Fuzzy Logic Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

User's Guide

Version 2

How to Contact The MathWorks:

508-647-7000	Phone

508-647-7001	Fax

The MathWorks, Inc. 24 Prime Park Way Natick, MA 01760-1500	Mail

`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

Fuzzy Logic Toolbox User's Guide

Printing History:

January 1995	First printing
April 1997	Second printing
January 1998	Third printingRevised for MATLAB 5.2
January 1999	Minor revisions for Release 11 (Online only)

Forward

The past few years have witnessed a rapid growth in the number and variety of applications of fuzzy logic. The applications range from consumer products such as cameras, camcorders, washing machines, and microwave ovens to industrial process control, medical instrumentation, decision-support systems, and portfolio selection.

To understand the reasons for the growing use of fuzzy logic it is necessary, first, to clarify what is meant by fuzzy logic.

Fuzzy logic has two different meanings. In a narrow sense, fuzzy logic is a logical system, which is an extension of multivalued logic. But in a wider sense—which is in predominant use today—fuzzy logic (FL) is almost synonymous with the theory of fuzzy sets, a theory which relates to classes of objects with unsharp boundaries in which membership is a matter of degree. In this perspective, fuzzy logic in its narrow sense is a branch of FL. What is important to recognize is that, even in its narrow sense, the agenda of fuzzy logic is very different both in spirit and substance from the agendas of traditional multivalued logical systems.

In the Fuzzy Logic Toolbox, fuzzy logic should be interpreted as FL, that is, fuzzy logic in its wide sense. The basic ideas underlying FL are explained very clearly and insightfully in the Introduction. What might be added is that the basic concept underlying FL is that of a linguistic variable, that is, a variable whose values are words rather than numbers. In effect, much of FL may be viewed as a methodology for computing with words rather than numbers. Although words are inherently less precise than numbers, their use is closer to human intuition. Furthermore, computing with words exploits the tolerance for imprecision and thereby lowers the cost of solution.

Another basic concept in FL, which plays a central role in most of its applications, is that of a fuzzy if-then rule or, simply, fuzzy rule. Although rule-based systems have a long history of use in AI, what is missing in such systems is a machinery for dealing with fuzzy consequents and/or fuzzy antecedents. In fuzzy logic, this machinery is provided by what is called the calculus of fuzzy rules. The calculus of fuzzy rules serves as a basis for what might be called the Fuzzy Dependency and Command Language (FDCL). Although FDCL is not used explicitly in Fuzzy Logic Toolbox, it is effectively one of its principal constituents. In this connection, what is important to

recognize is that in most of the applications of fuzzy logic, a fuzzy logic solution is in reality a translation of a human solution into FDCL.

What makes the Fuzzy Logic Toolbox so powerful is the fact that most of human reasoning and concept formation is linked to the use of fuzzy rules. By providing a systematic framework for computing with fuzzy rules, the Fuzzy Logic Toolbox greatly amplifies the power of human reasoning. Further amplification results from the use of MATLAB and graphical user interfaces – areas in which The MathWorks has unparalleled expertise.

A trend which is growing in visibility relates to the use of fuzzy logic in combination with neurocomputing and genetic algorithms. More generally, fuzzy logic, neurocomputing, and genetic algorithms may be viewed as the principal constituents of what might be called soft computing. Unlike the traditional, hard computing, soft computing is aimed at an accommodation with the pervasive imprecision of the real world. The guiding principle of soft computing is: Exploit the tolerance for imprecision, uncertainty, and partial truth to achieve tractability, robustness, and low solution cost. In coming years, soft computing is likely to play an increasingly important role in the conception and design of systems whose MIQ (Machine IQ) is much higher than that of systems designed by conventional methods.

Among various combinations of methodologies in soft computing, the one which has highest visibility at this juncture is that of fuzzy logic and neurocomputing, leading to so-called neuro-fuzzy systems. Within fuzzy logic, such systems play a particularly important role in the induction of rules from observations. An effective method developed by Dr. Roger Jang for this purpose is called ANFIS (Adaptive Neuro-Fuzzy Inference System). This method is an important component of the Fuzzy Logic Toolbox.

