t}, \Sigma_{k,t} \right) \). The measurement \( V(i_t) \) can also be modelled normally when coupled with the zero mean Gaussian noise, given by \( N(V(i_t), \Xi_{k,t}) \).

Covariance Form

A classifier aims to gain an accurate model of the system state \( W_{k,t} \) and noise variance \( \Xi_{k,t} \), which, given a measurement vector \( \varphi(i_t) \), can be used to predict a measurement. Using the equation for the model introduced above, this prediction is given by the univariate normal distribution centred on \( w'_{k,t} \varphi(i_t) \), with variance \( \Xi_{k,t} \), \( N\left( w'_{k,t} \varphi(i_t), \Xi_{k,t} \right) \). As the expected value for this prediction of the measurement given is equal to the mean, \( w'_{k,t} \varphi(i_t) \), there is a clear relation between this and the approximation \( \hat{V}_t(i) \) in the framework introduced for the linear approximation architecture above.

Using the system state \( W_{k,t-1} \sim N\left( w_{k,t-1}, \Sigma_{k,t-1} \right) \) at time \( t-1 \), the information given by measurements can be included by making the system state \( W_{k,t} \) conditional on those
measurements. The following updates are used for doing this conditional on the measurement $N(V(i_t), \Xi_{k,t})$;

$$
\zeta_{k,t} = I_{S_k}(i_t) \Sigma_{k,t-1} w' \left( I_{S_k}(i_t) \phi(i_t) \Sigma_{k,t-1} \phi(i_t) + \Xi_{k,t} \right)^{-1}
$$

$$
w_{k,t} = w_{k,t-1} + \zeta_{k,t} \left( V(i_t) - w'_{k,t-1} \phi(i_t) \right)
$$

$$
\Sigma_{k,t}^w = \Sigma_{k,t-1}^w - \zeta_{k,t} \phi(i_t)' \Sigma_{k,t-1}^w
$$

This form of Kalman filter update is called the Covariance Form. The difference between the measurement $V(i_t)$ and its estimate $w'_{k,t} \phi(i_t)$ before $V(i_t)$ is known is called the measurement residual. This is weighted by the Kalman gain $\zeta_t$, which determines how much the current estimate is corrected. It is clear that when the noise variance $\Xi_{k,t}$ becomes very small there is a higher weighting, and conversely, when the weight covariance $\Sigma_{k,t}^w$ becomes small, the weighting is much smaller. This effect is intuitive as measurements taken when the noise is low should have larger relevance than those taken in high noise.

**Inverse Covariance Form**

In situations where there is no knowledge of what correct values might be, setting an initial value for $W_{k,t-1}$ may introduce unnecessary bias on the estimate. An approach to the Kalman update that can be used with no information about the initial system state exists operates on the inverse of the covariance, giving the update

$$
\left( \Sigma_{k,t}^w \right)^{-1} = \left( \Sigma_{k,t-1}^w \right)^{-1} + I_{S_k}(i_t) \phi(i_t) \Xi_{k,t}^{-1} \phi(i_t)'
$$

$$
\left( \Sigma_{k,t}^w \right)^{-1} w_{k,t} = \left( \Sigma_{k,t-1}^w \right)^{-1} w_{k,t-1} + I_{S_k}(i_t) \phi(i_t) \Xi_{k,t}^{-1} V(i_t)
$$

so the weight estimate can be found using the equation $w_{k,t} = \left[ \left( \Sigma_{k,t-1}^w \right)^{-1} \right]^{-1} \left[ \left( \Sigma_{k,t}^w \right)^{-1} w_{k,t} \right]$. Matrix inversion only needs to occur when the weight vector needs to be calculated but not for the update itself. If nothing is known about the initial system state, this situation will be preferable since the initial inverse covariance $\left( \Sigma_{k,t-1}^w \right)^{-1}$ will be set to zero to avoid bias, meaning that it won’t be invertible.

**Equivalence to Recursive Least Squares**

The Recursive Least Squares (RLS) algorithm defines a method to accurately track the weight vector that minimises the estimate of the error of the MSE, $\epsilon_{k,t}$ given earlier. It can be proved (Drugowitsch and Barry, 2006) that the Kalman filter is equivalent to the RLS algorithm. An additional result of the proof is that in the RLS update, the assumption is not made that noise is normally distributed. So the Kalman filter can be used to obtain a minimum variance weight with the assumption of Gaussian noise disregarded.

**Estimating Error and Weight Simultaneously**

Up until now, the measurement noise variance $\Xi_{k,t}$ has been assumed known, which is equivalent to the final approximation being known before the weight vector is
estimated. However, this is not the case in LCSs, where the optimal weight vector is found at the same time as the approximation error is estimated.

In the Kalman system model, there are more degrees of freedom introduced than actual measurements. In order to decrease this number, the measurement noise variance is assumed constant over all measurements, $\Xi_k$. The approximation that minimises the model error is sought. This is based on the Covariance Constraint condition, which states that the measurement minus error covariance is equal to the measurement minus truth error covariance, or $I_{\mathcal{S}_i}(i,_{\mathcal{V}i} - w_{\mathcal{S}_i,k}^t\phi(i,))^2 \approx \Xi_k$.

This constraint combined with the assumption that there is no process noise means that the model error at time $t$ can be defined by $\Xi_k^{-1} \sum_{m=0}^t I_{\mathcal{S}_i}(i,_{\mathcal{V}i} - w_{\mathcal{S}_i,k}^t\phi(i,))^2$, which is equal to the approximated MSE $\varepsilon_{k,t}$ scaled by a constant term. Using this equivalence and the Principle of Orthogonality, it can be proved (Drugowitsch and Barry, 2006) that the noise variance can be removed from the error expression to be minimised, and the weight vector tracking conforms to the Principle of Orthogonality, and so is optimal in terms of the mean squared.

Due to this fact and the equivalence of the Kalman filter to the RLS algorithm, the assumption of Gaussian noise is no longer necessary for error and weight estimation. This means that the error and weight updates can be used in situations where noise is not Gaussian, and the minimal variance will still be attained.

