

G52GRP - Final Group Report
R *Fuzzy* tool-kit

gp12-jmg

Craig Knott (cxk01u)
Luke Hovell (lxh01u)
Nathan Karimian (ndk01u)
George Tretyakov (gxt01u)
Ben Bradley (bxb01u)
With supervision from Jon Garibaldi (jmg)

March 20, 2013

Abstract

This document is the final group report for the group gp12-jmg, for the G52GRP module at the University of Nottingham, School of Computer Science. It details the progress of our project, a fuzzy logic manipulation tool-kit, after it's completion; as well as how we have worked together, what we have achieved, how successful we were, and our process for completing the project. Below are some acronyms that will be used within the document, and their accompanying definitions.

CRAN	The Comprehensive R Archive Network
FIS	Fuzzy Inference System
(G)UI	(Graphical) User Interface
IMA	Intelligent Modelling and Analysis
I/O	Input/Out
MF	Membership function

Contents

1	Description of the problem to be solved	4
2	Background information and research	4
2.1	What is Fuzzy Logic?	4
2.2	Pre-Existing Systems	5
2.3	Platforms and tools researched	5
3	Requirements Specification	6
3.1	Functional Requirements	6
3.2	Non-Functional Requirements	8
4	Updated designs of the system and UI	9
5	Implementation and testing of the system	12
5.1	Key Implementation Decisions	12
5.2	The <i>R</i> Fuzzy Logic Manipulation Code	13
5.2.1	The implementation process	13
5.2.2	The testing process	14
5.2.3	Ensuring Usability	15
5.3	CRAN release	15
5.4	The Graphical User Interface Input System	15
5.4.1	The implementation process	15
5.4.2	The testing process	16
5.4.3	Ensuring Usability	16
5.5	Cross compatibility of the two systems	17
6	The successes and limitations of the project	18
6.1	What was achieved	18
6.2	Technical	19
6.3	Managerial and team working	20
6.4	Problems encountered	22
7	Updated Project Time plan	23
A	Overview of the source code hierarchy	26
B	Extended testing appendix	27
B.1	Functional Requirements Testing	27
B.2	R Test Cases	28
B.3	GUI Test Cases	33
B.4	User Trials	36
C	User Manuals	39
C.1	<i>R</i> Fuzzy Tool-Kit	39
C.2	Java Graphical Input System	48
D	Process of CRAN release	54
E	Initial Designs of the System	58
F	Minutes from meetings held during the life time of the project	60

1 Description of the problem to be solved

As stated by Pedro Albertos and Antonio Sala, “Fuzzy logic is a “natural” way of expressing uncertain information”[1]. It is a form of logic that deals with approximate reasoning, as opposed to fixed, exact values; the variables can have truth values that range from 0 to 1. Fuzzy logic is believed to be better for handling and sorting data, and is an excellent choice for many control system applications due to the way it mimics human control logic. The key advantage of fuzzy logic, as explained by Lotfi Zadeh, is that it allows us to make decisions in an environment of imprecision, uncertainty, and partiality of truth[9].

Our project, set forth by the Intelligent Modelling and Analysis group here at the University of Nottingham¹, was to re-implement the work that they had begun on a fuzzy logic tool-kit, written in the high-level multi-paradigm language, *R*. The aim is for the project to be released online, on CRAN² - a goal that the team have yet to achieve. The tool-kit was developed so that the team had a custom framework for working with fuzzy logic (a large part of their research), as opposed to being bound by the restraints of other such suites, for example MATLAB³.

The aim of our project is to rewrite this tool-kit into a new, standardised format, and then to release this as an *R* package on CRAN. This code will be accompanied by a graphical input system, that will allow users to create the necessary data structures to manipulate within the *R* code. Our system will allow for the entry of variables (constructed from a number of membership functions), and rules, which can then be combined to create the aforementioned necessary data structure for fuzzy logic manipulation, a fuzzy inference system.

2 Background information and research

As all of the group members were new to the concept of fuzzy logic, some research was necessary to ensure we fully understood what was expected of us. We also researched already existing systems, and possible platforms to implement our system with. Below listed are all the investigated materials, and the results of these investigations.

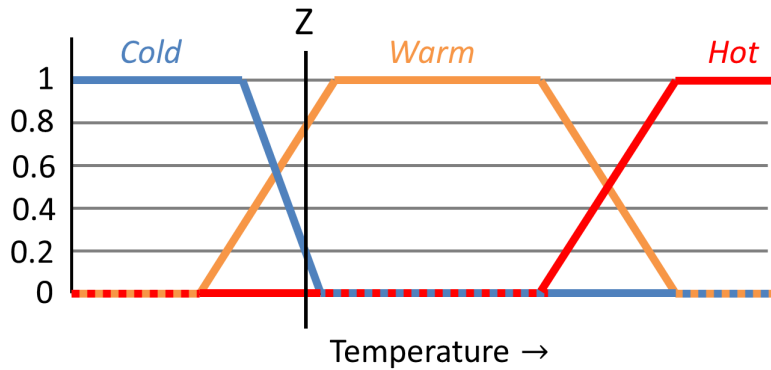
2.1 What is Fuzzy Logic?

As mentioned above, fuzzy logic deals with approximate reasoning, with truth values ranging from 0 to 1, as opposed to classic logic, which has strict truth values of *either* 0 or 1. It is better for handling and sorting data, as values are not restricted, and can have larger variation. An example of a graphical representation of a fuzzy logic set is shown overleaf. On this example, the x axis represents temperature, and the y axis represents the degree of truth. As the temperature increases, you can see how the truth values of the functions increase and decrease, which gives a more accurate representation of the level of warmth. At the point Z , indicated by the black marker, the values of the membership functions cold, warm and hot, are 0.2, 0.8 and 0 respectively. We can use these values to interpret that the temperature is “slightly cold”, “fairly warm”, and “not at all hot”. which is much more precise than a boolean expression of “warm”.

¹Intelligent Modelling and Analysis group, UoN <http://ima.ac.uk/>

²Comprehensive R Archive Network <http://cran.r-project.org/>

³MATLAB Software <http://www.mathworks.co.uk/products/matlab/>



2.2 Pre-Existing Systems

After researching fuzzy logic, we decided to look at systems that already achieved what the IMA group were trying to achieve. Obviously we looked at the two tool-kits that the group had already produced, but we also looked at the fuzzy logic toolbox in MATLAB, and a system we found on the internet called “XFuzzy 3.0”.

Firstly, we looked at the source code for the Version 0.7 Fuzzy Tool kit and Version 1.7 Fuzzy Tool kit, both produced by the IMA group. These would be used, in conjunction with our specification, to create the brand new fuzzy tool kit. These gave us an idea of the amount of code we would be required to write, how advanced the code was, supplied us with useful formulae for fuzzy logic, and showed us some tips and tricks involving the *R* language.

We then looked at MATLAB, which we had also used during our learning of *R* and fuzzy logic. Which is a multi-purpose tool that has a small section devoted to the use of fuzzy logic. This was similar to what we would be implementing, but our system would be focusing entirely on fuzzy logic, and had scope to go much more in-depth. The MATLAB fuzzy tool kit was useful so that we could see how a graphical user interface could be applied to fuzzy logic, and to give us possible ideas on how we should implement ours. One advantage that our project would have over MATLAB was that it would be free of charge and open source, whereas MATLAB requires a purchase.

The final product we looked at was XFuzzy 3.0, which was very similar to what we would be creating. The software itself had not been updated since 2003, so was used as more of a reference on alternative methods used to implement fuzzy logic. This software was command line based, which made the plotting and manipulation of graphs much more difficult for the user. Our project aims to solve this problem by implementing a graphical user interface to the system so that the users can actually visualise their graphs, to make manipulation much easier.

2.3 Platforms and tools researched

We also needed to research platforms and tools that could be used for the implementation of the system. We decided that the system would be best received if it were available on multiple platforms, so we focused on all major operating systems. We wrote the source code in *R*, which already has a cross platform console, and the graphical user interface in Java, so that users would not be restricted by their operating systems if they wanted to use our software. We looked into

appropriate algorithms and formulae for generating the values we required, and found that the MATLAB function documentation combined with the 0.7 and 1.7 tool-kit source codes gave us a good idea on the best formulae to use, and an explanation of how to use them. As far as tools go, we decided to use the cross platform *R* console available on CRAN for the implementation of our code, and the adoption of SVN ⁴ for version control, as both of these were easy to access and use.

We also had to look into tools that were necessary to the success of the project, this included a possible Java/*R* interpreter, and a Java graph drawing package. As far as Java/*R* interpreters, we could not find a suitable package that would achieve the functionality that we required, and the complexity of creating our own was far too great. It was for these reasons that this feature was dropped.

As far as a Java graph drawing package, we found two main contenders, JUNG ⁵ and JFreeChart ⁶. Upon completing further research, we decided that we would use the JFreeChart package. This is due to JUNG not being a complete product, and requiring extra programming to build a tool that uses JUNG, as opposed to the ability to use JFreeChart straight away. The JFreeChart package is small and easy to use, full of features, and comprehensively documented. Describing graphs in JFreeChart was simple enough for aesthetically pleasing diagrams, yet powerful enough to allow us to tailor our graphs to our needs. Both packages came with full support forums, but it seemed logical to use the product that was already fully developed, and that didn't require us to build a framework around it before we could use it.

3 Requirements Specification

Below we have listed the functional requirements (things that our software must accomplish) and non-functional requirements (restraints placed on our system) of our software system. As our system incorporates both a command line *R* interface, and a Java graphical user interface, some requirements are tailored specifically to one or the other. Any requirements that apply only to the *R* interface will be marked with the letter "R", and a "G" will be used for the GUI. An asterisk (*) next to a requirement means that it has changed since writing our previous report, and reasoning for this will be elaborated in section 6.1.

3.1 Functional Requirements

1. Users will be able to create and manipulate Membership Functions, this includes...
 - 1.1 ...Gaussian curves
 - 1.1.1 ...and specify the parameters
 - 1.2 ...double Gaussian curves
 - 1.2.1 ...and specify the parameters
 - 1.3 ...trapezoidal graphs
 - 1.3.1 ...and specify the parameters

⁴<http://subversion.apache.org/>

⁵<http://jung.sourceforge.net/>

⁶<http://www.jfree.org/jfreechart/>

- 1.4 ...triangular graphs
 - 1.4.1 ...and specify the parameters
- 1.5 ...evaluating membership functions (R) *
- 1.6 ...defuzzifying membership functions (R) *
- 1.7 ...plotting membership functions *
- 2. Users can manipulate FIS objects, from which they can...
 - 2.1 ...create a new FIS, where they can specify...
 - 2.1.1 ...a FIS name
 - 2.1.2 ...a FIS Type
 - 2.1.3 ...an And Method
 - 2.1.4 ...an Or Method
 - 2.1.5 ...an Implication Method
 - 2.1.6 ...an Aggregation Method
 - 2.1.7 ...a Defuzzification Method
 - 2.2 ...add an input variable, from which they can...
 - 2.2.1 ...add a number of membership functions
 - 2.2.2 ...specify the parameters for these membership functions
 - 2.3 ...add an output variable, from which they can...
 - 2.3.1 ...add a number of membership functions
 - 2.3.2 ...specify the parameters for these membership functions
 - 2.4 ...add a manipulation rule, from which they can
 - 2.4.1 ...specify a rule (R)
 - 2.4.2 ...use the “Logical Rule Builder” (G)
 - 2.4.3 ...invert any inputs or outputs *
 - 2.5 ...evaluate a FIS (R) *
 - 2.6 ...plot a 3D surface of the FIS (R)
- 3. The user will be able to use file I/O to save and read files
 - 3.1 The user can load in a FIS from a file
 - 3.2 The user can save a FIS to a file
- 4. Users will be able to print certain objects, including...
 - 4.1 ...FIS structures (R)
 - 4.2 ...FIS rules (R)
- 5. The user can use inbuilt variables to create a new FIS (R)

3.2 Non-Functional Requirements

Availability and Extensibility

The availability of the *R* fuzzy tool-kit project will be solely via the internet, and accessible to the public as a package on CRAN. It will be continually kept up-to-date throughout this development year as additions are made to it via a central SVN repository, with the latest version being presented through CRAN. After the academic year of 2012/2013 is over, responsibility of the software is passed on solely to the IMA group at the University of Nottingham, to do with as they please.

The system will be extendible and expandable once the final, official release is uploaded onto CRAN. The source code can be requested and any further development beyond the scope of this project can take place, under jurisdiction of the IMA group. Considering the storage medium is not under our control, availability; such as uptime, will be a responsibility of the CRAN online service.

Maintainability

The code is kept as modular and as readable as possible by the development team, ensuring maintenance and modification (such as the addition of new features) is easy to accomplish by external users. In regards to the project source code, this was maintained and modified using SVN on a central repository but, after the life time of the project, can be changed as seen fit.

Operability and usability

The toolbox is designed to be streamlined and as easy to use as possible due to the wide spectrum of potential users. The GUI element of the system will provide an interface to construct fuzzy logic data structures in a simplified manner along with the aid of manuals and documentation.

Quality

The system should look, feel, and perform up to the standard and expectation of the user. The final release product will contain minimal faults.

Resource requirements, constraints and required capacity

In order to use the final version of our program, the user will need to have the Java Virtual Machine (JVM) ⁷ installed or be familiar with the R console available on the R website⁸.

Platform Compatibility

The final program will be compatible with any recent version of Windows, MacOS or UNIX.

Security, reliability and robustness

User input is a potential security vulnerability in any program and, as such, input validation and verification is paramount. The final software is to contain adequate error trapping for all user entry fields (for instance, ensuring the user is entering numbers where necessary). This is to avoid the program behaving unexpectedly, passing incorrect/unreliable data, or simply crashing.

⁷Java download <http://www.java.com/en/download/>

⁸<http://www.r-project.org/>

Building on this, error checking and validation is required for the file I/O, which is achieved through a multitude of error checks, and testing the input files against regular expressions designed to catch invalid formats.

Documentation

Both the R package and Java classes will have commented code explaining what each segment or block of code does. For example, a function's inputs, outputs and inner workings will all be commented; which should give a clear idea of its purpose and usage. We will be writing JavaDoc comments in our Java code, so that documentation can be automatically generated for us. Further documentation will include both a manual of the *R* system and the graphical user input system, which will explain their functionality and usage.

Disaster Recovery

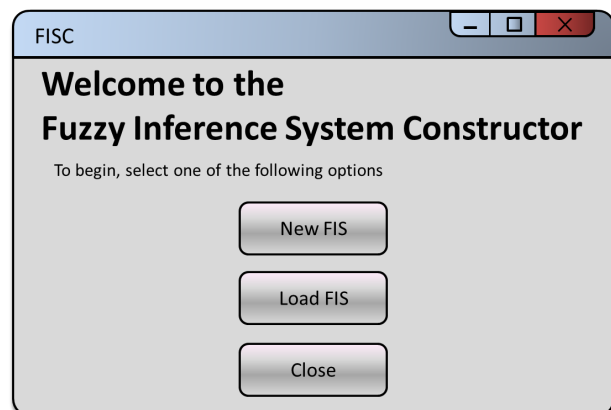
The source files for our project, as well as the documentation, is all stored on the University of Nottingham's Computer Science servers, and are thus backed up to prevent data loss. When the project is finished and uploaded to CRAN, it will be the responsibility of their server(s).