The Fuzzy Logic Toolbox is highly impressive in all respects. It makes fuzzy logic an effective tool for the conception and design of intelligent systems. The Fuzzy Logic Toolbox is easy to master and convenient to use. And last, but not least important, it provides a reader-friendly and up-to-date introduction to the methodology of fuzzy logic and its wide-ranging applications.

Lotfi A. Zadeh
Berkeley, CA
January 10, 1995

Contents

Before You Begin

1 Introduction

2 Tutorial

3 Reference

Before You Begin

This section describes how to use the Fuzzy Logic Toolbox. It explains how to use this guide and points you to additional books for toolbox installation information.

What Is the Fuzzy Logic Toolbox?

The Fuzzy Logic Toolbox is a collection of functions built on the MATLAB® numeric computing environment. It provides tools for you to create and edit fuzzy inference systems within the framework of MATLAB, or if you prefer you can integrate your fuzzy systems into simulations with Simulink®, or you can even build stand-alone C programs that call on fuzzy systems you build with MATLAB. This toolbox relies heavily on graphical user interface (GUI) tools to help you accomplish your work, although you can work entirely from the command line if you prefer.

The toolbox provides three categories of tools:

- Command line functions
- Graphical, interactive tools
- Simulink blocks and examples

The first category of tools is made up of functions that you can call from the command line or from your own applications. Many of these functions are MATLAB M-files, series of MATLAB statements that implement specialized fuzzy logic algorithms. You can view the MATLAB code for these functions using the statement

```
type function_name
```

You can change the way any toolbox function works by copying and renaming the M-file, then modifying your copy. You can also extend the toolbox by adding your own M-files.

Secondly, the toolbox provides a number of interactive tools that let you access many of the functions through a GUI. Together, the GUI- based tools provide an environment for fuzzy inference system design, analysis, and implementation.

The third category of tools is a set of blocks for use with the Simulink simulation software. These are specifically designed for high speed fuzzy logic inference in the Simulink environment.

How to Use This Guide

If you are new to fuzzy logic, begin with Chapter 1, "Introduction." This chapter introduces the motivation behind fuzzy logic and leads you smoothly into the tutorial.

If you are an experienced fuzzy logic user, you may want to start at the beginning of Chapter 2, "Tutorial," to make sure you are comfortable with the fuzzy logic terminology in the Fuzzy Logic Toolbox. If you just want an overview of each graphical tool and examples of specific fuzzy system tasks, turn directly to the section in Chapter 2 entitled, "Building Systems with the Fuzzy Logic Toolbox." This section does not include information on the adaptive data modeling application covered by the toolbox function `anfis`. The basic functionality of this tool can be found in the section in Chapter 2 entitled, "anfis and the ANFIS Editor GUI."

If you just want to start as soon as possible and experiment, you can open an example system right away by typing

```
fuzzy tipper
```

This brings up the Fuzzy Inference System (FIS) editor for an example decision making problem that has to do with how to tip in a restaurant.

All toolbox users should use Chapter 3, "Reference," for information on specific tools or functions. Reference descriptions include a synopsis of the function's syntax, as well as a complete explanation of options and operation. Many reference descriptions also include helpful examples, a description of the function's algorithm, and references to additional reading material. For GUI-based tools, the descriptions include options for invoking the tool.

Installation

To install this toolbox on a workstation or a large machine, see the *Installation Guide for UNIX*. To install the toolbox on a PC or Macintosh, see the *Installation Guide for PC and Macintosh*.

To determine if the Fuzzy Logic Toolbox is already installed on your system, check for a subdirectory named `fuzzy` within the main toolbox directory or folder.