Experiments have been carried out (Drugowitsch and Barry, 2006) to compare the rate of convergence for gradient based methods and methods which track the optimal weight and approximation vectors accurately. These experiments demonstrated that gradient based methods have a slower, noisier rate of convergence, and concluded that unless there are spatial or computational restrictions that don’t allow for it, Kalman filter classifiers should be used.
Chapter 3

Existing Programs

For this project, the implementations used are those created by Dr Alwyn Barry. A completed, functional version of XCS and a partial implementation of XCSR were provided.

3.1 The Java Implementation of XCS

The version of XCS on which the implementations of XCSR and XCSF were based was written in java by Dr Alwyn Barry. This implementation uses an object oriented code structure to represent data and to separate the functionality into several classes. There are object classes to represent many different objects; for example for a classifier, a message, an action, a condition, and many others. The details of these are outlined below.

Fig. 3.1 A class diagram of the main classes in the Java XCS system.
3.1.1 Object Classes

The main class in the Java implementation of XCS is the ClassifierSystem class. This is the class that is called by the GUI when a run is initialised. The ClassifierSystem class holds references to the environment that the system will learn within, and the parameters with which the system will run. There is also a reference to the population of classifiers for the current run of the system.

The ClassifierSystem class also holds a reference to the message that has been received from the environment in the current iteration. It also maintains references to all of the action sets in the current iteration, and the selected actions sets used within both the current and the previous iterations. The iterations are maintained by a timing object that is also kept in this class.

Along with these references, ClassifierSystem also keeps references to the current reward and the payoff from the previous iteration, references to the classifiers selected and produced by the genetic algorithm, an object to compile and write out reports for the system, and an agent that can provide roulette wheel selection.

This class has many functions within the system. It is concerned with creating the components necessary for the classifier system, initialising a run of the system, finalising runs, and resetting the system. It deals with updating the prediction, prediction error, accuracy and peer group estimate values of the classifiers within the action set. It also updates the fitness of each classifier within the action set.

As well as these, it also deals with matching the message against classifiers, creating match sets, selecting action sets and sending actions to be performed to the environment. It also deals with getting the reward from the environment and giving this reward to the classifiers. The final thing that this class deals with is that it decides whether the genetic algorithm is due to run, and it can run it. The parents for this action are selected from the action set using a roulette wheel selection agent.

The Environment class as referred to in ClassifierSystem is set up so that it can provide the test environment for the classifiers within the system. It provides methods to set the local parameters, set up the environment for a new run, perform a step, get an input message, give an output message, and it also has many other methods to return information about the state of the system.

The implementation comes with several test problems, including the Boolean 6-multiplexer problem which has been explained in the literature review section of this report. The 6 multiplexer is constructed in the Mux6Environment class, which extends the Environment class.

All of the classifiers in the population are held in the Population class. The methods in this class are involved in the manipulation of the classifiers. There is a method to set up a new population; either an empty one or one made up of some specified classifiers. These are also methods to allow the system to get, locate or check for the presence of specified classifiers, to check for and to locate duplicate classifiers, and to add and remove classifiers.
The Population class also deals with subsumption, allowing the checking of whether classifiers can be subsumed, and also it deals with processes related to macro- and microclassifiers. All of the classifiers in the population are saved as macroclassifiers, but are manipulated to act like microclassifiers.

There is a Classifier class which is used for manipulation of the classifiers in the population. There are methods to create, randomise and set specific values in classifiers, and also to check for equality of classifiers and to copy them. The classifiers’ values and its parameters can be reset, and they can also be generalised with given probability.

As with the Population class, the Classifier class has methods to check for matching and subsumption of classifiers, and classifiers and be crossed over and mutated. The numerosities of classifiers can also be incremented or decremented. From this class, it is also possible to update the peer group estimate, the prediction, the accuracy, and the fitness. The class also has a number of methods to return information about the classifiers parameters.

The SystemPrediction class maintains the match set, and it consists of the action sets of all the classifiers matched in the current iterations.

As discussed in the literature review, classifiers are made up of a condition and an action. These take the form of messages, and there is a class called Message to represent these. The Message class has a bit attribute, and an integer value to represent the length of the message. There are various methods in the Message class to manipulate the messages. Messages can be created, values can be set or randomised, and messages can also be cloned or copied.

The Condition class which holds the information about the condition part of the classifiers is very similar to the Message class, and its methods are very similar and work in a manner analogous to in that class. The only main difference is that there is also a method to set wildcards in the Condition class. The Action class also has some methods very similar to as in both the Condition and Message classes.

3.1.2 The User Interface

This implementation of XCS also came with a user interface package that displays information graphically. This is very convenient for testing, as there are separate windows that can display the populations; listing the classifiers, their prediction, error, accuracy, fitness, numerosity, action set and their experience; also, a window to display and allow easy manipulation of parameters, and several other windows.

The GUI is invoked by running the class XCSTopFrame, which uses threading to execute an instance of XCS. All of the various attributes of the window are added in the XCSTopFrame class, and the separate windows are constructed in separate classes. XCSTopFrame calls the class XCSThread, which creates a thread where the new ClassifierSystem is initialised and the steps occur. Because of this threading, the classifier system can run and at the same time the GUI can be
manipulated. The XCSTopFrame then initialises a new classifier system by initialising a new environment and new parameters.

![Fig 3.2 The GUI for the XCS implementation, including the population window.](image)

### 3.1.3 The Overall Running of XCS

When the system is run, the XCSTopFrame is called, which, as mentioned before, calls the XCSThread which starts a thread. A new ClassifierSystem is initialised, as is an Environment to run the system on and the Parameters for the system to use. XCSTopFrame then iterates through the XCS one step at a time, and once all iterations are completed, it finalises the run.

When the XCSTopThread class initialises the classifier system, the ClassifierSystem class initialises a population, where the Population class creates classifiers and adds them to the population. The ClassifierSystem then initialises the Reports class. The Reports class provides a file that reports a measure of the size of the population, the performance of the XCS, and the system error. Lastly, the ClassifierSystem initialises all of the data to their starting positions; that is the time, the environment, the population, the system prediction etc.

After initialising the classifier system, the XCSTopThread runs steps, where it gets the ClassifierSystem class to firstly get the message input from the environment. Next this message is matched against all of the classifiers in the population. A match set is then created, the best match set is selected, the action is sent to the environment, and then the prediction, accuracy, error and fitness are updated in the given action set.
3.2 The Partial Java Implementation of XCSR

Due to the object oriented nature of the implementation of XCS, it is very easy to change the way that some classes behave without changing too much of the other classes. It would be necessary to ensure that the same methods are utilised within the classes, so that then no further changes would be needed in other classes, although it would still be possible to make larger changes.