4 Updated designs of the system and UI

In this section, we have included the designs for the user interface of the system, followed by a quick explanation and justification of each. These designs were the basis for the GUI of our Java input system, and thus are detailed enough to indicate what elements are to be used, and to do. Note that, for obvious reasons, no designs for the *R* code part of the specification could be produced. It is worth noting that the initial designs of the system are present in the appendixE, for comparison. The reason we decided to do both initial designs *and* updated designs, is that studies suggest that an iterative design methodology can improve the usability of a product[2].

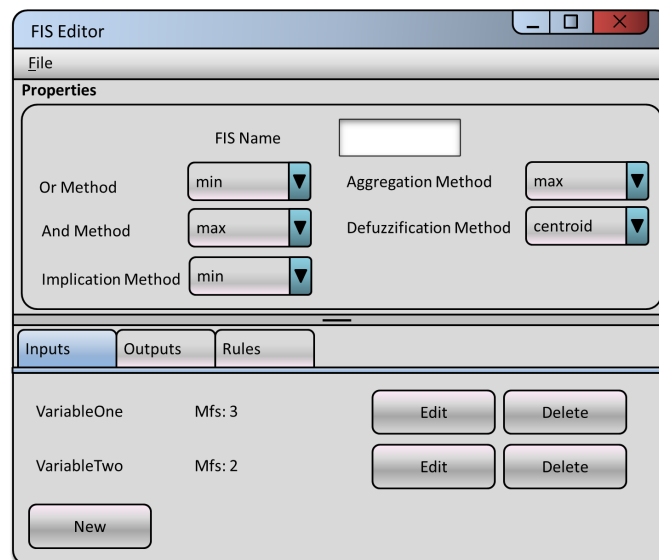
The Main Menu

The main menu would be the greeting screen for the system, and as such needed to both be aesthetically pleasing and welcoming for the user. Instead of directly launching the FISEditor, which may have over loaded the user with too much information, the main menu was launched on start up, to ease the user into the system. They were faced with a choice of leaving, creating a new FIS, or loading a previous FIS. Both of the latter options would then launch the FISEditor window, but not before the user had chosen to view it. The main menu was as simple as it could be, whilst still providing reassurance that the user had launched the correct program, and giving them ample directions to begin their journey.

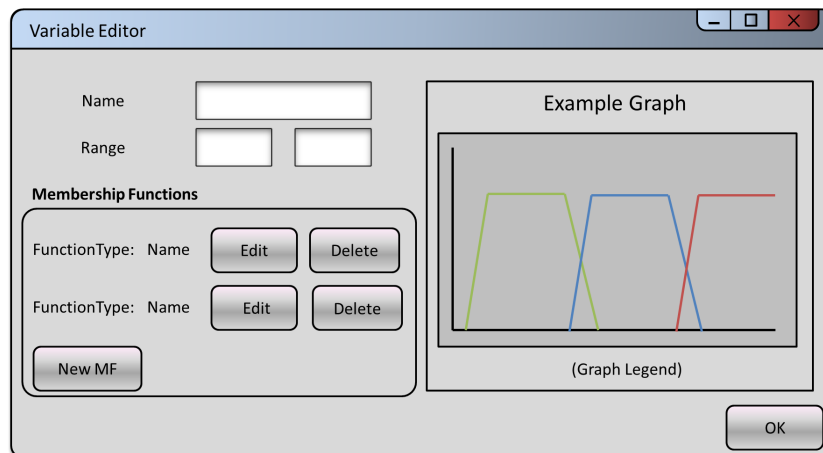


The FIS Editor

This was the main window of the system, and would be what the users would be interacting with the most. It was vital for this to be easy to use and interact with, so that the user would not be confused. The function selection was changed from text input boxes, to combo boxes, so that less mistakes could be made, which led to less user frustration. The inputs, outputs and rules were all listed on separate tabbed panes, so that the user would not be overwhelmed with information, and could browse at their leisure. When creating a new variable or rule, a new window would be launched, so the user could see the clear separation between the FIS parameters and the new data objects they were creating. Each created object had the ability to be edited or deleted as the user saw fit, giving them complete control of the system.



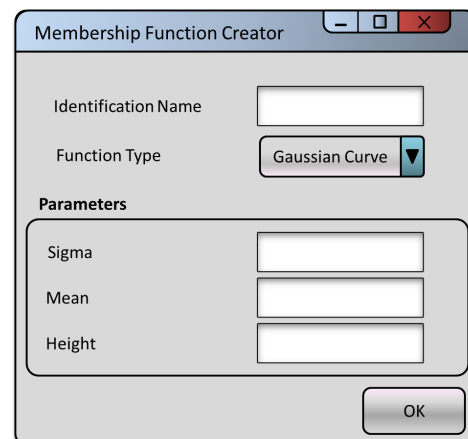
The Variable Editor



This would be the second most used screen, as variables were one of the vital objects necessary to construct a functional FIS. This window allowed for the specification of a variable, and all the membership functions that were contained within it. Each of the membership functions created would be done so in a separate window, so that the separation of the variable inputs and the membership function inputs was obvious. Each of these membership functions could be edited or deleted as necessary, to facilitate the complete control of the user. Some changes were made to the original design of the variable editor window, including the removal of the “Number of functions” parameter, and the repositioning of the graph viewer panel. This was because the number of functions could be calculated by the system automatically (saving the user time and effort working it out and entering it themselves), and the graph viewer was moved because we realised it was an important aspect, and required more space.

The Membership Function Creator

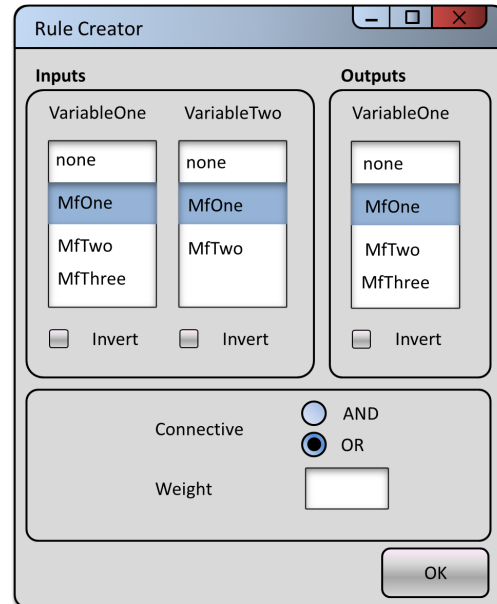
The membership function creator was another well used window, and was designed to be as simple as possible for the user to enter in membership functions. The user could specify a name, and then use a combo box to select which type of membership function they were specifying. This prevented them from entering in unsupported functions. After selecting a function type, the parameters box would be updated, with labels and text entry fields that were related to the membership function being entered. This further reduced errors that the user could make, as it was explicitly stated what they were required to enter.



A slight alteration to the original design was the removal of the graph viewer panel, which was removed as we didn't feel it necessary to be on both the variable editor *and* the membership function creator.

The Rule Creator

The final operable section of the system was the rule creation window. This was used, after the user had specified some number of variables, to create logical rules that would govern the actions of the system when it was evaluated in the *R* code. The creation of rules in the *R* code required the user to specify a range of numbers, which could often be confusing, and lead to mistakes. The GUI, however, allowed the user to view the rules they were creating, and select from a list of possible options, so that they could see the rules forming before them. The logical rule creator was one of the main advantages of the GUI, as its design was simple, but its functionality was great, and it greatly reduced mistakes made during the rule creation process.



5 Implementation and testing of the system

5.1 Key Implementation Decisions

During the life time of the project, we needed to make decisions over which tools would be most appropriate for the tasks we needed to complete. It was important for us to use the best possible tool for each job, so that we were not hindered, and could complete the task to the highest standard. Discussed below are the different categories of tools we needed to decide upon, and which we chose for each task.

Main programming language - *R*

In the specification of the project, the language that our supervisor required us to write the tool-kit in was the high-level multi-paradigm language *R*. A lot of the code for the tool-kit had already been produced by the IMA group, and rewriting this all from scratch in a different language would have been more difficult than simply recreating the code in *R*, even if we did have to learn it as a new language.

GUI Language - Java

We were also required to write a graphical user interface for our system, which would be used for creation of FIS structures, as doing this with a GUI is much easier than attempting to do so through a command line interface. We decided that Java would be the best language to write this in, as looking for a graphical interface package for *R* returned very few viable options. The main reason we chose Java was that all of the group members had done at least one semester's worth of Java coding, and were mostly comfortable with it.

GUI Package - Swing

After deciding on using Java, we needed to pick a suitable library to write the graphical user interface in. We decided on Swing⁹, as some members had worked with this package before, and most group members were partaking in the graphical user interface module (*G52GUI*), which was being taught using Swing.

Graph Drawing Package - JFreeChart

The graph drawing package we decided upon was JFreeChart, as this was a freely available library, with all the capability and features that we required for this task. There was also ample help available online, so that any problems we encountered could be solved. The actual research part of this can be found in more detail earlier in this report (*Section 2 - Background information and research*).

Operating Systems Supported - Windows, MacOS and UNIX

To appeal to a wider audience, we decided to tailor the tool-kit to be usable on all three of the main operating systems available at the time. The *R* Console available on CRAN is already available on a wide variety of UNIX platforms, Windows and MacOS - which means that our source code could be run on any of the main operating systems without us needing to do any extra work. Our GUI would also be usable on these main operating systems, as it were written in Java, which is a cross platform language.

5.2 The *R* Fuzzy Logic Manipulation Code

5.2.1 The implementation process

During the implementation process of both the *R* code, and the Java GUI, we followed a slightly modified, and more iterative version of the linear sequential model of software engineering [5], which follows an analysis, design, code generation, testing and support pattern. We adopted this method, but introduced an iterative element to it, to make the approach more agile, and to allow us to respond to change more effectively.

The *R* code was to both work in conjunction with the Java GUI, and to be a standalone file release on CRAN. The code was developed in a modular and independent design style, and as such new features were easy to implement when necessary (the modularity being supported by the functional programming style of *R*). No external packages were required for our *R* code, which helped in it's task to be a stand alone application.

⁹<http://docs.oracle.com/javase/tutorial/uiswing/start/about.html>

The *R* development began after the examination of the existing source code produced by the IMA group. From here, the *R* coding group was able to identify its key goals, and plan how these could be modified to fit the aims of our project. Preparation included agreeing upon basic implementation tools and how to reimplement the code. The original source code was split up into independent subsections, so that we could program parts that were not reliant on others. After creating all of these independent sections we began, following our supervisor's advice, programming the FIS structure that would be required by all following functions. This FIS structure was implemented using a series of list and sub-lists to store all the necessary data, as this fit with the general functional theme of the language. Following the FIS structure completion, the FIS manipulation functions could be produced, which included the adding of rules, variables, and membership functions.

During the development, we noticed that the storing of values in membership functions needed reworking. Luckily, due to our modular coding style, this change could be adopted into the system with little difficulty. When the GUI coding was beginning, interaction between the two systems was considered. Due to the raw processing power of the *R* language, we decided that the majority of manipulation would take place in the *R*, whereas the GUI would be more of an input system, which allowed users to easily create the FIS structures that they required. We needed to agree upon cross compatibility standards of the FIS structures, which is discussed further in section 5.4 (*Cross compatibility of the two systems*).

After file input and output were completed, we could begin work on the final functions required to meet the objectives of the project. These were the evaluation and plotting functions that were discussed in the project brief, which would return graphical and numerical values for the FIS structures we were creating. These required the most time to implement, as they required a thorough understanding of fuzzy logic and the evaluation process. This problem was tackled with intensive research, breaking these functions into their core components, and coding small subsections at a time. During this time, the *R* team worked on releasing the code onto CRAN, as this was a vital goal for the project, and was stated as being more important than finishing this last function. Luckily we were able to achieve both the goal of uploading a package to CRAN, and completion of the evaluation function.

5.2.2 The testing process

Obviously, as each function was being programmed it was being tested against both MATLAB and the original source code to make sure that the values being produced were as they were expected to be. The testing of the *R* was purely functional, and as such we had to ensure that every element worked as expected, and appropriate error messages were delivered where failures occurred.

The general process of testing was to use the MATLAB or previous tool-kits to produce a FIS structure, calculate the values of the different functions we had implemented, and then record these results. We could then test what our system actually produced against these results, to see whether or not we had produced a system that worked correctly. It was important to check not only the individual elements worked, but that the system worked as a whole, and that communicating sub-sections of the system were doing so correctly. A complete listing of the tests conducted can be found in the appendix B.2.

5.2.3 Ensuring Usability

Whilst we had no control over the interface our users were using, we could still ensure that our code was produced to be as easy to use as possible. The main way that we could do that for our *R* code was to ensure two things; error trapping was present wherever necessary, and error messages were appropriate and useful. We made sure that whenever the user was required to enter data, appropriate checks would take place before any processing would. This included ensuring that the user was specifying the correct number of inputs, and that they were of the correct format. This meant that all errors displayed were ones that we had constructed ourselves, and not the default *R* ones. This meant the error messages could be much more specific, and guide the user through the problem they were having with exact instructions. In the example code below, the user has attempted to enter a membership function without a name, but a useful error message is provided, informing them that they cannot.

```
> g1 <- gaussMF("", 0:10, c(1,1,1))
Error in gaussMF("", 0:10, c(1, 1, 1)) :
Error: You must name all objects
```

5.3 CRAN release

Throughout the lifetime of the project (including in the initial brief) our supervisor had made it very clear that uploading a package to CRAN was paramount. This would mean that there would be a solid first release of the code, that could then be distributed across the internet. This meant that members of the public could access and modify the code to their liking, and submit possible changes. This made for a much more user-based implementation of the system, as the users would be able to access and edit the code themselves. This goal was one that the IMA group had not yet achieved, which was the reason for its importance. By the end of the project, we had submitted our *R* package for approval by the CRAN team, unfortunately they did not respond to us in time for the project deadline. Hopefully the package will be up and available on CRAN in time for the open day and presentation. The process for submitting to CRAN is a long one, and in the appendix D we have detailed the process necessary, and the steps we took to upload our package.

5.4 The Graphical User Interface Input System

5.4.1 The implementation process

After some production had gone into the *R* code, we felt it suitable to start the graphical input system for our software. The objective of the GUI was to make the entry of data much simpler, as it was laid out in a clearly visible fashion, as opposed to the confusing command line interface of the *R* console. To aid this, we designed the GUI to be in an “order” of sorts. The user would complete tasks in a chronological order, to make the process much easier for them to understand and replicate [4]. The implementation process itself took three main iterations. The first of these iterations was the initial graphic design process, in which all of the members of the GUI coding team were assigned a window to build, but not to add any functionality. The reason for this stage was because we knew Java Swing could be difficult to work with, and getting the window layout completed as early as possible would benefit us greatly in the long run. The second stage was to put these separate windows together to create one system, and add the actions and functionality

that it required. The purpose of this second stage was to get the “bulk” of the software done and allow for communication of data between the different parts of the system. The final stage was polishing and evaluation. In this stage we rewrote the system from the ground up, so that it was in a higher quality state, following a pseudo model-view-controller design pattern, with well commented and documented code. It was in this stage that any final features could be added or removed.

Through this iterative process of creating a small part of the system and then improving and expanding upon it; we were able to produce a final system that was polished, (to our knowledge) error free, and generally of a higher quality. There was no structured order to the implementation of the GUI, but this was countered by us using a kanban board which had a list of all the tasks required to be completed. This gave the programmers the freedom to code what they felt they could, or what they wanted to. Any bugs would be (when they were found) put on the kanban board, so that someone with more time could fix them, and they would not be forgotten about.