Typographical Conventions

To Indicate	This Guide Uses	Example
Example code	`Monospace type` (Use **Code** tag.)	To assign the value 5 to A, enter `A = 5`
Function names	`Monospace type` (Use **Code** tag.)	The `cos` function finds the cosine of each array element.
Function syntax	`Monospace type` for text that must appear as shown. (Use **Code** tag.) *`Monospace italics`* for components you can replace with any variable. (Use **Code-ital** tag.)	The `magic` function uses the syntax *`M`* = `magic(`*`n`*`)`
Keys	**Boldface** with an initial capital letter (Use **Menu-Bodytext** tag.)	Press the **Return** key.
Mathematical expressions	Variables in *italics*. Functions, operators, and constants in standard type. (Use **EquationVariables** tag.)	This vector represents the polynomial $p = x^2 + 2x + 3$

MATLAB output	`Monospace type` (Use **Code** tag.)	MATLAB responds with
		`A =`
		`5`

To Indicate	This Guide Uses	Example
Menu names, menu items, and controls	**Boldface** with an initial capital letter (Use **Menu-Bodytext** tag.)	Choose the **File** menu.
New terms	NCS *italics* *(*Use **Body text-ital** tag.*)*	An *array* is an ordered collection of information.

1

Introduction

What Is Fuzzy Logic?

Fuzzy logic is all about the relative importance of precision: How important is it to be exactly right when a rough answer will do? All books on fuzzy logic begin with a few good quotes on this very topic, and this is no exception. Here is what some clever people have said in the past:

Precision is not truth.

—Henri Matisse

Sometimes the more measurable drives out the most important.

—René Dubos

Vagueness is no more to be done away with in the world of logic than friction in mechanics.

—Charles Sanders Peirce

I believe that nothing is unconditionally true, and hence I am opposed to every statement of positive truth and every man who makes it.

—H. L. Mencken

So far as the laws of mathematics refer to reality, they are not certain. And so far as they are certain, they do not refer to reality.

—Albert Einstein

As complexity rises, precise statements lose meaning and meaningful statements lose precision.

—Lotfi Zadeh

Some pearls of folk wisdom also echo these thoughts:

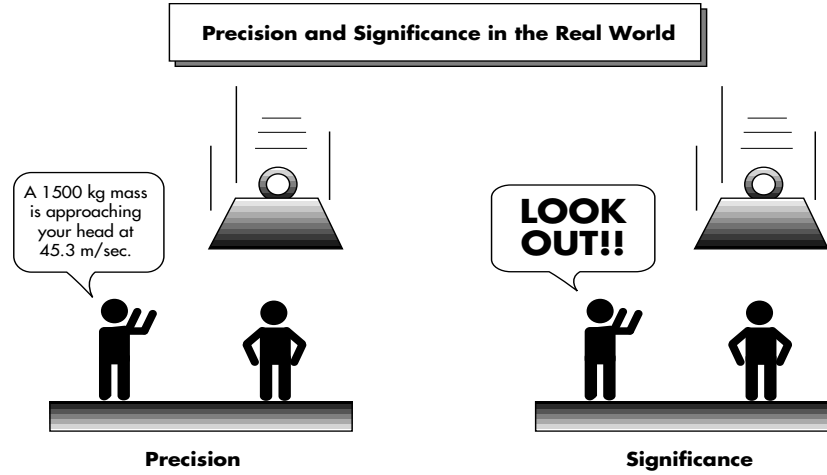
Don't lose sight of the forest for the trees.

Don't be penny wise and pound foolish.

The Fuzzy Logic Toolbox for use with MATLAB is a tool for solving problems with fuzzy logic. Fuzzy logic is a fascinating area of research because it does a good job of trading off between significance and precision—something that humans have been managing for a very long time.

Fuzzy logic sometimes appears exotic or intimidating to those unfamiliar with it, but once you become acquainted with it, it seems almost surprising that no one attempted it sooner. In this sense fuzzy logic is both old and new because,

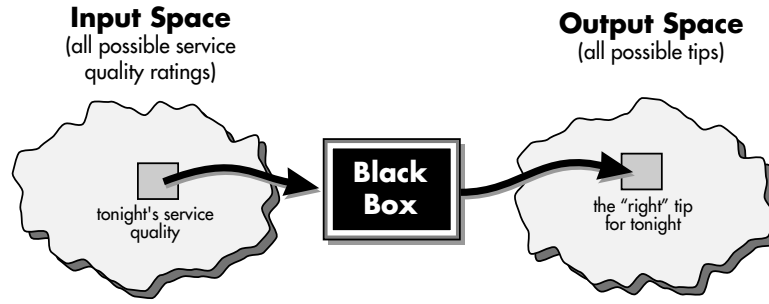
although the modern and methodical science of fuzzy logic is still young, the concepts of fuzzy logic reach right down to our bones.