In the partial implementation by Dr Barry, the major changes that have been implemented are to the condition class. In a slight change to the original specifications of XCSR as described by Wilson (2000), instead of using a set of interval predicates with centre and spread values, \( int_i = (c_i, s_i) \), a lower and upper bound system, \( int_i = (l_i, u_i) \), had been used, similar to as described in Wilson’s description of XCSF (Wilson, 2002). This meant introducing a new class `Range` inside the `Condition` class. The `Range` class has values for maximum and minimum values, and a Boolean wildcard value. The `Range` class allows setting values, getting values, checking for wildcards, checking for equality, and cloning a range. This is very similar to some of the other classes as outlined above.

The other main changes to the system were made to accommodate this change in the classifiers from binary to real numbers. This meant using array lists to represent classifiers instead of a vector. Aside from one or two other minor changes, this change was the only change made to the `ClassifierSystem`, `SystemPrediction`, `ActionSet`, and `Message` classes.

Due to the change from binary to real numbers, changes also needed to be made to how the system performs mutations and subsumption, as outlined in the literature review. This meant that some functions in the `Action`, `Classifier`, `Condition` and `Population` classes had also been slightly modified.
Part II

XCSR
Chapter 4

Design

4.1 Requirements Analysis

The main required functionality of XCSR is as described in the Literature Review section of this report. As the implementation as started by Dr Barry uses a lower and upper bound method, a so-called ‘ordered bound representation’, of representing classifiers instead of the centre-spread representation as outlined by Wilson (2000), it would be best to stick to that method. It has been shown (Stone & Bull, 2003a) that both of these methods of representing the ranges of real numbered classifiers introduce bias into the system, however an ordered bound representation introduces marginally less than the centre-spread representation.

The partial implementation also uses Boolean wildcards, whereas the original specification (Wilson, 2000) says that wildcard bits should be saved as a ‘bit’ that accepts any value in the range, in this case as a ‘bit’ with minimum value set to 0.0, and maximum set to 1.0. The theoretical advantage of using Boolean wildcards over setting the values of the interval to the extreme values is that if a classifier has an attribute mutated to a wildcard, and then this is later mutated back to an interval, any previous knowledge that this attribute has learnt will not be lost. Clearly, if the values are set to extremes for a wildcard mutation, and then mutated back to a random interval, then this attribute will have lost all previous knowledge about the value range and will be ‘starting from scratch’. It would be best to stick with this method for this implementation, and then after the program is completed, tests should be carried out to compare the performance of this system with one which represents wildcards as specified by Wilson (2000).

The revised method of mutation as implemented by Dr Barry is exactly as specified by Wilson (2000), with an increment or decrement by a specified amount. Also, covering and crossover in XCSR should happen in exactly the same way as in XCS, so no changes should be made to these operations.

The subsumption action should work in the same was as in XCS, except some additional checking will need to be added here to check whether a classifier is more general or not. In XCS, generality was simply a comparison of the wildcard values, whereas in XCSR, the generality is measured as a comparison of both the wildcard values and of the bounds of the ranges.
A test problem for the system should also be implemented. This should be the real 6-multiplexer as outlined by Wilson (2000).

Some minor changes also need to be made to the GUI for XCSR. The classifiers will need to be shown with the upper and lower bounds instead of bit values in the classifier population window. All occurrences of the term ‘XCS’ in the GUI must also be replaced with the term ‘XCSR’.

4.2 The Implementation of XCSR

It is easy to see from the requirements specification that there were two main changes to be made to the partial implementation XCSR. These are that an environment should be constructed, and that the GUI should be modified.

The real 6-multiplexer problem works in much the same way as the Boolean 6-multiplexer; whereas the Boolean 6-multiplexer takes the bit numbers that appear in the classifier, the real 6-multiplexer takes the real numbers from the interval $[0.0, 1.0]$, rounds these numbers up or down and does the same thing. That is, a value in the range $[0.0, 0.5)$ is interpreted as 0.0, and a value in the range $[0.5, 1.0]$ is interpreted as 1.0. This binary value is then used as a reference to a further ‘bit’ to check, which is also interpreted in this way. As such, the implementation of the real 6-multiplexer problem varied very little from that of the Boolean multiplexer; the values were simply rounded and then manipulated in a way analogous to how the Boolean multiplexer worked.

The changes to be made to the GUI were relatively simple to implement. The main change to be made to the GUI was that the classifiers conditions needed to be displayed as a set of intervals rather than as bit strings. These changes needed to be implemented in the Population View window and in the System Prediction View. In the original implementation of XCS, the classifiers were displayed by calling the `getPrediction()` method in the `Classifier` class. This returns a number of bits for the implantation of XCS, however after the implementation of the intervals in the conditions, this will return a number of intervals of type long which will be impractically large to display in the window. This was remedied with a for-loop that scrolled through every interval in the condition and used an if-else condition to print either a ‘#’ or the maximum and minimum values in parentheses rounded to two decimal places. After this, other small changes that needed to be made were to resize the Population Window and the System Prediction window and to spread out the top rows of the tables so that all of the columns aligned with their headings.
Fig. 4.1 A screenshot of the modified population window after a run of XCSR.

A large number of the other changes to the GUI simply involved changing ‘XCS’ to ‘XCSR’ in the code. This was the case for the title of the window, the title of the graph, the title of the parameters window, and the menu item to start a new run. Slightly harder to change was the text in the parameters window, which was aligned to the right. This was done by setting the co-ordinates of the Label, so the ‘left’ coordinate simply needed decreasing by the size of the letter ‘R’.

A small number of other changes needed to be made to the partial implementation to ensure that it would run. These changes were mostly done to fix errors in the partial implementation. One more notable change was made to the **Condition** class in the method that checks for equality.
Chapter 5

Testing on XCSR

As previously mentioned, the implementation for XCSR used Boolean wildcard values as opposed to simply setting the interval value to extreme values. To test the effects of this change, a number of test runs of the real 6 multiplexer problem should be run on this version of XCSR and also one that sets the values to extremes.