To avoid any potential copyright infringements, I would like to take this time to talk about our usage of JFreeChart. To draw graphs in Java on our variable constructor window, we used a free open source Java package, called JFreeChart. This allowed us to easily plot graphs and display them on objects in the Java Swing library. Obviously we take no credit for this package, but are free to use it, as we have included the libraries files in our source code folder, and have included a reference and link to their website in every class that we have used the package's functionality.

5.4.2 The testing process

Testing was vital to ensure compatibility between the data the user entered, and how it was displayed in the *R* code later when it was being manipulated. There were two different types of testing that needed to be undertaken; usability testing, and functional testing. Usability testing consisted of allowing different test subjects to use the system to see how they felt about it. From this we could gather whether or not the system was inhibiting them, or if it were their knowledge of fuzzy logic. The functional testing was to ensure that all mentioned functional requirements were present in the system, and ensuring that all values, data and calculations that was in the system were correct. For this we checked our functional requirements against our system to make sure we had included everything. Obviously, going through each of the tests in this document would take up far too much time, but a complete listing of the tests conducted, and their results, have been documented in the appendix B.3.

5.4.3 Ensuring Usability

There were two main elements to consider to ensure the usability of the system. These were the simplicity of the system, and the facilitation of rectification of errors. This meant that the system had to be designed and produced in such a way that it was easy and intuitive to use, but any errors that were made by the user were explained and could be solved, with out leaving the user feeling helpless or frustrated.

To keep the system simple, we kept it clean and uncluttered, so the user would never be confused as to what they were doing or where it was. To further help this, we made sure that whenever the user was adding a new data structure (for example, a new membership function to a variable),

the input dialogue would be opened in a new window. This was so the user knew they were doing something new, and would not be confused with the values that they had already entered. Simplicity was important, because studies have shown that users lose more than 40% of their time to frustration, and that in most of these cases, the user ends up angry at themselves, angry at the computer, or feeling a sense of helplessness[3].

To further the simplicity of the system, suitable error messages were important; as they informed the user that they had performed an incorrect action and that it needed rectifying. As the users of the system would be ranging in ability, from none to a lot, we needed to make sure that the error messages were appropriate for any audience. We looked at Ben Shneiderman's paper on the future of emerging systems [7], which gave us guidelines on how to write suitable error messages, and generally design our system better. This included being positive about errors, and focusing on what should be done to fix them, as opposed to condemning the user for their mistakes; a few examples of these can be seen below.

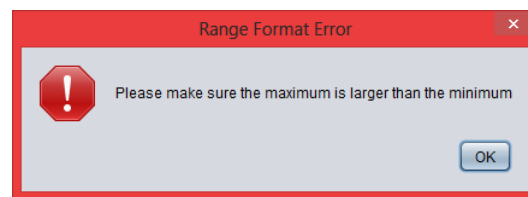


Figure 1: Error produced when the user specifies the minimum of a range to be greater than the maximum

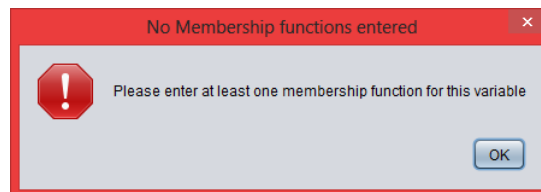


Figure 2: Error produced when the user attempts to create a variable with no added membership functions

5.5 Cross compatibility of the two systems

In the original specification, we were required to produce a graphical user interface that would aid the usage of the *R* manipulation system. As the project progressed, we decided that this GUI system would act as a FIS construction system, that would allow users to easily create the FIS structures they needed, that could then be manipulated by the more powerful *R* code. This meant that the two systems were (for the most part) independent. There was, however, one part of the system that was specifically necessary to be compatible between the two systems, and this was the *.fis* save file format. This format was based mostly on the MATLAB equivalent. The GUI coding team and *R* coding team decided on a format that would be uniform across both systems, and regular expressions were produced so that the validity of these files could be verified.

6 The successes and limitations of the project

6.1 What was achieved

In respect to our requirements, we produced a piece of software that was more or less exactly what our supervisor had specified. We produced a revised and reconstructed fuzzy logic manipulation tool-kit in *R*, and a fully functional graphical input system in Java. I have included numbers in brackets after certain statements in the following passage, each of which refers to a requirement that the statement references.

The main purpose of our system was the creation and manipulation of “Fuzzy Inferencing Systems”. These were objects consisting of variables and rules. Both our *R* code, and the accompanying GUI allowed for the creation of these FIS objects (2, 2.1). Our systems both also allowed for the entry of the parameters of these FIS objects, including a name (2.1.1), a type (2.1.2), an “And Method” (2.1.3), an “Or Method” (2.1.4), an “Implication Method” (2.1.5), an “Aggregation Method” (2.1.6) and a “Defuzzification Method” (2.1.7). These parameters defined the way the the FIS would be evaluated, and could be easily changed in both our systems. As mentioned before, variables are a required part of a FIS for it’s processing, and as such both of our systems included the ability to create and edit these variable objects - both inputs and outputs (2.2, 2.3). Variable objects store a list of membership functions, and a range in which they could be plotted between (2.2.1, 2.2.2, 2.3.1, 2.3.2). The membership functions that the user could enter were either Gaussian (1.1), double Gaussian (1.2), triangular (1.3) or trapezoidal (1.4) (1). Naturally the parameters of all of these functions were specifiable and editable (1.1.1, 1.2.1, 1.3.1, 1.4.1). After being specified in either the GUI or the *R* code, the individual membership functions that were created could be further manipulated using some functions provided in the *R* tool-kit. These functions were “evalMF”, which evaluated the membership function (1.5), “defuzz”, which returned a “crisp” defuzzified value for a membership function (1.6), and “plotMF”, which would plot a number of membership functions on a graph (1.7). The other main functionality that the *R* code possessed was the ability to evaluate fuzzy inference systems as a whole, which would evaluate the system using the rules, inputs variables, and outputs variables (2.5). Included in the *R* system was a function, “tippertest”, that would create and return a fully functioning FIS object, that could be used to help users understand the format of FIS structures, or to conduct some practice analysis (5). This FIS (and any other, for that matter) could be printed and displayed to the user using the “showFIS” function (4.1).

The system also allows for the creation and specification of “Rules”. These are a set of logical constructs that govern how the system should react dependant on certain values, for instance “(*food* = *delicious* \vee *service* = *excellent*) \rightarrow *tip* = *generous*” (2.4). In the *R* command line interface, the user could specify a rule using a vector of numbers, which would consist of the value of each variable in relation to their membership function, the weight of the rule and the connective to be used (2.4.1). This could often be confusing, and wasn’t overly user friendly, so the GUI employed a “Logical Rule Builder” that would allow the user to create their rule by selecting values from combo boxes that would have the names of membership functions in them (2.4.2). Both the *R* and the GUI also possessed the ability to invert values (for instance, *food* \neq *delicious*) (2.4.3). These rules could be printed out in the command line interface, to be displayed back to the user (4.2).

As it was required for the GUI and *R* tool-kit systems to interact, the ability to save and load files from disk was present in both systems, with vigorous error trapping present in each (3, 3.1, 3.2).

Elements listed in the functional requirements (section 3.1) that were marked with an asterisk (*) were done so to indicate that a change of their specification had taken place between the interim report specification and now. The reasons for most of these changes were due to the fear of incompleteness, or that they strayed too far from the updated aims of the project. We decided that, instead of having a fully-fledged *R* fuzzy tool-kit, and a parallel GUI system, we would focus on processing in the *R* system, and data entry in the GUI system. This meant that features such as evaluation of membership functions (1.5), defuzzification of membership functions (1.6), and plotting of membership functions (1.7) were removed from the GUI, and kept solely in the *R* code. This decision also meant that the functionality of evaluating FIS structures was also removed from the GUI specification, as we wanted to keep all processing within the *R* tool-kit (2.5).

6.2 Technical

In this section we will look at our system as a whole, what it has achieved, and what technologies were used to achieve this.

The *R* provided the ability to both create FIS structures, and to later evaluate them. We chose to implement the manipulation functionality in the *R* code only, as this was a more powerful processing language, and was designed for this kind of work. This also meant that the *R* code could be a single file, and distributed as a complete package on its own (making it eligible for release onto CRAN). The *R* code provided all the functionality that was required from the specification, and the code is written in an elegant and easy to understand way, with comments where necessary. Considering no members of the group knew how to program in *R* prior to the project beginning, we believe that the code we produced was of a decent standard.

Sharing of data between the *R* code and the GUI was necessary so that the structures created in the GUI could be manipulated. An agreed style of the *.fis* structure was established and both systems needed to ensure that the files they were producing were of the exact same format. To do this, we wrote a selection of regular expressions that would scan the files being read in, and throw an error if it were not of the correct format. This prevented any badly formatted files being read in, and potential crashing either system.

The most prominent success of the GUI was the ease of creation of FIS structures. As the *R* code was entirely command line based, the production of the FIS structures necessary could sometimes be a tedious job, and prone to errors. The GUI facilitated a much simpler input method, error trapping on every section, and a clear indication of what was needed for each specific part of the system. The GUI not only provided simple and intuitive creation of the necessary data structures, but it gave the user full control of the editing and deletion of them as well. The user was in control of the system at all points which meant that any mistakes they made could be rectified at a moments notice.

To further emphasize the usefulness of graphical systems, the variable editor window had a graphical representation of the membership functions that were being added to it (this functionality was provided by the JFreeChart package). This meant that the user could see exactly what they were constructing, as they were constructing it (where as in the *R* they were required to run a separate function for this), entirely automatically.

6.3 Managerial and team working

It's important to evaluate the success of not only the project, but those working on it, and how organised and successful we were as a collective. The main aspect of working as a team is communication within that team. Good communication means that any short comings in the project can be identified earlier, and hopefully also rectified, which leads to a healthier project. We used three main techniques to facilitate better communication within our group, these were; weekly informal and formal meetings, the use of the online social media website Facebook¹⁰, and mobile telecommunications. The weekly meetings gave us an opportunity to raise issues that we had been facing in the week, inform the other group members of our progress, and to plan for the week ahead. The informal meetings would be for discussing smaller, trivial parts of the project, where as the formal meetings were there to raise larger issues, and brainstorm as a group. Facebook was used as a online meeting point, where all information could be displayed to all members of the group. As most of the members were frequent users of the service, it was almost second nature to them to begin posting on the group Facebook wall to update others on problems and progress. Text messaging and phone calls were used mostly in emergencies, when vital information was needed, or when absence for meetings arose. As a group, we communicated fairly well, but a few members of the group were sometimes forgetful, and would not update the others on their progress, or any problems that they had. This, however, was covered in the formal and informal meetings, where all the events of the week were summarised. With the adoption of these methods, and how most group members adhered to them, it's fair to say that the group, as a whole, communicated well.

We were determined to work well in other areas of team working, and as such we researched into how to be a better team unit. We discovered that there are five main weaknesses to teams, and after noting these, we could then employ methods to prevent them. These dysfunctions were as follows; the absence of trust, a fear of conflict, a lack of commitment, the avoidance of accountability, and an inattention to results [6]. There were many methods used to avoid these dysfunctions, and each came with a varying degree of success. These methods were as follows; promises of constant communication (including informing other group members of our strengths and weaknesses, so that we could be assigned to appropriate tasks), promotion of constructive criticism (this was to ensure that only the best ideas were finalised, and that the team worked together, as opposed to being led as a dictatorship), realism of importance (as this module was worth 15% of our second year grade), the usage of version control for traceability of accountability (to ensure transparency within the group, and the acceptance of ones mistakes), and the constant evaluation of the project as a whole (a process in which we would explain our individual progress, and compare this to the greater project, so that we did not lose track of the global success). We feel as though, having these dysfunctions in mind and already having planned solutions for them, that working as a team was much easier, and fear of failure (in regards to team working) was much less of a worry, which allowed for the focus of the group to be on the work itself, as opposed to the team. This lead to a much more prosperous working environment within the group, and allowed us to progress at a steady rate throughout the entirety of the project life time.

¹⁰<http://www.facebook.com>

As far as the management of the project goes, the second half of the project was far more successful than the first. We did get a large portion of the work done in the first half of the project (first half here meaning prior to the Christmas Holidays), but this pales in comparison to how much more organised the second half was. After the G52SEM examination in January, we decided to begin employing some techniques of agile development to the management of the project. This included two main aspects; more frequent informal meetings, and a kanban board. The informal meetings were essentially five minute stand up meetings, that allowed us to discuss our individual progress, which meant we could evaluate the global progress. We also used an online kanban board service¹¹, which we set up and documented all the tasks that would need to be completed to finish the project. These tasks could be colour coded, and assigned to specific group members. The general practice was to complete the task you were currently on, and then browse the kanban board for a task you feel you would be able to complete, and then focus on this. The columns on the kanban board were “Next” (indicating tasks not yet started), “Analysis” (tasks that required further identification or explanation), “Development” (tasks that were being actively developed), “Acceptance” (tasks that were being quality assured), and “Live” (for those tasks that were completed). This was a much more formal way of assigning tasks, as opposed to the first half of the project, where tasks would only be assigned in the meetings, and some members could go days without having any tasks to complete. With our new kanban board system, any free time could be dedicated to one of the tasks on the kanban board, which allowed for a quicker pace to be applied to the development of the project. Both the daily scrum meetings, and the kanban board facilitated a much more organised project, which allowed us to complete the project on time, without worry that any of the features were either underdeveloped, missing, or of a lacking quality.

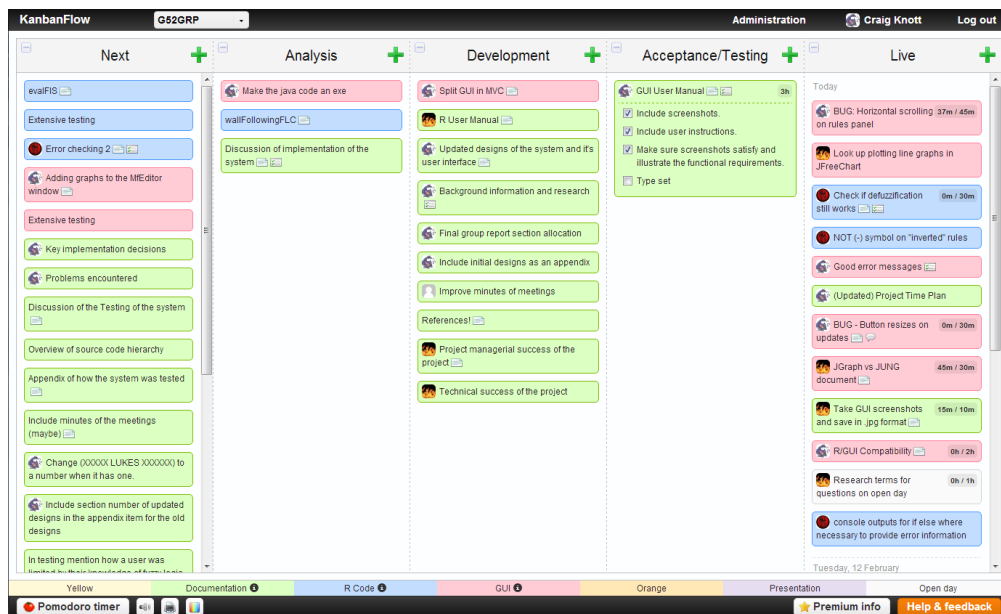


Figure 3: A screen shot of the kanban board, as it was early in February

¹¹<http://www.kanbanflow.com>

With all that said above, we feel as though both the project was managed well, and that we worked well as a team. We feel as though we should have employed more formal management techniques earlier than at the half way mark, but do not feel as though this negatively impacted us too much, as the project manager was still very much organised, and attempted to assign tasks wherever possible. If we were to attempt a project like this again, we would also employ more agile techniques, such as “Sprints” (where a specific task would be the subject of a several week work cycle), burn down charts (visual representations of the work remaining in the project), and more formal time management techniques, like wide-band delphi time estimations coupled with a Gantt or PERT chart, so that a general plan of the flow of the project could be identified from the beginning.