5.1 Design of Code Modifications

To set the values of the wildcards to the extremes, the only class that needed to be changed was the Condition class. The major noticeable change to this class is that the Boolean wildcard variable was removed from the Range. Once this was removed, any occurrences of the Boolean wildcard value needed to be changed. For the most part, this simply involved changing the setting of maximum and minimum values to 1.0 and 0.0 respectively or returning the value of the evaluated logical expression \((\text{min} == 0.0) \&\& (\text{max} == 1.0))\). Perhaps the only notable change in functionality was for when a wildcard gets set from true to false. In this case, an if-condition checks if the value was a wildcard beforehand, and if so sets the maximum and minimum values to two random values. These values are then ordered and constrained.

5.2 Implementation of Code

The changes outlined above were made to the implementation of XCSR, along with the window being renamed to “XCSR Viewer (non-Boolean wildcards)” to ensure no confusion arises during the testing of these changes.

5.3 Experiments

A number of experimental runs of the real 6 multiplexer problem were run in each version of XCSR, and the results logged. These experiments were run with the same parameters as used by Wilson (2000) for the initial test problems for XCSR.
5.4 Results and Analysis of Experiment

5.4.1 Manipulating the Results Data

The results of these tests were output by the Reports class as .csv files, or comma separated value files. An example of the start of an output file is as shown below;

```
0,1.000,0.990,0.001
1,0.500,0.499,0.002
2,0.667,0.626,0.003
3,0.500,0.527,0.004
...
```

This is a file that reports the step number on the left, the size of the population on the second, next is the performance of the system (the proportion of non-zero reward), finally followed by the System Error (the absolute difference between the received reward and the reward predicted by the current action set), per exploration all averaged using a 50 point moving average.

As the results of a run naturally fluctuate a lot, these all look very ‘noisy’ when represented graphically. Also, as the system is based on random selection and other such random factors, it is possible to have anomalous test runs. So in order to reduce this noise and to reduce the chances of examining an anomalous run, it is necessary to average the values from several test runs.

To do this averaging, a piece of code was created written in Perl. This program simply put all of the values from the output files into a large matrix, and then row by row it averaged each of the columns from each file, and printed this average value to a new file. This file, unlike the files given by the XCSR system, is a .dat file, with spaces between each value instead of commas. Also, whereas the output of the system is given to three decimal places, the value printed by the Perl code was to done so to four decimal places to minimise the loss of accuracy.

5.4.2 Viewing and Analysing the Results Data

The two .dat files returned by the Perl program were now in a form that could be easily input and viewed in the graphing application gnuplot. Gnuplot is a command-line program that can plot functions and data in 2- or 3-dimensional forms. It takes data files that are typically arranged into columns of data, and by default plots a graph with the first column taken as the values for the $x$-axis and subsequent columns are used for the $y$-axis. A file can be written for gnuplot to instruct it how to treat data and allows customisation of the graphs, like setting the ranges of the axes, setting the graph legend, and how often to put tics on the axes. The results of the averaging files as plotted using gnuplot are shown below.
Fig. 5.1 The average of 5 real 6 multiplexer runs on XCSR with Boolean wildcards.

Fig. 5.2 The average of 5 real 6 multiplexer runs on XCSR with wildcards set to extremes in the interval.
It is clear to see that the population size remains much higher in the XCSR where the wildcards are set to the extremes in the interval than it does in the XCSR where Boolean wildcards are used. Other than this difference, it can be seen that the system error is slightly lower for the XCSR with Boolean wildcards, and the system performance is slightly higher. However, these differences are only marginal.

5.5 Conclusions

It is clear that there appears to be a marginal increase in system performance as a result of using Boolean wildcards, although this is not a substantial increase. The most noticeable difference however is that the system attains a higher population in the tests with wildcards set to the extremes of the range, and then this number drops off at a lower rate than the tests with Boolean wildcards. Without further analytical testing, it would be impossible to state a reason for this, although there is an intuitive notion as to why this change may occur.

As explained in the literature review, when the system runs, classifiers get mutated with a given probability. As mentioned earlier in this section, this means that in the system which does not use Boolean wildcards, any previous knowledge that a classifier might have can be lost with this given probability if its value is mutated to become a wildcard. Similarly, if the classifier is mutated back from a wildcard value, it will have to regain all of the previous experience. This means that random values will be introduced into the population, and as a result of this, it may be the case that more covering has to occur, which may be a reason why this modified version of XCSR attains a higher peak in the population. Because of this factor, it may follow that the population as a whole will suffer in its performance and its error.

It has been shown Kovacs (1997) that XCS generates complete and maximally general maps, and Wilson (2000) provided evidence that XCS can do this for continuous domains, and so despite these slight differences, both of these systems will still create complete mappings.
Part III

XCSF
Chapter 6

Design

6.1 Requirements Specification

As with XCSR, the main required functionality of XCSF is as described in the Literature Review section of this report. As XCSF is designed to work for integer input vectors rather than for binary ones, it should be implemented in a way similar to the implementation of XCSR. The main functionality of XCSF will come from the implementation of XCSR. This means that the ranges will also be represented by maximum and minimum values rather than centre and spread values. This is, unlike with XCSR, as specified by Wilson (2002). Similarly, the wildcards will be represented by Booleans instead of by setting the interval values to the extremes.

As the implementation will be based on the implementation of XCSR, the Condition class should not need any modification. Also, the mutation, crossover and subsumption deletion operations should occur in the same way as in XCSR, and as such should not require any alteration.

The main requirement of XCSF is that it must have a weight vector for each classifier. It must be possible to modify this weight vector using the modified delta rule (Wilson 2002), and to calculate the system prediction using this weight vector. In order to use the modified delta rule, it will be necessary to introduce a variable for $\eta$, the correction rate.

6.2 The Implementation of XCSF

As mentioned, the main requirement of XCSF is that it must maintain a weight vector for each classifier. It was decided that the best way to implement this weight vector was to design a new class for this. This would be integrated into the Classifier class along with the already implemented Condition and Action classes (although the system will be restricted to just one dummy action).