6.4 Problems encountered

It is foolish to think that any project will transpire without some sort of problems occurring. The difficulty lies in attempting to plan for these problems, and solve them when they inevitably show up. When working in a group, the problem domain increases from just technical problems, to also including social problems. Below we have discussed some of the most prevalent problems that occurred throughout the lifetime of the project, and how we solved these as a group.

Group Communications

Communication between group members is a vital part of team work, as it allows for the discussions of problems, progress, and projected progress. We were placed in randomly assigned groups, and as such had no way of communication other than our weekly meetings. We decided it would be beneficial for the group to create a Facebook page, to be used for group discussions, and to exchange mobile telephone numbers, so that we could contact each other in emergencies. In the second half of the project life time, we also implemented a kanban board online, which allowed to us to visualise and allocate all the tasks necessary to complete the project (A more in-depth analysis of our team working can be viewed in section 6.3 (*Managerial and team working*)).

Lack of specialised knowledge

It should come as no surprise that, as a group, we did not have a thorough understanding of the *R* programming language or fuzzy logic. I say this because, when working, you will often be given tasks of which you do not know the full domain space, and you must adapt your knowledge to these specific problems. As the project was near enough dependant on these two new subjects, it was vital that the group conducted appropriate research into what they were and how they worked. After some initial *R* tutorials that we found online, and using some resources provided by our supervisor, we slowly began understanding how to use *R*, and what fuzzy logic was. We used both the MATLAB fuzzy toolkit, and the source code already produced by the IMA group to help further advance our knowledge of the subject, whilst simultaneously writing our new code, to apply our new knowledge in a practical way. There were many quirky problems that we were subject to, due to the syntax and general operation of the *R* programming language, but due to the extensive resources available online, we were able to solve these problems with a little research and team work. We would often find ourselves meeting with our supervisor, or other members of the IMA group to clarify certain aspects of either fuzzy logic, or the *R* code, so that we could continue progressing.

Incomplete specification

At the beginning of the project, one of the members of the IMA group produced a brief specification of all the functions that needed to be produced for the fuzzy toolkit, from which we could begin our initial work. As we progressed further into the project, we began noticing that some parts of the specification were non-nonsensical or contradictory, and halted our progression. To resolve this, we organised a meeting with this same IMA group member, and worked through the “bugs” in the specification, so that all the functions followed the same basic layout, and were all compatible with each other.

Source control issues

As source control was a relatively new concept to most of the members of the group, and none of us had used source control for a project of this size before, some problems occurred in our code management. Most of these problems were accidental overwriting of other’s work, forgetting to add files to the repository before committing changes, and forgetting to put informative commit comments. Luckily, due to the very nature of our source control software (SVN), overwriting of code could be reverted and solved. The other two problems (forgetting to add files before committing or not adding comments to commits) were not really problems, but more mild set backs. Forgotten files could be added and committed in a separate commit, and comments could be added if a subsequent commit were made. Admittedly these were not ideal solutions, but mistakes happen, and they were not crippling mistakes.

7 Updated Project Time plan

The time plan laid out below is a list of all the tasks necessary to complete the project, with their associated deadlines attached. The idea of planning out the project as a goal driven project was to break the project into small sections, to make it less daunting to begin. We followed Susan Wilson’s 10 guidelines to goals that work [8], which allowed us to produce effective and realistic goals.

- 26/09/12 Provisional Group Allocation
- 01/10/12 Project Brief Received
- 02/10/12 Initial Formal Meeting with Supervisor
- 09/10/12 Allocation of roles within the group
- 16/10/12 Open/Close issues on Indefero
 - Check in and check out files from SVN repository
- 30/10/12 Upload project description to Indefero
 - Membership functions of the R Code
- 01/11/12 Upload all agendas and minutes to Indefero, upload any henceforth as well
- 02/11/12 **Hand In of Project Site**
- 06/11/12 “FIS” functions of the R Code (excluding *evalFis*)
 - Research into packaging of R
 - Preliminary system specification
- 13/11/12 File Input/Output functions of the R Code
- 20/11/12 Defuzzification function completed

27/11/12 First Draft of the Interim Report completed
 04/12/12 Polishing of the first draft into final draft completed
 10/12/12 **Hand in of the Interim Report**
 31/12/12 Re-write GUI to be new, standardised format
 31/01/12 Have R code completed, regardless of efficiency (excluding *evalFIS*)
 Plotting functions of the R code
 Printing functions of the R code
 Have GUI fully functioning, regardless of efficiency, and extra features
 04/02/13 Begin final group report
 06/02/13 *tipperTest* function completed
 10/02/13 Bug fixes and polishing of GUI and R
 13/02/13 Create interaction between R and GUI
 14/02/13 Completion of all R (excluding *evalFIS*) and all GUI (excluding graphs)
 05/03/13 First Draft of the final group report
 12/03/13 Polishing of R and GUI Code
 22/03/13 **Hand in of final report and software**
 23/04/13 Begin preparations for the open day and presentation
 08/05/13 **Open Day**
 10/05/13 **Presentation**
 PROJECT COMPLETION DATE

Words in text	8,554
Words in headers	171

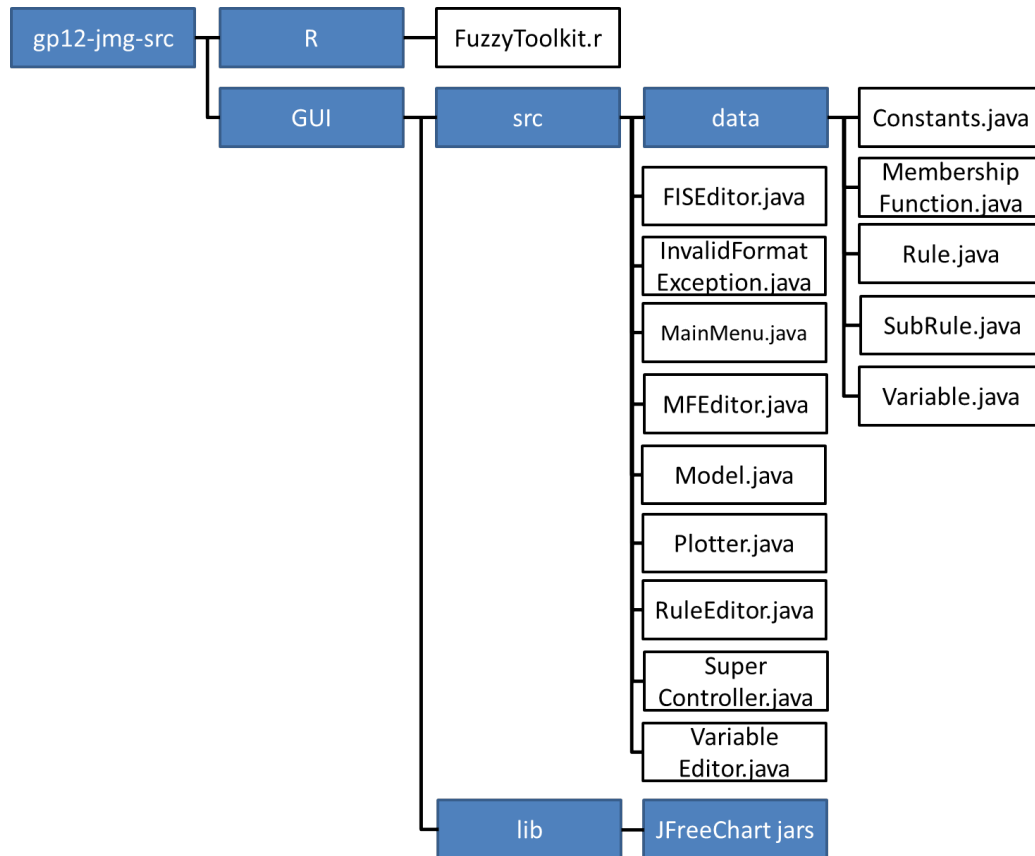
Calculated with the TeXCount web service
<http://app.uio.no/ifi/texcount/online.php>

References

- [1] Pedro Albertos and Antonio Sala. Fuzzy logic controllers. advantages and drawbacks. 1998.
- [2] Gregg Skip Bailey. Iterative methodology and designer training in human-computer interface design. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 198–205. ACM, 1993.
- [3] J. Lazar, A. Jones, and B. Shneiderman. Workplace user frustration with computers: An exploratory investigation of the causes and severity. *Journal - Behaviour & Information Technology*, 2006.
- [4] David J Osborne. *Ergonomics at work*. Publisher - Wiley, 1987.
- [5] Roger S Pressman and Darrel Ince. *Software engineering: a practitioner's approach*, volume 5. Publisher - McGraw-hill New York, 1992.
- [6] Patrick M Lencioni San Francisco. The five dysfunctions of a team: A leadership fable. 2006.
- [7] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. 1982.
- [8] Susan B Wilson and Michael S Dobson. *Goal Setting: How to Create an Action Plan and Achieve Your Goals*. Publisher - Amacom, 2008.
- [9] L.A. Zadeh. Is there a need for fuzzy logic? *Journal - Information Sciences*, 2008.

A Overview of the source code hierarchy

The source code is split into two distinct sections, the Fuzzy logic manipulation code, written in *R*, and the Graphical user input system, written in Java. The *R* code is included in the directory *gp12-jmg-src/R/FuzzyToolkit.r*, and the source code of the Java is included as a project exported from Eclipse, saved in the directory *gp12-jmg-src/GUI*, which includes all the source files, and the libraries necessary to use them. Included also in the *gp12-jmg-src/GUI* directory is a file *FISC.jar*, which is a standalone executable of the Java project.



B Extended testing appendix

B.1 Functional Requirements Testing

In this section I have listed all the functional requirements from our requirements specification, and where they can be found in our system. This testing done to ensure that the system fit the functional requirements that we had generated from our supervisors specification.

Users will be able to...	This can be found...
Create Gaussian functions	The membership function creator panel of the GUI, or the gaussMF function in the R
Create double Gaussian functions	The membership function creator panel of the GUI, or the gaussbMF function in the R
Create trapezoidal functions	The membership function creator panel of the GUI, or the trapMF function in the R
Create triangular functions	The membership function creator panel of the GUI, or the triMF function in the R
Evaluation membership functions	The evalMF function in the R code
Defuzzify membership functions	The defuzz function in the R code
Plot membership functions	The plotMF function in the R code, or the Variable editor panel of the GUI
Create FIS objects	The newFIS function in the R code, or the FIS editor panel of the GUI
Create input variables	The addVar function in the R code, or the inputs tab of the FIS editor panel of the GUI
Create output variables	The addVar function in the R code, or the outputs tab of the FIS editor panel of the GUI
Add membership functions to variables	Using the addMF function in the R, automatic in the GUI
Add manipulation rules	Using the logical rule builder in the GUI, or the addRule function in the code
Evaluate FIS structures	Using the evalFIS function in the R code
Load a FIS from a file	Using File->Load or the Load FIS button on the GUI, and readFIS in the R code
Save a FIS to a file	Using File->Save or File->Save As options in the GUI, or writeFIS in the R code
Print FIS objects	Using the showFIS function in the R code
Print Rules	Using the showRule function in the R code

B.2 R Test Cases

Membership Functions Related Testing		
Test	Expectations	Actual Results
The user can create a Gaussian function that will be stored in memory	A set of related data stored in a given object with the relevant name, range (from user input) and values (derived from both range and input parameters).	Expectations met exactly. The given name was stored, as was the given range. The membership function type was stored correctly, and the derived values were accurate.
The user can create a Gaussian Bell function that will be stored in memory	A set of related data stored in a given object with the relevant name, range (from user input) and values (derived from both range and input parameters).	Expectations met exactly. The given name was stored, as was the given range. The membership function type was stored correctly, and the derived values were accurate.
The user can create a Trapezoidal function that will be stored in memory	A set of related data stored in a given object with the relevant name, range (from user input) and values (derived from both range and input parameters).	Expectations met exactly. The given name was stored, as was the given range. The membership function type was stored correctly, and the derived values were accurate.
The user can create a Triangular function that will be stored in memory	A set of related data stored in a given object with the relevant name, range (from user input) and values (derived from both range and input parameters).	Expectations met exactly. The given name was stored, as was the given range. The membership function type was stored correctly, and the derived values were accurate.
An empty string is given as a name in the creation of any membership function.	An error is output to console, alerting the user that they have entered an empty string and stopping the membership function creation from continuing.	Expectations met, console outputs appropriate message.
An illegal character is given as part of the input parameter for name.	An error is output to console, alerting the user that they have given an illegal character in the name string and stopping the membership function creation from continuing.	Expectations met exactly, console outputs appropriate message.

FIS Structure Related Testing		
Test	Expectations	Actual Results
User is able to create a FIS with default parameters providing just a name.	A FIS structure with the default settings is stored in memory with the given name.	Expectations met exactly. FIS successfully created and stored.
User is able to create a FIS with specified input parameters.	A FIS structure with the specified user input parameters is stored in memory.	Expectations met exactly. FIS successfully created and stored.
User adds an input variable to an existing FIS structure without any input variables.	The FIS will store the new input variable at the first index of the list which holds input variables and their data.	Expectations met exactly. The input variable was successfully stored in the first index location of the list.
Users adds multiple input variables to an existing FIS structure which has one or more input variables already.	The input variables will be appended to the list, as in, their index value will be the increment of the last variable.	Expectations met exactly. New input variables are always appended to the end of the list, ensuring data is not accidentally erased.
User adds an output variable to an existing FIS structure without any output variables.	The FIS will store the new output variable at the first index of the list which holds output variables and their data.	Expectations met exactly. The output variable was successfully stored in the first index location of the list.
Users adds multiple output variables to an existing FIS structure which has one or more output variables already.	The output variables will be appended to the list, as in, their index value will be the increment of the last variable.	Expectations met exactly. New output variables are always appended to the end of the list, ensuring data is not accidentally erased.
User adds a rule definition to an existing FIS structure without any rules.	The rule will be stored as a vector in the first index of the list.	Expectations met exactly, the rule is stored at the first index location of the matrix holding the rule vector variables.
User adds multiple rules to an existing FIS structure with one or more rules.	The new rules will be appended/binded to the increment of the existing rule's index location in the matrix.	Expectations met exactly, when rules are added they are appended/binded to the end of the matrix, ensuring nothing is ever overwritten accidentally.
User adds a membership functions to a specified input variable without any existing membership function objects stored.	The membership function object will be stored in the specified variable, at the first index location of the list.	Expectations met, the given membership function is stored at the given input variable.
Users adds multiple membership functions to a specified input variable with one or more existing membership functions.	The membership functions will be appended to the end of the input variable's list which holds membership functions.	Expectations met, membership functions are stored incrementally ensuring no data is ever overwritten.

FIS Structure Related Testing (Continued)		
Test	Expectations	Actual Results
User adds a membership functions to a specified output variable without any existing membership function objects stored.	The membership function object will be stored in the specified variable, at the first index location of the list.	Expectations met, the given membership function is stored at the given output variable.
Users adds multiple membership functions to a specified output variable with one or more existing membership functions.	The membership functions will be appended to the end of the output variable's list which holds membership functions.	Expectations met, membership functions are stored incrementally ensuring no data is ever overwritten.
User gives a FIS structure an empty string for a name.	An error is output to console informing the user they have not provided a name for the FIS structure upon creation, which is then stopped.	Expectations met, appropriate message output.
User gives a FIS structure a string with illegal characters for a name.	An error is output to console informing the user that they have entered an illegal character upon creation, and the creation of the FIS structure is stopped.	Expectations met, appropriate message output to console.

I/O Related Testing		
Test	Expectations	Actual Results
When user writes a FIS to a file, the filename is given without a ".fis" extension.	The code should automatically handle this and take the string given from filename and simply concatenate it with another string consisting of ".fis" and store it back into the filename variable.	Expectations met, string concatenated and file stored at given location.
When user reads a FIS from a file, the filename is given without a ".fis" extension.	The code should automatically handle this and take the string given from filename and simply concatenate it with another string consisting of ".fis" and store it back into the filename variable.	Expectations met, string concatenated and file with the new filename is passes as the parameter.
User enters illegal character in filename parameter for either the reading function or the writing function.	An error message informing the user that have entered an illegal input character is output to console, and the function is stopped from further execution.	Expectations met, appropriate message is output.
User tries to read a FIS from a file which does not exist.	An error message is output to console informing the user that the file does not exist and further execution stops.	Somewhat successful, the message is generated to the console, but it is from an inbuilt R function and is somewhat unclear.
User writes a FIS to a file.	The data from memory is stored with the relevant syntax in a .fis extension file.	Expectations met, a file was generated in the given directory with all the data stored in the exact format used by both the R code and Java code.
User reads a FIS from a file.	The data stored in the file with a .FIS extension from the specified directory is translated accordingly into a FIS which is stored in memory.	Expectations met, a FIS structure is generated from the file with accurate values.
User uses the showing function	All the data concerned with the given FIS should be displayed in an ordered manner.	Expectations met, the FIS is shown clearly and concisely via the console in a visually comprehensible manner.

Plotting Related Testing		
Test	Expectations	Actual Results
User plots the membership functions of an existing input variable	A visual graph display will initiate, with the all the membership functions associated with the given variable plotted accordingly.	Expectations met, the graph is rendered with accurate data.
User plots the membership functions of an existing output variable	A visual graph display will initiate, with the all the membership functions associated with the given variable plotted accordingly.	Expectations met, the graph is rendered with accurate data.
User specifies a non-existing variable accidentally with a given variable index parameter.	The R will output a clear error message informing the user it does not exist, and stop any further execution of the function.	Somewhat met the expectations, but an inbuilt R-generated error is instead produced which is less clear than anticipated.
The plotting function is used with Gaussian and Gaussian Bell membership functions.	The plots rendered will be joined via a curved line instead of a series of straight lines for these types of membership functions only.	Expectations met, the plots are rendered with joining curved lines.
The user uses the plotting function to test the colour system by giving it working values.	The plotted lines will all have different colours from one another, and be marked with a legend for a user's visual clarity.	Expectations met, all lines are rendered with different colours and a legend showing each individual membership function's name is output above the graph area.
Miscellaneous functions		
Test	Expectations	Actual Results
User chooses to defuzzify a given membership function using the defuzz function.	A defuzzified value is returned (differing depending on the given type).	Expected result met.
User enters an unknown defuzzification type for the defuzz method	An error message is output to console informing the user they have entered an unsupported defuzzification function.	Expected outcome met, appropriate console message output.
User evaluates a given membership function (for testing purposes only, typically a user will never use this function by itself).	A membership function is returned with the calculated values depending on the given type.	Success, an accurate membership function is returned.

B.3 GUI Test Cases

Main Menu		
Test	Expectations	Actual Results
The user will be able to create a new FIS	There will be an option to create a new FIS	“New FIS” button is present, and will take the user to a blank FIS editor to construct their new FIS
The user will be able to load a FIS from file	There will be an option to load a FIS	“Load FIS” button is present, and will take the user to a file chooser, from which they can select and load their desired FIS, which will be opened in the FIS editor
The user will be able to close the system	There will be an option to exit the system	The user can click the “X” at any point, or the “Close” button

Membership Function Creator		
Test	Expectations	Actual Results
The user can name membership functions	There will be an input field to enter membership function name	There is a labelled text field that allows the user to enter the name of the function
The user can create Gaussian curves	There is a way to specify and create Gaussian functions	There is a labelled combo box, from which users can select Gaussian, which will generate labelled input boxes that enumerate the parameters required
The user can create Gaussian B curves	There is a way to specify and create Gaussian B functions	There is a labelled combo box, from which users can select Gaussian B, which will generate labelled input boxes that enumerate the parameters required
The user can create trapezoidal membership functions	There is a way to specify and create trapezoidal functions	There is a labelled combo box, from which users can select trapezoidal, which will generate labelled input boxes that enumerate the parameters required
The user can create triangular membership functions	There is a way to specify and create triangular functions	There is a labelled combo box, from which users can select triangular, which will generate labelled input boxes that enumerate the parameters required
The user can specify the specific parameters for each membership functions	There is input boxes for the function parameters	After selecting a function type, input fields are automatically generated and labelled, and the user can enter the required parameters

FIS Editor		
Test	Expectations	Actual Results
The user will be able to create a new FIS	There will be a way to create blank FIS objects	There is a File option, "New", which will erase the user's FIS and launch a fresh FIS Editor
The user will be able to load a FIS from file	There will be a way to load a FIS from a file	There is a File option, "Load", which launching a file browser, from which the user can select a FIS file to be displayed on the FIS editor
The user will be able to save the current FIS	There will be the ability to save FIS objects	There is a File option, "Save", which saves the current FIS to the disk (if it already exists)
The user will be able to save a copy of the current FIS	There is a way of saving the file with a new name	There is a File option, "Save As", which saves the current FIS to the disk after the user specifies a name
The user will be able to specify a FIS name	There will be a way of naming the FIS	There is a labelled text field that the user can enter the name of the FIS into
The user will be able to specify FIS method parameters	There will be an input that allows the user to select and, or, agg, imp and defuzz methods for the FIS	There are labelled combo boxes for each FIS method, which a list of possible options
The user will be able to create input variables	There will be a way to do this	There is an "Inputs" panel, with a "New" button on it, which launches the variable editor
The user will be able to edit input variables	There will be a way to do this	Next to each variable is an "Edit" button, which launches the variable editor with the values stored in the selected variable
The user will be able to delete input variables	There will be a way to do this	Next to each variable is a "Delete" button, which removes the selected variable
The user will be able to create output variables	There will be a way to do this	There is an "Outputs" panel, with a "New" button on it, which launches the variable editor
The user will be able to edit output variables	There will be a way to do this	Next to each variable is an "Edit" button, which launches the variable editor with the values stored in the selected variable
The user will be able to delete output variables	There will be a way to do this	Next to each variable is a "Delete" button, which removes the selected variable
The user will be able to create rules	There will be a way to do this	There is a "Rule" panel, with a "New" button on it, which launches the rule editor
The user will be able to edit rules	There will be a way to do this	Next to each rule is an "Edit" button, which launches the rule editor with the values stored in the selected rule
The user will be able to delete rules	There will be a way to do this	Next to each rule is a "Delete" button, which removes the selected rule

Variable Editor		
Test	Expectations	Actual Results
The user can specify a name	There is an input box for the variables name	There is a labelled text field that allows for entry of variable name
The user can specify a minimum bound	There is an input box for the minimum range	There is a labelled text field that allows for the entry of bounds
The user can specify a maximum bound	There is an input box for the maximum range	There is a labelled text field that allows for the entry of bounds
The user can add membership functions	There is an option to create membership functions	There is a “Membership functions” panel, in which is a “New MF” button, which launches the membership function editor
The user can edit added membership functions	There is an option to edit membership functions	Next to each created membership function is an “Edit” button that launches the membership function editor and loads in the values of the selected membership function
The user can delete added membership functions	There is an option to delete membership functions	Next to each created membership function is a “Delete” button that removes it from the system
The user can plot membership functions	There is the ability to display membership functions graphically	Once at least one membership function has been created, a graph will be plotted with all membership functions of the variable

Compatibility between screens		
Test	Expectations	Actual Results
Information can be passed between the rule editor and the FIS editor	There is some storage mechanism that allows data to flow between these two screens	Rules created are retrieved from the rule editor instance they were created in, and stored in the data model of the system
Information can be passed between the variable editor and the FIS editor	There is some storage mechanism that allows data to flow between these two screens	Variables created are retrieved from the variable editor instance they were created in, and stored in the data model of the system
Information can be passed between the variable editor and the membership function editor	There is some storage mechanism that allows data to flow between these two screens	Membership functions created are retrieved from the membership function editor instance they were created in, and stored in an array list object in the variable instance it is being added to

Rule Creator		
Test	Expectations	Actual Results
Users can input values for each input variable in the system	There is some way of specifying a membership function for each input variable created	Each variable has a combo box of each of its membership functions, from which users can select
User can input values for each output variable in the system	There is some way of specifying a membership function for each output variable created	Each variable has a combo box of each of its membership functions, from which users can select
User can specify a logical connective	There is some way of specifying the connective of the variables	There is a set of radio buttons that allow the user to select one of two connectives
Users can specify weight of the rule	There is some way of specifying the weight of the rule	There is a text field that allows the user to enter a value of between 0 and 1 for the weight
Users can invert any input or output variable	There is some way of toggling the inverted and non inverted status of each variable	There is a check box under each variable that can be toggled to show inversion

B.4 User Trials

To get some real user feedback on the graphical user interface, we decided to host a user trial. In this, one non-computer science student was given the system, and a list of tasks was dictated for them to complete. We hoped that this would raise any potential issues that low and middle level users would have with the system, which we could then rectify. The task to complete was to use the GUI to create an entire working FIS object, load this into the R code, and then conduct some basic plotting. A complete listing of each task, the actions taken by the user, and what information this provided can be seen below.

- 1. Create a new FIS object**

Trivial selection of “New FIS”

- 2. Enter the name, “TipperTest”**

Trivial entry of name

- 3. Create a new input variable**

User was slightly confused as to what an input variable actually was, but looking around the screen they quickly noticed the “Input Variables” header, and the large “New” button under it, which they then pressed.

- 4. Name the variable “Food”, and put the range as 0 and 10**

Slight confusion as which box symbolised the minimum and which symbolised the maximum, but assumed them to be in order. Trivial entry of data.

- 5. Add a Gaussian Curve, with name “Poor”, sigma 1.5, mean 0, and height 1**

User did not know what a Gaussian curve (limited by knowledge of fuzzy logic), so we had to instruct that this was a membership function. At this point they understood to click the “New MF” button. They asked for clarification if “Gaussian B” was the same as “Gaussian” (limited by knowledge of fuzzy logic again, not by the system). After selecting the correct option from drop down, parameters were entered successfully.

6. **Add a Gaussian Curve, with name “Good”, sigma 1.5, mean 5, and height 1**
Trivial repetition of prior task
7. **Add a Gaussian Curve, name “Excellent”, sigma 1.5, mean 10, and height 1**
Trivial repetition of prior task
8. **Add an output variable, with name “Tip”, and range 0 to 30**
User attempted to add a second input variable, before realising their mistake. Navigated to the “Outputs” panel, and created the variable with no troubles.
9. **Add a triangular membership function, with name “Cheap”, left 0, mean 7, right 14 and height 1**
User recalled how to do this from prior task
10. **Add a triangular membership function, with name “Average”, left 7, mean 14, right 21 and height 1**
Trivial
11. **Add a triangular membership function, with name “Generous”, left 17, mean 24 right 30 and height 1**
User accidentally forgot to add one parameter. Presented with error message. When asked for their opinion on the error message, they said it was friendly and informative, but the exclamation mark was a little scary. They also said it was annoying that they had to enter in *all* the values again.
12. **Create a rule, “If food is excellent then tip is average”**
Accessing the rule editor was a simple task. However the user was slightly confused with the concept of “if then” statements. They did however manage to work out how to create the rule without help, but stated that prompts (like “IF” before inputs, and “THEN” in between the inputs and outputs panels) would have been more useful.
13. **Save the file as “TipperTest”**
User didn’t use the CTRL+S shortcut, as they said they didn’t expect it to be in this system. They simply clicked “File”, and then “Save As”.
14. **Load the file into R**
User was unsure how to actually use the R interface. This is something we cannot help, and our system cannot be designed in a way to help teach *R* or the user of it’s interface. After opening the console for the user, and loading the fuzzy toolkit source file, we then showed them a list of all the functions that were in the system, and asked them to pick which would be most appropriate to load the file in. They correctly identified the readFIS function.
15. **Plot the membership functions of the “Food” variable**
Once again the user was shown a list of the functions in the system. They deduced that the function to use was plotMF as this had the word “plot” in it, and they were told to plot a membership function. They looked at the source code for a moment or two, at which point they were able to deduce to use the FIS we had created with readFIS, the word “input”, and the variable they wanted to use (as there was only one, they put 1). This then drew the graphs representing the “Food” variable.

We asked our user if they would prefer to use the GUI to create FIS objects, or the *R* code. They said they (as a mid/low level user) would much prefer the GUI system, but appreciated that more advanced users could make use of the *R* code. They said that the GUI was simple to user, and designed well. They were not really sure what they were doing due to not knowing anything about fuzzy logic, but felt as though the system had enough labels and guides that finding their way around, and following our instructions was easy. They said they found the *R* very difficult to use, but not because it was badly designed, just because they struggled with the command line interface. With some practice they said they could see themselves adopting it very well, and that the code was commented well enough to facilitate this.

What we managed to learn about our system is that there are two limiting factors. How well the user understands fuzzy logic (which is a limiting factor we mentioned in our requirements, and is perfectly acceptable), and how well the user understands command line interfaces. Due to the scope of the project, we did not have the time or resources to expand the *R* command line interface to be more user friendly, and unfortunately could do nothing about this. If we were to complete the project again, we could perhaps migrate all of the functionality of the *R* code to the GUI, to make a complete GUI package, that allowed the user to easily create and manipulate fuzzy logic sets. However, our supervisor only wanted the core functionality to be in the *R* code, as this was what he and the IMA group would be using to process fuzzy sets, and they did not want to have to launch a large GUI system to do so.

C User Manuals

C.1 R Fuzzy Tool-Kit

R *Fuzzy* TOOL-KIT USER MANUAL

Preface

This is a short user manual produced to guide the user through any actions that they wish to partake in whilst using the *R* fuzzy manipulation system we created. It covers all aspects of the system, giving code samples, pictures where necessary, explanations of what you can achieve and, more importantly, how you can achieve this. Due to this being included as an appendix of our final group, the formatting is slightly different to what the user manual would actually be like. It is also worth noting that all of the source code for our *R* project is appropriately commented, so that each function explains what the inputs are, what the outputs are, and what processes it goes through.

The basics

Before using any of the functions available in the tool kit, we must load it into our interpreter. To do this, you can either drag and drop the file directly from a folder into the *R* interpreter window, or you can type;

```
> source("path\\FuzzyToolkit.r")
```

Where *path* is the file path of wherever you have saved the fuzzy tool kit.

We also need to quickly look at the vector data structure. A vector is simply a list of items, for instance numbers (1,2,3) or characters (a,b,c). Vectors are used throughout our system for storing the values of the necessary data structures, and as such, a brief example of the creation of a vector can be seen below.

```
> a <- c(1,2,3)
> a
[1] 1 2 3
```

As you can see here, we create a vector *a*, with the values 1,2 and 3 in it. The “c” in front of our brackets simply means the *combination* of the values inside. The $<-$ is the assignment operator, which simply assigns the variable on the left to be the values on the right. In this case, we have assigned the value of *a* to be the vector (1,2,3).