In the new WeightVector class, the main methods to be implemented were to initialise and to set the weight vector, to compute the system prediction, and to update the weights. Also, the ability to return the weight vector as a string would be needed.
In the method employed to initialise the weight vector, passing an integer would initialise an array of that number of random doubles. These doubles were initialised in the range [-1.0, 1.0]. Also, if passing an array of doubles to this method, the weight vector would be initialised using these values for its weights.

The method in the WeightVector class used to compute the prediction simply takes in the input message as a variable, and then multiplies each value in this message raised to the power of its position by the corresponding value in the weight vector, and then sums these values. That is, it takes the input message \((x_0, x_1, x_2 \ldots)\) and the weight vector \((w_0, w_1, w_2 \ldots)\) and calculates the result of the polynomial \(w_0x_0^0 + w_1x_1^1 + w_2x_2^2 + \ldots\). This is slightly different to how Wilson (2002) defined the computed prediction; in his specification, Wilson took the input vector augmented with a constant \(x_0\), whereas this takes the input vector indexed from \(x_0\), and raises the \(x_0\) in it to the power of 0. As a result, this value becomes 1.0, and so it can be viewed as a constant.

The modified delta rule is used in the method updateWeights, where a for-loop simply iterates through every value in the weight vector and substitutes values into the equation as specified by Wilson (2002). In order to do this, the correction rate parameter \(\eta\) is needed. This was simply placed into the Parameters class, and initialised as 1.0. In order to allow easy modification of this value by users in test runs, it was also added to the parameters window in the GUI. Similarly, the mutation increment was also added to this window.

To test the new WeightVector class, a short test program was written. This test program initialises a new weight vector using both of the methods as outlined above, and also set a new message. These were then used to test the methods computePrediction and updateWeights. First a target value was set, next the prediction was calculated and the weight vector was updated using the error calculated, and this process was repeated a number of times. The system could be seen to converge to the target value as expected.

The weight vector class then only needed to be added to the Classifier class to be fully integrated into the system. This meant initialising a weight vector in a similar manner to how conditions and actions were treated throughout the class. As a result of this, a clone method and a copy method needed to be added to the WeightVector class. An updateWeights method was also added to the Classifier class, which will be called from the ClassifierSystem class when the predictions are updated.

The final major change made to the system is that an environment was implemented. The initial test environment that has been implemented is one that produces a linear approximation of the function \(y = x^2\). In order to do this, the environment simply returns a reward of the exact value of the function. The actual error value is calculated in the Classifier class, and the weight vector is updated in the WeightVector class using the modified delta rule.

To add flexibility, the environment was constructed so that all functions can be entered with ranges, and these will be scaled and shifted so that the \(x\) and \(y\) values are both in the range [0.0, 1.0]. To change the approximation back after the system has
run, the inverse of this scaling and shifting needs to be used. This is particularly useful for functions such as the sine and cosine functions, which may be desired to be approximated with $y$ in the range $[-1, 1]$ and $x$ in the range $[0, 2\pi]$. 
Chapter 7

Testing

7.1 Approximating the function $y = x^2$

Once XCSF had been implemented, the system was tested using the environment as described above; creating a linear approximation to the function $y = x^2$. This was done using the same parameters as used by Wilson (2002) in his initial tests. The system was allowed to run for one test, and the results were collected. A small change was made to the GUI to display the weight vector values at the output in the population window.

The population at the end of the 100,000 iterations was then sorted by fitness using the spreadsheet program Microsoft Excel. The top 5 performing classifiers there thus identified and a graph of the lines that they represent was plotted using gnuplot as with the experiments in XCSR. This can be seen below;

![Graphs showing the top performing classifiers in an XCSF approximation of $y = x^2$.](image)

**Fig. 7.1** Top performing classifiers in an XCSF approximation of $y = x^2$

Clearly the system yields an accurate set of classifiers for this problem, which implies that it is running as would be expected.
7.2 Approximating the function $y = \sin(x)$

One of the hardest functions for a system to approximate linearly is a sine curve. This is because the function exhibits a large change in gradient at the angles $x = \pi / 2$, and $x = 3\pi / 2$. XCSF was used to approximate the curve $y = \sin(x)$ using the same parameters as used for the approximation of the curve $y = x^2$. As mentioned in the implementation section of this report, the function was scaled and shifted to ensure that $x$ and $y$ are in the range $[0.0, 1.0]$. As the sine curve was to be approximated with $y$ in the range $[-1.0, 1.0]$ and $x$ in the range $[0, 2\pi]$, this means that the resulting approximation is actually that of the equation $y = (\sin(2\pi x) + 1)/2$.

Below is the graph of this equation alongside the graph created by all of the classifiers in the system at the end of the 100,000 runs.

![Graphs](image)

**Fig. 7.2** All classifiers in a linear approximation of the function $y=\sin(x)$.

7.3 Analysis of Tests

It is clear that the system produces accurate classifiers to approximate the functions $y = x^2$ and $y = \sin(x)$. As expected, for the function $y = x^2$, the system produces very accurate classifiers, and the resulting approximation can be demonstrated with very few of the top performing of these classifiers. In contrast, the approximation for the function $y = \sin(x)$ can only be shown using a large number of classifiers that only cover small intervals. However, for both systems, it is clear that an accurate approximation is yielded.

In the approximation of the sine curve, it is clear that when $x > 3\pi / 2$, or $x > 0.75$ on the graph above, the classifiers exhibit a much more shallow gradient than that of the section of the curve which they are approximating. This is most probably because the system has not run for long enough, meaning that the modified delta rule has not been able to update the weight vectors to get the $y$-intercept value low enough and the gradient value large enough to accurately approximate this section of the curve. If the
system was to run for more runs, then this portion of the approximation should align more with the actual line being approximated.

One other notable feature of the classifiers is that there are very few classifiers that have close to zero gradient; that is classifiers that are approaching being parallel to the \( x \)-axis. In a normal linear approximation of the sine function, one would expect the system to produce few classifiers for the straighter portions of the curve, and many classifiers for the portions with a rapid change in gradient. However, this does not appear to be the case in this run of the system, and indeed in several subsequent test runs.
Part IV

Conclusions
Chapter 8

Conclusions

8.1 XCSR and XCSF

In both of the finished implementations, XCSR and XCSF, it was shown that the systems perform as expected for the real multiplexer problem and for function approximation respectively. Both produce graphs that are of the same shape as those created by runs for Wilson. Also, when the individual classifiers in various XCSF problems are shown graphically, it is clear to see that they do give an accurate linear approximation of functions. From this, we can conclude that the systems are complete working implementations of the systems as outlined by Wilson (2000, 2002).