This assignment operator will be vital for the majority of the objects we will be creating in the system later.

FIS creation

A FIS (or Fuzzy Inference System) is the basic data structure that is used for fuzzy logic manipulation. This is a required object for evaluation of fuzzy logic systems; it consists of system parameters, variables and rules. Creating a FIS is generally the first thing that should be done, as all other data will be adding to this FIS. For this, we require the use of a function *newFIS()*.

```
> FIS1 <- newFIS("MyFirstFIS")
```

This will now have created a new FIS, which can be referenced by the name "FIS1". The name of the FIS itself, will be "MyFirstFIS".

```
> FIS1
$name
[1] "MyFirstFIS"
$type
[1] "mamdani"
$version
[1] "1.0"
$andMethod
[1] "min"
$orMethod
[1] "max"
$impMethod
[1] "min"
$aggMethod
[1] "max"
$defuzzMethod
[1] "centroid"
$inputList
NULL
$outputList
NULL
$ruleList
NULL
```

This will now have created a new FIS, which can be referenced by the name "FIS1". The name of the FIS itself, will be "MyFirstFIS".

This FIS has a set of default values. In order to specify specific methods, simply enumerate the methods you wish to include as parameters to the function. The order of parameters for a FIS is as follows; *FISName*, *FISType*, *andMethod*, *orMethod*, *impMethod*, *aggMethod* and finally *defuzzMethod*. The code snippet below shows this

```
> FIS2 <- NewFIS("MySecondFIS", "mamdani", "min",
                 "max", "min", "max", "centroid")
> FIS2
$name
```



```

[1] "MySecondFIS"
$type
[1] "mamdani"
$version
[1] "1.0"
$andMethod
[1] "min"
$orMethod
[1] "max"
$impMethod
[1] "min"
$aggMethod
[1] "max"
$defuzzMethod
[1] "centroid"
$inputList
NULL
$outputList
NULL
$ruleList
NULL

```

Membership function creation

There are four different membership functions that can be created in this version of the fuzzy tool kit. These are Gaussian curves, double Gaussian curves, trapezoidal graphs, and triangular graphs.

```
> g1 <- gaussMF("Gaussian", 0:10, c(1,1,2))
```

The first of the membership functions, the Gaussian curve, is created as shown above. The first parameter specified is the name of the membership function, "Gaussian", followed by the range it lies between, and the vector at the end is a collection of the parameters of the function. The three values necessary for a Gaussian curve are "Sigma", "Mean", and "Height", specified in that order.

```
> g2 <- gaussbMF("GaussianB", 0:10, c(4,5,1,4,1))
```

Next we have the double Gaussian curve, named GaussB. The first parameter specified is the name of the membership function, "GaussianB", followed by the range it lies between, and the vector at the end is a collection of the parameters of the function. The five values necessary for a Gaussian curve are "Left_Sigma", "Left_Mean", "Right_Sigma", "Right_Mean" and "Height", specified in that order.

```
> t1 <- trapMF("Trapez", 0:10, c(2,6,1,4,1))
```

The third function we can make is the trapezoidal membership function. The first parameter specified is the name of the membership function, "Trapez", followed by the range it lies between, and the vector at the end is a collection of the parameters of the function. The five values necessary for a Gaussian curve are "Left_Foot", "Left_Shoulder", "Right_Shoulder", "Right_Foot" and "Height", specified in that order.

```
> t2 <- triMF("Triangle", 0:10, c(4,5,1,1))
```

The final function present in this version is the triangular membership function. The first parameter specified is the name of the membership function, "Triangle", followed by the range it lies between, and the vector at the end is a collection of the parameters of the function. The four values necessary for a Gaussian curve are "Left", "Mean", "Right" and "Height", specified in that order.

Once a membership function has been created, it can be added to a variable. Creation of variables is discussed below, and the process of adding membership functions to variables is described in that section.

Creating and adding variables to a FIS

Once a FIS has been created, it is necessary to create input and output variables so that the system can be evaluated. These variables consist of a name, a range, and a set of membership functions (creation of which is explained in the previous section). The function used to create a new variable and add it to a FIS is *addVar*, which requires four parameters. These parameters are; the FIS that this variable is to be added to, a flag to state whether the variable is an input or output, a name for the input, and a range that the membership functions within the variable must adhere to. The FIS you are adding the variable will be "assigned" to itself with the new variable added. This is a confusing statement, and is best displayed.

```
> TestFIS <- newFIS("TestFIS")
> TestFIS <- addVar(TestFIS, "input", "VariableName", (0:10))
```

This code achieves two things, the first of which is the creation of a new FIS object referenced and named "TestFIS". The second is to assign TestFIS to be itself, with the addition of a new variable, that is defined in the addVar function. The variable defined above will be assigned to TestFIS, be an input (as opposed to an output, if the string "output" were specified), have the name "VariableName", and have a range of between 0 and 10. We can view this in the *R* console, by simply writing *TestFIS*.

```
> TestFIS
$inputList
$inputList[[1]]
```

```

$inputList [[1]] $inputName
[1] "VariableName"

$inputList [[1]] $inputBounds
[1] 0 1 2 3 4 5 6 7 8 9 10

```

I have cut some of the output out, but you can see that our new variable has been added into the list of inputs, with the name and range that we specified. If we were to repeat this process, but instead specify “output”, instead of “input” for the second parameter, the variable would be added as an output, and *TestFIS* would look slightly different.

```

> TestFIS <- newFIS("TestFIS")
> TestFIS <- addVar(TestFIS, "output", "NewVariable", (0:10))
> TestFIS

$outputList
$outputList [[1]]
$outputList [[1]] $outputName
[1] "NewVariable"

$outputList [[1]] $outputBounds
[1] 0 1 2 3 4 5 6 7 8 9 10

```

As mentioned in the last section, membership functions can be added to variables, and they must if you wish for the system to function correctly. This is a simple process, and is much like adding a variable to a FIS. First of all you must have a valid FIS and a valid input or output variable added to this FIS. The code below goes through the process of adding membership functions.

```

> TestFIS <- newFIS("TestFIS")
> TestFIS <- addVar(TestFIS, "input", "NewVariable", (0:10))
> GaussianCurve <- gaussMF("FirstMF", (1:10), c(1.5,5,1))
> TestFIS <- addMF(TestFIS, "input", 1, GaussianCurve)

```

The code above firstly creates a new FIS object, named and referenced as “TestFIS”. It then assigns a new variable to TestFIS, called “NewVariable”, which an input between 0 and 10. We then create our membership function “GaussianCurve”, which is a Gaussian Curve named “FirstMF”, with parameters 1.5, 5 and 1. Finally we add the membership function to our FIS. This is achieved with the *addMF* function, which assigns TestFIS to being itself with a new membership function. The function *addMF* requires four parameters; the FIS it is being added to, whether or not it is an input or output, the index of the variable it is being added to (the first variable being signified as “1”), and the membership function that is actually being added.

```

> TestFIS

$inputList
$inputList [[1]]

```

```

$inputList [[1]] $inputName
[1] "NewVariable"

$inputList [[1]] $inputBounds
[1] 0 1 2 3 4 5 6 7 8 9 10

$inputList [[1]] $membershipFunctionList
$inputList [[1]] $membershipFunctionList [[1]]
$inputList [[1]] $membershipFunctionList [[1]] $mfName
[1] "FirstMF"

$inputList [[1]] $membershipFunctionList [[1]] $mfType
[1] "gaussMF"

$inputList [[1]] $membershipFunctionList [[1]] $mfX
[1] 1 2 3 4 5 6 7 8 9 10

$inputList [[1]] $membershipFunctionList [[1]] $mfParams
[1] 1.5 5.0 1.0

```

Creating logical rules

To govern the evaluation of fuzzy inference systems, we need a set of rules. These rules are created as a simple vector; how this is done is explained in this section. It is worth noting that this can be a tedious process to complete in the *R* manipulation system, and is much easier to complete in the FIS Construction System that is packaged with the *R* system. Once a number of variables have been added (both input and output) rules can be created. Rules have the following format.

$$(input_1, input_2, input_n), (output_1, output_2, output_m) (weight) : connective$$

Where $input_x$ is the value of the x th input variable, $output_y$ is the value of the y th output variable, $weight$ is the weight of rule, between 0 and 1, and the connective is either 1 (*AND*) or 2 (*OR*). That is to say that a rule will consist of Z parameters, where Z is equal to the number of input variables plus the number of output variables, plus two (weight and connective). I understand that this may be confusing (hence my suggestion to use the graphical user interface for logical rule building), but I will try to explain this further with an example.

Say that we have created a new FIS that has two input variables and one output variable. Instantly we can say that any rule created for this system will require $2 + 1 + 2 = 5$ parameters. The code for creating this system is given below.

```

> TestFIS <- newFIS("TestFIS")

> TestFIS <- addVar(TestFIS, "input", "Food_Quality", (0:10))
> inFoodQ1 <- gaussMF("Delicious", (1:10), c(1.5,5,1))

```

```

> inFoodQ2 <- gaussMF("Rancid", (1:10), c(1.5,5,1))
> TestFIS <- addMF(TestFIS, "input", 1, inFoodQ1)
> TestFIS <- addMF(TestFIS, "input", 1, inFoodQ2)

> TestFIS <- addVar(TestFIS, "input", "Service_Quality", (0:10))
> inServiceQ1 <- gaussMF("Excellent", (1:10), c(1.5,5,1))
> inServiceQ2 <- gaussMF("Mediocre", (1:10), c(1.5,5,1))
> inServiceQ3 <- gaussMF("Poor", (1:10), c(1.5,5,1))
> TestFIS <- addMF(TestFIS, "input", 2, inServiceQ1)
> TestFIS <- addMF(TestFIS, "input", 2, inServiceQ2)
> TestFIS <- addMF(TestFIS, "input", 2, inServiceQ3)

> TestFIS <- addVar(TestFIS, "output", "Tip", (0:10))
> outTipQ1 <- gaussMF("High", (1:10), c(1.5,5,1))
> outTipQ2 <- gaussMF("Medium", (1:10), c(1.5,5,1))
> outTipQ3 <- gaussMF("Low", (1:10), c(1.5,5,1))
> TestFIS <- addMF(TestFIS, "output", 1, outTipQ1)
> TestFIS <- addMF(TestFIS, "output", 1, outTipQ2)
> TestFIS <- addMF(TestFIS, "output", 1, outTipQ3)

```

The system consists of an input “Food Quality”, with membership functions of “Delicious” and “Rancid”, and a second input “Service Quality” with membership functions of “Excellent”, “Mediocre” and “Poor”. There is also an output function of “Tip” which has three membership functions, “High”, “Medium” and “Low”. Now that we have a FIS system, we can create the rules for it. For instance, say I wished to create the following rule

$$\text{If } (Food\ Quality = Delicious \vee Service\ Quality = Excellent) \rightarrow Tip = High$$

There are two things we know already, we need a vector, and this vector needs to be of length five. Now, for each of the inputs and outputs, we need to specify which of the membership functions they are referring to. For example, *Food Quality = Delicious* is the “Food Quality” variable referring to the “Delicious” membership function, which is the first membership function of that variable. This is then signified by the number 1. The values for service quality and tip can also be calculated in the same way. “Excellent” is the first membership function for “Service Quality”, and “High” is the first membership function for “Tip”. This means the values we require so far are 1, 1, and 1. For reference, a 0 would mean no membership function value selected, which would imply the specific variable is not included in the rule. Now, we have our first three values, but still require a value for weight and connective. This is as simple as specifying two more numbers, we give the weight a value of 1 as well, and we give the connective a value of 2. The reason for the connective being 2, is that 2 represents “OR” (which is what we mean with the \vee symbol), as opposed to 1, which represents “AND”. Our rule is now as follows

1 1 1 1 2

Which means the first membership function of the first input, the first membership function of the second input, the first membership function of the first output, with a weight of one, and connected by logical “OR”. We are now ready to add our rule to the system.

```
> rule1 <- c(1,1,1,1,2)
> TestFIS <- addRule(TestFIS, rule1)
```

This has the effect of first creating a rule (which is just a vector), and assigning TestFIS to itself, with this new rule added.

```
> TestFIS

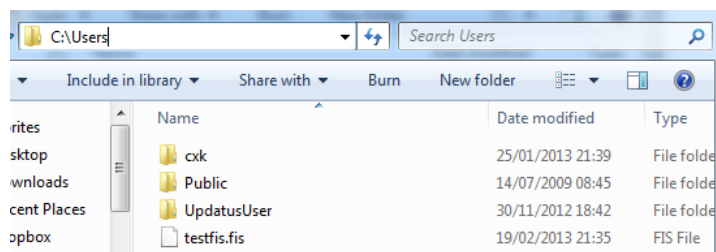
$ruleList
      [,1] [,2] [,3] [,4] [,5]
[1,]      1      1      1      1      2
```

Reading and writing to file

FIS objects can be saved or read in from file, this is especially useful if you use the graphical FIS constructor system to create your FIS objects, as these can then be read in and manipulated (which is much quicker than creating the FIS inside of the *R* code). Writing files to the disk is a simple function call, using *writeFIS()*, which is demonstrated below.

```
> writeFIS(TestFIS, "C:\\Users\\testfis.fis")
```

Where you are free to put any file path as the second parameter. Please remember that this must be an absolute address, and you need to specify the file name and type of the FIS object.



Similarly with writing files from the system, we can also read files into the system, to then manipulate. When we read a FIS in, we need to assign it to a variable, otherwise it will not be later usable.

```
> fisFromFile <- readFIS("C:\\Users\\testfis.fis")
```

Remember that the path you give must be absolute, including both the file name and the file type (which will be ".fis"). Any FIS object exported by either the *R* code using the method above, or from the graphical FIS construction system can be read into the system, as long as the files have not been tampered with or corrupted.

Evaluation of FIS structures

FIS structures, once created using the methods detailed above (or the graphical FIS construction system) can be evaluated with the *R*, to return crisp values, for fuzzy information. There are several different methods of evaluation that can be applied to the FIS structures within the *R* code, and these are discussed below.

```
> g1 <- gaussMF("Good", 0:10, c(1,1.5,5))
> defuzz <- (g1, "centroid")
```

This is the defuzzification function, which will return a crisp value for a given membership function. The *defuzz* function takes two parameters, a membership function to be evaluated, and a type of defuzzification to use. The supported defuzzification methods in the *R* system are; *centroid*, *bisector*, *mom*, *som* and *lom*.

```
> testFIS <- newFIS("testFIS")
> testFIS <- addVar(testFIS, "input", "testVar", 0:10)
> g1 <- gaussMF("GaussF", 0:10, c(1.5,5,1))
> testFIS <- addMF(testFIS, "input", 1, g1)
> plotMF(testFIS, "input", 1)
```

The *plotMF* function is a slight modification of the default *R plot* function, which allows for the specific plotting of a number of membership functions. Only membership functions that have been added to a variable within a FIS can be plotted. This is done with the *plotMF* function with three parameters, the FIS object, whether or not it is an input or output variable and the index of the membership function.

```
> testFIS <- newFIS("testFIS")
> testFIS <- tippertest()
> gensurf(testFIS)
```

The final function *gensurf*, which uses the output of *evalfis* is a function that plots a 3D surface of two inputs against an output. This produces a visual representation of the system as a whole.

C.2 Java Graphical Input System

FUZZY INFERENCE SYSTEM CONSTRUCTOR USER MANUAL

Preface

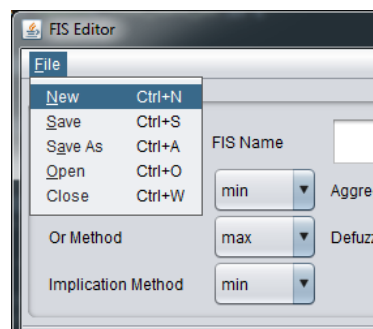
This is a short user manual produced to guide the user through any actions that they wish to partake in whilst using the FIS Construction System. It covers all aspects of the system with explanations of what you can achieve and, more importantly, how you can achieve this. Due to this being included as an appendix of our final group, the formatting is slightly different to what the user manual would actually be like, we apologise for this.

Creating and specifying a new FIS

There are two methods to creating a new FIS, dependant on where in the system you currently are. For users that have just launched the system, and are faced with the main menu (pictured below), creation of a new FIS is as simple as clicking the “New FIS” button that is present.



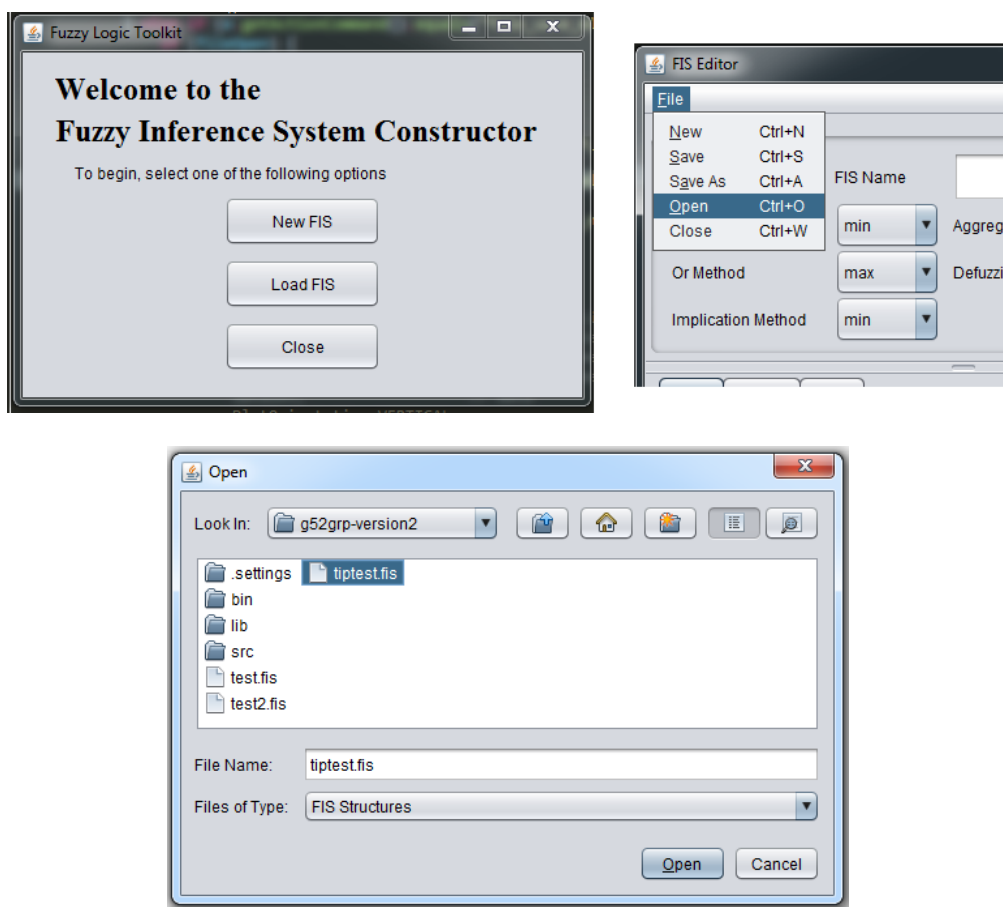
For users that are on the main FIS editor screen, creating a new FIS requires you to click on the “File” menu option, followed by “New” (this can be achieved by pressing *CTRL* followed by *N*).



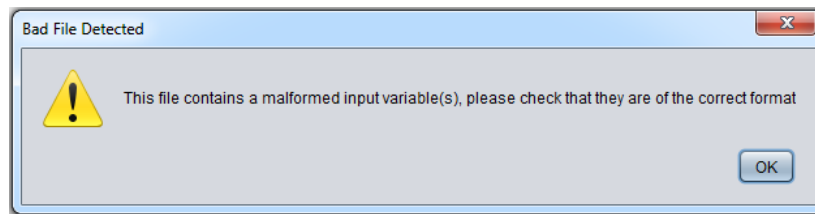
Both of these options will present the user with new FIS object, with default values for evaluation methods, and a blank name field. These can be changed at the user's leisure, dependant on how their FIS is to be manipulated, and what they want to call it (please note that including a name is mandatory).

Loading a FIS from file

Loading a FIS from a file (including those produced by the *R* manipulation system) can be done through the "Load FIS" button present on the main menu screen of the system, or clicking "File" and "Open" on the menu of the FIS editor window (this can also be achieved by pressing *CTRL* followed by *O*). Both of these options will bring up a dialogue box, which will allow you to navigate to, and load, any ".fis" file. Any files created by either the FIS constructor, or the *R* manipulation system will be valid for loading into the system.

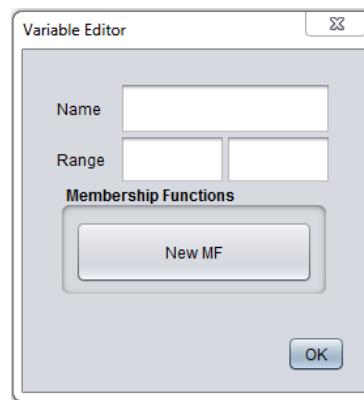


This will load all values saved in that ".fis" file into the system; this includes all parameters, variables, membership functions and rules. If any part of the loaded file is invalid, or of an incorrect format, an error message will be displayed, identifying the offending section.



Adding variables and membership functions to a FIS

Once a FIS is loaded or created, variables can be added to it. Each variable is constructed of multiple membership functions, and cannot be saved to the FIS unless at least one membership function has been created for it. Variables can be of either "Input" or "Output" type, and accessing inputs or outputs is as easy as selecting the correct tabbed pane (located below the FIS parameters panel). To create a new variable, simply select either "Inputs" or "Outputs" from the tabbed pane, and click the "New" button; this will bring up the following menu.



From here, you may specify the name of the variable, and the range it's values fall between. Any variable saved without a name will be automatically renamed to "unnamed", and bear in mind that the minimum of the range cannot be higher than the maximum. You will be able to see the "Membership Functions" panel, which contains a list of all the membership functions that have been added to the specific variable. Of course, at this point we will have none, as we have just created a new variable. Remember that variables cannot be saved without at least one membership function being added. Clicking on the "New MF" button will bring up the menu that is pictured below, this is membership function creation menu.

From here you can specify the name, type, and parameters of the membership function you are creating. To change the type, simply click the drop down box, and select the function you wish to create, the parameters necessary will be automatically generated for you. The currently supported membership functions are: Gaussian, double Gaussian (referred to as Gaussian B), trapezoidal and triangular. Any membership functions that are saved without a name are given the name “unnamed”. It is also worth noting that each of the input boxes requires a numerical value and as such, any empty boxes, or boxes with non-numerical data in them, will result in an error and the membership function not being saved. Click “OK” to add your membership function to your variable.

The screenshot shows the 'Membership Function Creator' dialog box. It has a title bar with a close button. Inside, there is a text field for 'Identification Name'. Below it is a 'Function Type' dropdown menu currently set to 'Gaussian B Curve'. Under the 'Parameters' section, there are five input fields: 'Left Sigma', 'Left Mean', 'Right Sigma', 'Right Mean', and 'Height'. An 'OK' button is located at the bottom right.

Removal or editing of variables and membership functions

Once a membership function or variable has been created, it will be displayed along with some brief information, followed by both an “Edit” and a “Delete” button. This is demonstrated below, on the variable editor menu, after adding our first membership function.

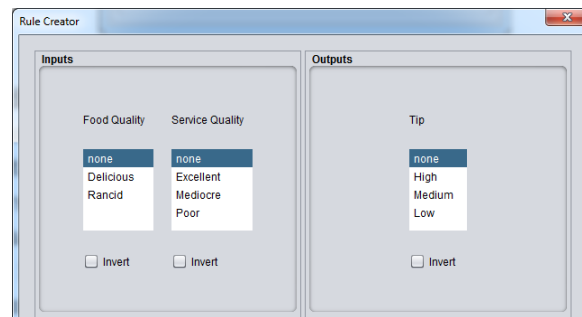
All membership functions and variables can be edited or deleted using these buttons. Clicking the “Edit” button will launch the respective creation panel (variable editor for variables, and membership function editor for membership functions) and the values of the variable or membership function will be loaded, so that they can be changed. Edits are allowed to any entered values, so long as they maintain the correct format (for example, you cannot change any of the numerical values to be strings). A confirmation dialogue will be launched before the deletion of a variable or membership function, in case this button is accidentally clicked, and removal is not what the user desires. You can see from the screen shot below, that once a membership function has been added to a variable, it can be saved, which allows for editing or removal.

The screenshot shows the 'Variable Editor' dialog box. It has a title bar with a close button. Inside, there is a 'Name' text field containing 'TestVariable'. Below it is a 'Range' section with two input fields, '0' and '10'. Under the 'Membership Functions' section, there is a list showing 'Gaussian: TestMF' with 'Edit' and 'Delete' buttons next to it. A 'New MF' button is also present. An 'OK' button is at the bottom right.

This screenshot shows the 'Inputs' tab of the Variable Editor. It has three tabs: 'Inputs', 'Outputs', and 'Rules'. Under the 'Input Variables' section, there is a table-like structure with 'TestVariable' and 'Mfs: 1'. To the right of this are 'Edit' and 'Delete' buttons. A 'New' button is located at the bottom.

Creating rules and adding them to a FIS

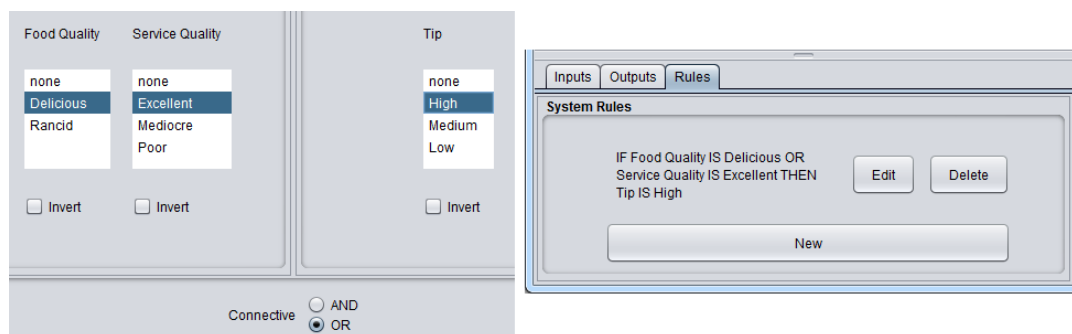
Once an input variable and an output variable have been created (each with multiple membership functions), we can start specifying rules that will be applied when the FIS is being evaluated in the *R* manipulation system. In the example below, I have created three membership functions, “Food Quality” (which has membership functions “Delicious” and “Rancid”), “Service Quality” (which has membership functions “Excellent”, “Mediocre” and “Poor”), and one output variable, “Tip” (which has membership functions “High”, “Medium” and “Low”). You can see that the Rule Editor (accessed from clicking the “Rules” tab on the FIS Editor window, and clicking “New” to create a new rule), shown below, displays all the variables in the system, along with each of their membership functions in an easy to see fashion.



Creation of rules from this menu is now a simple task of selecting the desired membership function from each of the variables, selecting an appropriate connective, and deciding upon the weight of the particular rule. For example, if I wished to create the following rule

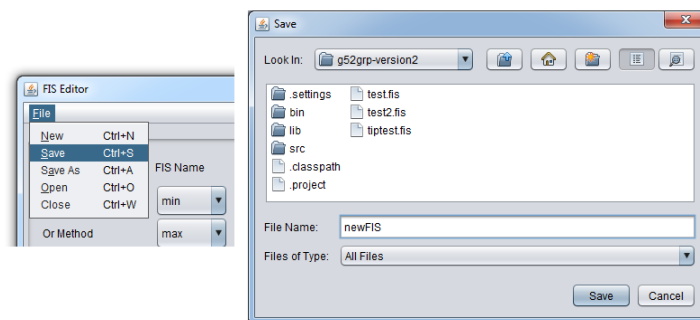
$$\text{If } (\text{Food Quality} = \text{Delicious} \vee \text{Service Quality} = \text{Excellent}) \rightarrow \text{Tip} = \text{High}$$

Meaning if the food quality is delicious or the service quality is excellent, the tip left will be high. I simply have to select “Delicious” from the “Food Quality” variable, “Excellent” from the “Service Quality” variable, and “High” from the “Tip” variable (along with selecting the “Or” connective from the bottom panel), followed by clicking “OK”, and my rule will be created. This can be shown below, along side how this well then be represented on the “Rules” panel of the FIS editor.



Exporting a FIS to the *R* manipulation system

The final task that can be completed using the FIS Construction system is the exporting of FIS objects, to be loaded in by the *R* manipulation system. This is done by, after creating a FIS, pressing “File” followed by “Save” or “Save As” on the FIS editor menu (alternatively, *CTRL* and *S* for “Save”, or *CTRL* and *A* for “Save As”). This will then bring up a navigation box, which will ask you where you wish to save the file to, and to specify a name for a file. This file will be saved to the given location, and can then be loaded into the *R* manipulation code using the *readFIS* function (see the *R* manipulation system user manual for further instructions on this).



Graphs

When plotting membership functions onto variable, you will be able to see a graphical representation of your data. This functionality is provided by the JFreeChart package¹² and comes with more functionality for manipulation of the graph. This includes the changing of formatting and colouring of the graph, saving the graph as an image or printing it, or setting the zoom level, to browse specific data. A picture of the variable editor with three membership functions plotted on a graph is shown below. The content menu shown is accessed by right-clicking anywhere on the graph.



¹²<http://www.jfree.org/jfreechart/>

D Process of CRAN release

In order to prepare a package for a CRAN submission, we needed to adhere to the following steps as performed in a Windows OS Environment. The package we will create will be purely based upon our own developed *R* code, and will not include the Java code for the initial release.

1. Start the *R* interpreter (64-bit version 2.15.3); at this stage we must ensure that absolutely no objects exist in the workspace (they can be passed into memory from a pre-loaded workspace) as these could be accidentally included when the package is eventually generated. To remove any objects from the workspace we use the command:

```
> rm(list=ls(all=TRUE))
```

We now have a blank workspace environment and are ready to proceed.

2. Set up the working directory; this is the directory where the package skeleton will be generated. It matters not if this is the same directory as the sourced *R* code. To set up a working directory, we first created a new directory called `R_Package` at our SVN directory:

```
H:\gp12-jmg\R_Package\
```

This would serve as the location in which anything package related would be placed. To set this as the working directory for the *R* Interpreter, we used the command:

```
> setwd("H:\\gp12-jmg\\R_Package")
```

Please note that escaped backslash characters are required when entering the command in the *R* interpreter.

3. Source the *R* code by using the following command:

```
> source("H:\\gp12-jmg\\R\\FuzzyToolkit.R")
```

The argument is the *R* source file, and an absolute path to it must be provided (again with escaped backslash characters).