Whilst the systems both perform as expected for the problems, they still take around twice as long as the systems implemented by Wilson (2000, 2002) to attain a suitably high performance. There are a few reasons as to why this might be the case, but it is important to stress that these reasons are unrelated to any changes to the system that have been made for this project. Two most likely reasons for this difference in the systems are due to implementation choices made by Dr Barry for the original implementation of XCS.

The first of these reasons is because of the function used by Dr Barry to update the accuracy of the classifiers. Wilson (1998) stated that the function to do this in the description of XCS was “the best one so far” of a number which he had experimented with. This function employed an exponential function to perform this update. In the algorithmic description of XCS (Butz and Wilson, 2001), a power function was used for this update. The method used by Dr Barry is slightly different to these two functions. The second of these possible reasons is because of the method used by Dr Barry for subsumption. His method for this also differed slightly from the method outlined in the algorithmic description by Butz and Wilson (2001).

As mentioned earlier, the system does not appear to approximate the sine function as efficiently as would be expected, however there is no obvious reason as to why this could have occurred; the weight vector class is implemented to work in exactly the way specified by Wilson (2002).

Due to the way in which the weight vector was implemented in the XCSF system, it should also be possible to run this system on n-dimensional environment problems.
There is no reason why a test environment could not be implemented to run the system on such a system.

### 8.2 Further Work

There are a number of improvements that could be made to the system. As far as can be seen, there are no major issues with the functionality of the systems, except that some optimisation of the speed of finding accurate classifiers could be undertaken. Also, the problems as discussed earlier with XCSF not producing as many classifiers in areas with a large change in gradient might need to be investigated further.

The main changes that could be implemented to the system are to do with the display of the results of a run of the system. A feature that was lacking from the implementation is that there is no way of allowing manipulation of the way in which the final population can be viewed in the GUI. As the population in the population window is displayed in a text area, there is no sorting mechanism that can be employed to view classifiers sorted by any of the fields displayed. Perhaps a good way of adding this functionality to the system would be to employ a table such as a JTable and to display the classifiers in this. This would also make the system more flexible for changes such as those made to the XCS system to implement the XCSR and XCSF systems. Instead of having to manually set the sizes of all of the columns and using a number of tabs to space the column headings out, these can be set automatically and no tabs need be used at all.

The main functionality that was missing from the implementation of the XCSF is that there was no way to display the final system output. In this report, the final approximations of the function were displayed by showing the classifiers from the system. The final system approximation, however, would include all of the classifiers and would take the form of one continuous line instead of a number of disjoint lines. The way to do this would be to sample the classifiers for different values in the input range, and to take the average output value and log this. This would give a number of points that could be used to plot the system’s approximation of the function.

### 8.3 Concluding Comments

This project was the first time that I have had the opportunity to work with a program with quite as many classes as Dr Barry's implementation of XCS. This made the program seem very daunting, and hard to initially get into. It took much longer than anticipated to develop an understanding of the workings of the program and how all of the classes interacted with each other. This meant that the progress of the project as a whole wasn’t as swift as initially planned, and the extent of work completed wasn’t as deep as was initially anticipated.

This was also the first time that I have ever worked with a program that another person has written, and a lot has been learned about how to read and understand programs designed by somebody else, and also lessons have been learned about how to ensure that any programs that I produce in the future can be understandable to a third party.
The other major reflection from this project is that Microsoft Word is not ideally suited to producing a document such as this one, and it would probably have been more preferable to learn how to use LaTeX initially to save time spent on formatting in the long run.

Overall I have learned a substantial about from doing this project both in terms of technical knowledge and of time management skills.
Chapter 9

References


Part V

Appendices
Appendix A

XCSR

Mux6Environment Class

The entire class written for the real 6-multiplexer environment.
package xcs.cs;

import xcs.cs.Parameters;
import xcs.cs.Message;
import xcs.cs.Action;
import java.lang.Long;

/**
 * The 6 Multiplexor Test Environment
 **/
public class Mux6Environment extends Environment {

/**
 * Calls through to this.localConstructor() to set the sizes
 * for the message and output for this problem, and then sets
 * up the other variables. You should place any initialisation
 * for your own environment variables within localConstructor().
 **/
public Mux6Environment ()
{
    super();
}

public void localConstructor ()
{
    MessageSize = 6;
    ActionSize = 2;
    MaximumRewardSteps = 1;
    /*optimalPopulation = new Condition[] {new Condition("00###0"),
        new Condition("00###1"),
        new Condition("01##0#"),
        new Condition("01##1#"),
        new Condition("10#0##"),
        new Condition("10#1##"),
        new Condition("110###"),
        new Condition("111###")};*/
}

protected void localGetInput ()
{
    input.randomise();
}

/**
 * @return the reward required
 * Give the maximum reward when the bit in the range 0 to 3 given by
 * the integer value of the message bits 4 and 5 is the same as the
 * bit value of the output message. Give the minimum reward otherwise.
 **/
public double getReward ()
{
    int address = ((int)Math.round(input.get(4))) + ((int)Math.round(input.get(5))) * 2;
    int bitValue = (int)Math.round(input.get(address));
    if ((rewardGiven) && (bitValue == output.getAction()))
        return parameters.MaximumReward;
    else
        return parameters.MinimumReward;
}
}
Condition Class

A comparison of the major change made to the Condition class.