4. Create the package skeleton; this step is where we use an in-built *R* command to automatically generate the directories, sub-directories and files that make up an *R* package (one is able to manually create a package, but for our project we decided to use the automatic generation method). We used the command `package.skeleton()` where the `name` argument results in what the name of the package will be, in our case `FuzzyToolkit`. The `code_files` argument is simply the source of the *R* source code file, again with an absolute file path. The console will output progress reports upon execution as visible below:

```
>package.skeleton(name="FuzzyToolkit", code_files="H:\\gp12-jmg\\R\\FuzzyToolkit.R")
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Copying code files ...
Making help files ...
Done.
Further steps are described in "./FuzzyToolkit/Read-and-delete-me".
```

At this point, a package had now been generated into our working directory with the appropriate sub-directories and files.

5. Editing the generated files to meet the CRAN requirements. Our package at this point will have the following structure:

```
FuzzyToolkit/
  man/
  R/
  DESCRIPTION
  NAMESPACE
```

The sub-directory `man/` contains the automatically generated manual pages for every object from the sourced *R* file. This means that every function and variable will have its own manual page. One is able to delete pages and/or combine them; however for our package, each individual function requires its own manual page. Each file will need to be edited as at first they will only contain blank templates (set similar to L^AT_EXstyle, but much more limited).

Each manual page for a function contains (and of which needs editing):

- (a) A function name, alias and title.
- (b) A function description, which should concisely state what the function actually does.
- (c) A function's usage, as in, its arguments.
- (d) How each parameter is used.
- (e) Further details, if necessary.
- (f) The returned value(s) from the function.
- (g) Any external references to sources, such as websites.
- (h) Author of the manual page.
- (i) Further notes, if necessary.
- (j) See also section, if necessary (refers other objects that might be related).
- (k) Examples of the function in use.
- (l) Standard keywords.

Each manual page for a data set, such as a variable contains (and of which needs editing):

- (a) A variable name, alias and title
- (b) A concise description of the dataset.
- (c) Its usage.
- (d) Any further details, if necessary.
- (e) A source for the dataset, if necessary (in our case this is not so).
- (f) Further references.
- (g) Example usage.
- (h) Standard keywords.

A single manual page for the entire package contains (of which needs editing):

- (a) A name, alias, document type and title.
- (b) A concise description of what the package does overall.
- (c) Any further details. Generally this section will automatically contain the Package, Type, Version, Date of creation/release, and license.
- (d) The author(s) of the package.
- (e) The maintainer of the package.
- (f) Any references, if necessary to external sources.
- (g) Standard keywords, if necessary.
- (h) Example of the package in use, typically consisting only of the most important functions.

The sub-directory `R/` contains the actual source code file(s), but does not need to be edited.

The DESCRIPTION file contains basic information about the package, with required fields for:

- (a) The package name.
- (b) The package version.
- (c) The date of package release in YYYY-MM-DD format.
- (d) Authors of the code included in the package.
- (e) The maintainer of the package.
- (f) The license type of the package.
- (g) Any dependencies the package may have. There should always be at least one regarding what version of R it relies on.
- (h) A concise description of the package, what it does and why it may be useful.

There are many optional fields that can also be included, further information can be found at:

[http://cran.r-project.org/doc/manuals/R-exts.html# The-DESCRIPTION-file](http://cran.r-project.org/doc/manuals/R-exts.html#The-DESCRIPTION-file).

6. Building the package; at this point, our package (with all its files and sub-directories) was finished and almost ready for an official release. However, submitting to CRAN requires the package in the .tar.gz format, which was accomplished by using the following command:

```
> system("R CMD build FuzzyToolkit")
```

A tarball (.tar.gz) archive is generated in the working directory which contains everything necessary to be considered a CRAN-ready *R* package.

7. Checking the package; the final step before submission, is to ensure that the package can actually be installed onto a given machine (or at least, an environment specifically supported by the *R* package). In order to check the archived package, we used the command:

```
> system("R CMD check FuzzyToolkit")
```

It essentially works like a debugger and ensures the required quality of all package elements. A new directory will be generated into upon execution of the command, which can be ignored/deleted assuming no errors were present, which was indeed the case for our package creation process. However, errors may occur, and one should check the error log as generated within this new directory and fix the stated issues appropriately.

After following the preceding instructions, our package archive would now meet the CRAN Repository Policy (<http://cran.r-project.org/web/packages/policies.html>). Once the package had successfully passed the previous “check” command, it had essentially been given the green-light to be submitted online, the process of which is detailed below:

1. Compile an email that will accompany the package. It should include the claim to the development of the package in question (in our case, the FuzzyToolkit), and ensure it states that what is being submitted is in fact a *R* package, and nothing else (primarily due to security concerns on behalf of CRAN).

2. Connect to the following FTP-based URL:

```
ftp://CRAN.R-project.org/incoming/
```

In order to do so, an FTP client is required. Simply submit the tarball archive using the client (in our case, FileZilla was used).

3. Once the archive has been successfully uploaded to the aforementioned address, the email must also be sent immediately after to:

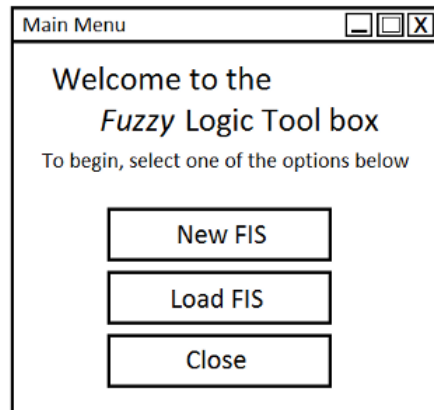
```
CRAN@R-project.org
```

This concludes the procedures and requirements to meet CRAN’s submission policies for an *R*-package. At the time of submitted this coursework (19/03/13) we have not recieved confirmation from CRAN as to whether or not our package has been accepted.

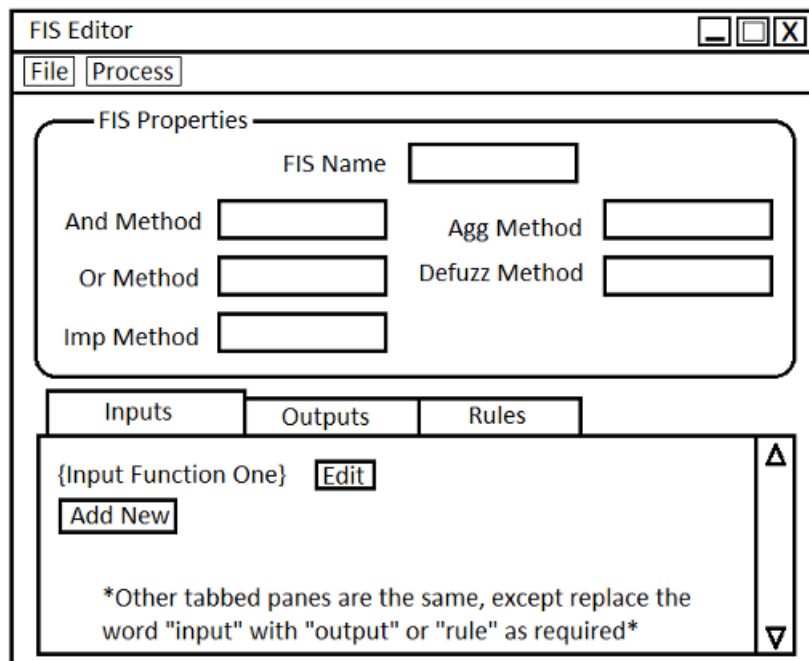
E Initial Designs of the System

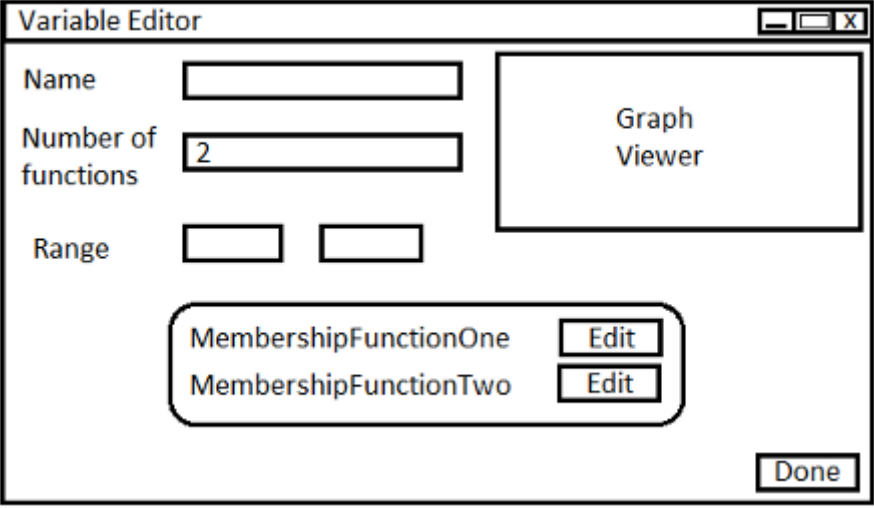
As part of the specification of the interim group report, we were required to produce initial designs of the system. We have included this as an appendix here, to show the progression of our designs, from initial concepts, to the final designs as display in section D.

Main Menu

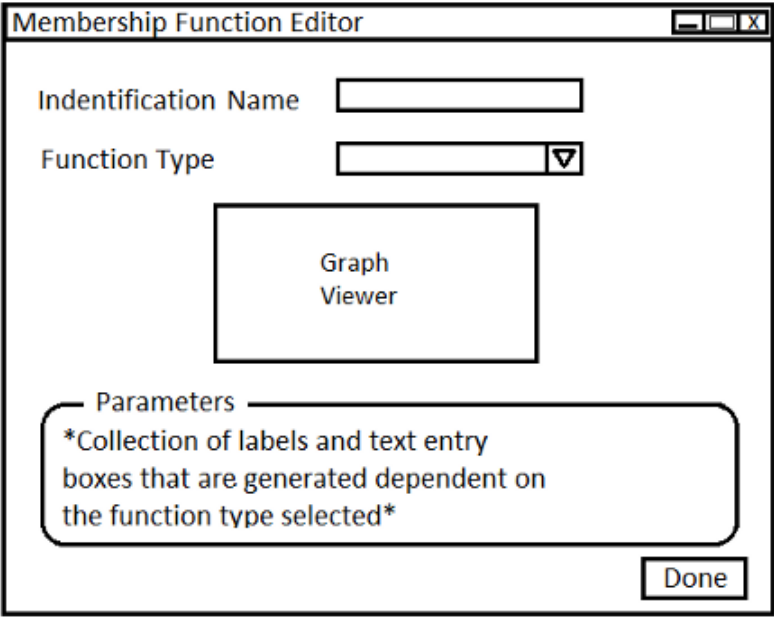


FIS Editor



Variable Editor

The Variable Editor window has a title bar with standard window controls. It contains several input fields: a text box for 'Name', a text box for 'Number of functions' containing the value '2', and two empty text boxes for 'Range'. To the right of these fields is a large rectangular area labeled 'Graph Viewer'. Below the input fields is a rounded rectangular container with two entries: 'MembershipFunctionOne' and 'MembershipFunctionTwo', each followed by an 'Edit' button. A 'Done' button is located in the bottom right corner.

Membership Function Editor

The Membership Function Editor window has a title bar with standard window controls. It contains two input fields: 'Identification Name' (a text box) and 'Function Type' (a dropdown menu with a downward arrow). Below these is a rectangular area labeled 'Graph Viewer'. At the bottom is a rounded rectangular container labeled 'Parameters' with the text '*Collection of labels and text entry boxes that are generated dependent on the function type selected*'. A 'Done' button is in the bottom right corner.

Rule Editor

The Rule Editor window is titled "Rule Editor" and contains the following elements:

- Inputs:** A section containing two input boxes. The first box is labeled "(Input 1)" and contains "Value a", "Value b", and "Value c". The second box is labeled "(Input n)" and also contains "Value a", "Value b", and "Value c". Each box has a vertical scrollbar on the right and an "Invert" checkbox below it. The "Invert" checkbox for "(Input n)" is checked.
- Logical connective:** A section between the input boxes with two options: "AND" (selected with a radio button) and "OR" (unselected with a radio button).
- Outputs:** A section on the right containing an output box labeled "(Output n)" with "Value a", "Value b", and "Value c". It has a vertical scrollbar and an "Invert" checkbox, which is currently unchecked.
- Weight:** A text field labeled "Weight" with the value "1" entered.
- Add Rule:** A button located at the bottom right of the window.

F Minutes from meetings held during the life time of the project

Due to the large quantity of meetings held during the project, including them all in this report would extend its (already large) length by far too much. For this reason, we have included two examples of the minutes taken, and uploaded the others onto our Indefero web-page. Please refer to them there (please note you will need to log in with your CS username and password to view these).

<https://code.cs.nott.ac.uk/p/gp12-jmg/doc/>

The two examples can be found overleaf.

Meeting Date	09/10/12
Meeting Time	10:00 - 10:30
Venue	JC-COMPSCI-B31
Apologies	None

Minutes

1. Allocation of group roles was decided as follows

Craig
Project Manager, R Coder, Head GUI Coder

Nathan
Head R Coder, GUI Coder

Luke
R Coder, GUI Coder

George
Secretary, GUI Coder

Ben
Package Manager

2. Discussion of the three priority objects of the project, these were

- (a) Non trivial re-engineering of the fuzzy tool kit, possible inclusion of *evalFIS* function in this
- (b) Release of an *R* package onto CRAN
- (c) Implementation of a GUI to allow for easier usage of the fuzzy tool kit

3. Discussion of necessary research took place, this included the process of releasing a package onto CRAN, and the *R* language in general

4. Finally, our supervisor informed us that the GUI did not need to be a full implementation of the system, it was more of a sub-system

Action Points

R-Team
Begin research and learning *R* via the following hyperlink
[http : //www.cyclismo.org/tutorial/R/](http://www.cyclismo.org/tutorial/R/)

George
Write up minutes of meeting, begin initial requirements capture

Ben
Research packaging procedure for CRAN

All members
Experiment with SVN, and the features of Indefero such as issues (of which everyone is to create one and solve one).

Meeting Date	16/10/12
Meeting Time	13:00 - 13:30
Venue	JC-COMPSCI-B31
Apologies	None

Minutes

1. General discussion of SVN which included
 - (a) How we had familiarised ourselves with how to check in, check out, add files and update the SVN directory.
 - (b) The general structure of the directories to which we would abide by.
2. General discussion of *R*/Java development regarding the fuzzy tool kit: the *R* development was yet to properly begin, and as such we decided that any development of the Java GUI would be postponed until the *R*-team further understood the tool kit's purpose.
3. Group member progress reports
 - (a) Luke and George had progressed with the mark down syntax for Indefero such that all documentation (typically Minutes and Agendas) were more easily accessible and modifiable online.
 - (b) Craig, Luke and Nathan had started developing the *R*-code for Type 2 fuzzy logic.
 - (c) Ben had started looking into package management to gain an understanding of what was required to create a package compatible with CRAN.
 - (d) All group members had submitted and fixed an issue via the Indefero feature, as well as gained an understanding of SVN.

Action Points

Craig, Luke, Nathan

Familiarise with MATLAB through an online tutorial and worksheet, the reason being that MATLAB's similarities to our project could help in our own code development and testing.

George, Luke

Continue to develop the mark down syntax and have it complete by next session. George will also write/start writing the project specification.

Ben

Continue to research packaging procedure for CRAN