Lines 70-75 of original:

```java
public boolean equals (Object obj)
{
    return ((obj instanceof Range) &&
            (min == ((Range) obj).min) && (max == ((Range) obj).max) &&
            (wildcard == ((Range) obj).wildcard));
}
```

Lines 70-76 of the final version:

```java
public boolean equals (Object obj)
{
    return ((obj instanceof Range) &&
            (wildcard && ((Range) obj).wildcard) ||
            (wildcard == ((Range) obj).wildcard) &&
            (min == ((Range) obj).min) && (max == ((Range) obj).max)));
}
```
Appendix B

XCSR Experiment

Condition Class differences

A comparison of the Range class that is inside the Condition class for the version of XCSR that uses Boolean wildcards and the version that sets the values to the extremes of the interval.
The version of XCSR that uses Boolean wildcards:

```java
public class Range implements java.lang.Cloneable {

    public double min, max;
    public boolean wildcard = false;

    public Range ()
    {
        this(Math.random(), Math.random());
    }

    public Range (double minValue, double maxValue)
    {
        super();
        min = minValue;
        max = maxValue;
        this.order();
        this.constrain();
    }

    public Range (double minValue, double maxValue, boolean isWildcard)
    {
        this (minValue, maxValue);
        this.setWildcard (isWildcard);
    }

    public Range (boolean isWildcard)
    {
        this (Math.random(), Math.random(), isWildcard);
    }

    public Range (double containsValue)
    {
        this (containsValue - Math.random() * containsValue, containsValue + Math.random() * (1.0 - containsValue));
    }

    public Range (double containsValue, boolean isWildcard)
    {
        this (containsValue);
        this.setWildcard (isWildcard);
    }

    public Object clone()
    {
        try
        {
            return new Range(min, max, wildcard);
        }
        catch (Exception any)
        {
            return null;
        }
    }

    public boolean equals (Object obj)
    {
        return ((obj instanceof Range) &&
                (wildcard && ((Range) obj).wildcard) ||
                (wildcard == ((Range) obj).wildcard) &&
                (min == ((Range) obj).min) && (max == ((Range) obj).max));
    }

    public boolean contains (double value)
    {
        return wildcard || ((min <= value) && (value <= max));
    }

    public void setMin (double minValue)
    {
        min = minValue;
        this.order();
        this.constrain();
    }
```
public void setMax (double maxValue)
{
    max = maxValue;
    this.order();
    this.constrain();
}

public void copy (Range other)
{
    min = other.min;
    max = other.max;
    wildcard = other.wildcard;
}

public void setWildcard (boolean value)
{
    if ((min == 0.0) && (max == 1.0))
        wildcard = true;
    else
        wildcard = value;
}

public double getMin ()
{
    return min;
}

public double getMax ()
{
    return max;
}

public boolean getWildcard ()
{
    return wildcard;
}

public boolean isWildCard ()
{
    return wildcard;
}

private void order()
{
    if (min > max) {
        double temp = min;
        min = max;
        max = temp;
    }
}

private void constrain()
{
    if (min < 0.0) {min = 0.0;}
    if (max > 1.0) {max = 1.0;}
    this.checkWildcard();
}

private void checkWildcard()
{
    wildcard = wildcard || ((min == 0.0) && (max == 1.0));
}
The version of XCSR that sets values to the extremes of the interval:

class Range implements java.lang.Cloneable {
    public double min, max;

    public Range () {
        this(Math.random(), Math.random());
    }

    public Range (double minValue, double maxValue) {
        super();
        min = minValue;
        max = maxValue;
        this.order();
        this.constrain();
    }

    public Range (boolean isWildcard) {
        if(isWildcard) {
            min = 0.0;
            max = 1.0;
        }
    }

    public Range (double containsValue) {
        this (containsValue - Math.random() * containsValue, containsValue + Math.random() * (1.0 - containsValue));
    }

    public Range (double containsValue, boolean isWildcard) {
        this (containsValue);
        this.setWildcard(isWildcard);
    }

    public Object clone() {
        try {
            return new Range(min, max);
        } catch (Exception any) {
            return null;
        }
    }

    public boolean equals (Object obj) {
        return (min == ((Range) obj).min) && (max == ((Range) obj).max);
    }

    public boolean contains (double value) {
        return (min <= value) && (value <= max);
    }

    public void setMin (double minValue) {
        min = minValue;
        this.order();
        this.constrain();
    }

    public void setMax (double maxValue) {
        max = maxValue;
        this.order();
        this.constrain();
    }
}


class Range {

    public void copy (Range other) {
        min = other.min;
        max = other.max;
    }

    public void setWildcard (boolean value) {
        if (value) {
            min = 0.0;
            max = 1.0;
        } else {
            if ((max == 1.0) && (min == 0.0)) {
                min = Math.random();
                max = Math.random();
                order();
                constrain();
            }
        }
    }

    public double getMin () {
        return min;
    }

    public double getMax () {
        return max;
    }

    public boolean getWildcard () {
        return ((min == 0.0) && (max == 1.0));
    }

    public boolean isWildcard () {
        return ((min == 0.0) && (max == 1.0));
    }

    private void order() {
        if (min > max) {
            double temp = min;
            min = max;
            max = temp;
        }
    }

    private void constrain() {
        if (min < 0.0) { min = 0.0; }
        if (max > 1.0) { max = 1.0; }
    }
}

60
The Perl file

Here is the file written in Perl that was used to average out the 5 runs of the system.

```perl
$files = 5;
for($i=0; $i < $files; $i++) {
    open("XCSR$i", "< report$i.csv") || die "Can't open file: $!
    ");
open (AVG, "> average.dat") || die "Can't create file: $!
    ");
for ($j = 0; $j < 20000; $j++) {
    $filea = <XCSR0>;
    $fileb = <XCSR1>;
    $filec = <XCSR2>;
    $filed = <XCSR3>;
    $filee = <XCSR4>;
    @matrix = ($filea,$fileb,$filec,$filed,$filee);
    $total1 = 0;
    $total2 = 0;
    $total3 = 0;
    for($k=0; $k < $files; $k++) {
        @tmp = split",",$matrix[$k]);
        $total1 += $tmp[1];
        $total2 += $tmp[2];
        $total3 += $tmp[3];
    }
    $average1 = $total1 / $files;
    $average2 = $total2 / $files;
    $average3 = $total3 / $files;
    print AVG "$tmp[0] $average1 $average2 $average3
    
    ");
    close("XCSR$i");
}
close(AVG);
exit;
```
The gnuplot file

The gnuplot file used to plot the average values

average.plt

# Gnuplot script file for plotting data in file "average.dat"
# This file is called average.plt
set autoscale
unset log
unset label
set key 10000,0.2
set xtic auto
set ytic 0.1
set grid ytics
set format y "%.1f"
set title "XCSR with Boolean wildcards"
set xlabel "Exploitations"
set ylabel "Performance"
set xr [0:20000]
set yr [0:1]
plot "average.dat" using 1:2 title "System Performance" with line,
"average.dat" using 1:3 title "System Error" with line,
"average.dat" using 1:4 title "Population" with line
Appendix C

XCSF

WeightVector class

The class for weight vectors written for XCSF.
package xcs.cs;

import java.lang.Math;

import xcs.cs.Condition;

/**
 * Provides a weight vector
 **/
public class WeightVector implements java.lang.Cloneable {

    protected double[] values;

    /**
     * initialises a new WeightVector of random numbers between -1.0 and 1.0
     **/
    public WeightVector (int size) {
        size = ((size >= 0) ? size : 0);
        values = new double[size];
        for (int i = 0; i < size; i++) {
            values[i] = (Math.random() * 2) - 1;
        }
    }

    /**
     * Set the WeightVector
     **/
    public WeightVector (double[] weights) {
        values = new double[weights.length];
        for (int i = 0; i < weights.length; i++) {
            values[i] = weights[i];
        }
    }

    public Object clone () {
        try {
            WeightVector result = (WeightVector) super.clone();
            result.values = new double[values.length];
            for (int i = 0; i < values.length; i++) {
                result.values[i] = values[i];
            }
            return (Object) result;
        } catch (Exception any) {
            return null;
        }
    }

    public void copy (WeightVector other) {
        values = new double[other.values.length];
        for (int i = 0; i < other.values.length; i++) {
            values[i] = other.values[i];
        }
    }

    /**
     * Compute the prediction
     **/
    public double computePrediction (Message input) {
        double result = 0;
        for (int i = 0; i < input.getExtent(); i++) {
            result = result + values[i] * Math.pow(input.get()[i], (double)i);
        }
        return result;
    }

    /**
     * Update the weight vector using the modified delta rule
     **/
    public void updateWeights (double error, Message input, double correctionRate) {
        int extent = input.getExtent();
        // Use the modified delta rule to update the prediction and the error.
        // Note that experience has already been updated elsewhere.
for (int i = 0; i < extent; i++) {
    values[i] = values[i] + ((correctionRate / (extent*extent)) * error * input.get()[i]);
}

public String toString () {
    StringBuffer result = new StringBuffer();
    for (int i = 0; i < values.length; ++i)
    {
        if (i > 0)
            result.append(',');
        result.append(Double.toString(values[i]));
    }
    return result.toString();
}

public double get(int i){
    if (i <= values.length)
        return values[i];
    return 0.0;
}
The test of the Weight Vector class

The class written to test the functions of the WeightVector class.

weighttest.java:

```java
package xcs.cs;
import java.lang.Math;
import xcs.cs.WeightVector;
import xcs.cs.Message;

public class weighttest {
    public static void main(String args[]) {
        // Code to set the message to the values written below.
        /**/
        double[] dble;
        dble = new double[2];
        dble[0] = 0.5;
        dble[1] = 0.6;
        Message msg = new Message(2);
        msg.set(dble);
        System.out.println("Initial message: " + msg.toString());
        /**/

        // Code to set the message to something random.
        /*
        Message msg = new Message(2);
        msg.randomise();
        System.out.println("Initial message: " + msg.toString());
        */

        // Code to set the weight vector to the values written below.
        /**/
        double[] dbl;
        dbl = new double[2];
        dbl[0] = 0.2;
        dbl[1] = 0.3;
        WeightVector weights = new WeightVector(dbl);
        System.out.println("Initial weights: " + weights.toString());
        /**/

        // Code to set the weight vector to something random.
        /*
        WeightVector weights = new WeightVector(2);
        System.out.println("weights: " + weights.toString());
        */

        double o = weights.computePrediction(msg);
        System.out.println("prediction: " + o);

        double t = 0.7734;
        int cnt = 0;
        while (Math.abs(t-o) > 0.00001) {
            weights.updateWeights((t-o), msg, 1.0);
            //System.out.println("new weights: " + weights.toString());
            o = weights.computePrediction(msg);
            //System.out.println("new prediction: " + o);
            cnt++;
        }
        System.out.println("final weights: " + weights.toString());
        System.out.println("final prediction: " + o);
        System.out.println("count: " + cnt);
    }
}
```
Appendix D

XCSF Tests

The test Environment

The environment written to test the XCSF system.
TestEnvironment.java:

```java
package xcs.cs;

import xcs.cs.Parameters;
import xcs.cs.Message;
import xcs.cs.Action;
import java.lang.Long;
import java.lang.Math;

/**
 * The XCSF Test Environment
 **/ public class TestEnvironment extends Environment {

    private static double maximumReward = 1.0;
    private static double minimumReward = 0.0;

    /**
     * Calls through to this.localConstructor() to set the sizes
     * for the message and output for this problem, and then sets
     * up the other variables. You should place any initialisation
     * for your own environment variables within localConstructor().
     **/ public TestEnvironment () {
        super();
    }

    public void localConstructor () {
        MessageSize = 2;
        ActionSize = 1;
        MaximumRewardSteps = 1;
        /*optimalPopulation = new Condition[] {new Condition("00###0"),
            new Condition("00###1"),
            new Condition("01##0#"),
            new Condition("01##1#"),
            new Condition("10#0##"),
            new Condition("10#1##"),
            new Condition("110###"),
            new Condition("111###")};*/
    }

    protected void localGetInput () {
        input.randomise();
    }

    /**
     * @return the reward required
     * Give the maximum reward when the the system prediction is equal to the
     * actual value of the function to be approximated. Give the minimum reward otherwise.
     **/ public double getReward (double prediction) {
        double fromy = -1.0;
        double toy = 1.0;
        double fromx = 0.0;
        double tox = 2*3.14159265;
        double x2 = input.get(1);
        double x = fromx + (tox - fromx) * x2;

        /**
         * The function y = x^2
         **/ double y = x * x;

        double y2 = (y - fromy) / (toy - fromy);
        if (rewardGiven) {
            return maximumReward;
        } else {
            return minimumReward;
        }
    }

    public void printOptimalPopulation () {
        for (Condition c : optimalPopulation) {
            System.out.println(c.toString());
        }
    }
}
```
double absError = (y > prediction) ? (y - prediction) : (prediction - y));
return (minimumReward - maximumReward) * absError + maximumReward;
return y2;
else {
    return minimumReward;
}
}