Fuzzy logic is a convenient way to map an input space to an output space. This is the starting point for everything else, and the great emphasis here is on the word “convenient.”

What do I mean by mapping input space to output space? Here are a few examples: You tell me how good your service was at a restaurant, and I’ll tell you what the tip should be. You tell me how hot you want the water, and I’ll adjust the faucet valve to the right setting. You tell me how far away the subject of your photograph is, and I’ll focus the lens for you. You tell me how fast the car is going and how hard the motor is working, and I’ll shift the gears for you.

A graphical example of an input-output map is shown below.



An input-output map for the tipping problem:
"Given the quality of service, how much should I tip?"

It's all just a matter of mapping inputs to the appropriate outputs. Between the input and the output we'll put a black box that does the work. What could go in the black box? Any number of things: fuzzy systems, linear systems, expert systems, neural networks, differential equations, interpolated multi-dimensional lookup tables, or even a spiritual advisor, just to name a few of the possible options. Clearly the list could go on and on.

Of the dozens of ways to make the black box work, it turns out that fuzzy is often the very best way. Why should that be? As Lotfi Zadeh, who is considered to be the father of fuzzy logic, once remarked: "In almost every case you can build the same product without fuzzy logic, but fuzzy is faster and cheaper."

Why Use Fuzzy Logic?

Here is a list of general observations about fuzzy logic.

- Fuzzy logic is conceptually easy to understand.

  The mathematical concepts behind fuzzy reasoning are very simple. What makes fuzzy nice is the "naturalness" of its approach and not its far-reaching complexity.

- Fuzzy logic is flexible.

  With any given system, it's easy to massage it or layer more functionality on top of it without starting again from scratch.

- Fuzzy logic is tolerant of imprecise data.

  Everything is imprecise if you look closely enough, but more than that, most things are imprecise even on careful inspection. Fuzzy reasoning builds this understanding into the process rather than tacking it onto the end.

- Fuzzy logic can model nonlinear functions of arbitrary complexity.

  You can create a fuzzy system to match any set of input-output data. This process is made particularly easy by adaptive techniques like ANFIS (Adaptive Neuro-Fuzzy Inference Systems), which are available in the Fuzzy Logic Toolbox.

- Fuzzy logic can be built on top of the experience of experts.

  In direct contrast to neural networks, which take training data and generate opaque, impenetrable models, fuzzy logic lets you rely on the experience of people who already understand your system.

- Fuzzy logic can be blended with conventional control techniques.

  Fuzzy systems don't necessarily replace conventional control methods. In many cases fuzzy systems augment them and simplify their implementation.

- Fuzzy logic is based on natural language.

  The basis for fuzzy logic is the basis for human communication. This observation underpins many of the other statements about fuzzy logic.

The last statement is perhaps the most important one and deserves more discussion. Natural language, that which is used by ordinary people on a daily basis, has been shaped by thousands of years of human history to be convenient and efficient. Sentences written in ordinary language represent a triumph of efficient communication. We are generally unaware of this because ordinary language is, of course, something we use every day. Since fuzzy logic is built

atop the structures of qualitative description used in everyday language, fuzzy logic is easy to use.

When Not to Use Fuzzy Logic

Fuzzy logic is not a cure-all. When should you not use fuzzy logic? The safest statement is the first one made in this introduction: fuzzy logic is a convenient way to map an input space to an output space. If you find it's not convenient, try something else. If a simpler solution already exists, use it. Fuzzy logic is the codification of common sense—use common sense when you implement it and you will probably make the right decision. Many controllers, for example, do a fine job without using fuzzy logic. However, if you take the time to become familiar with fuzzy logic, you'll see it can be a very powerful tool for dealing quickly and efficiently with imprecision and nonlinearity.

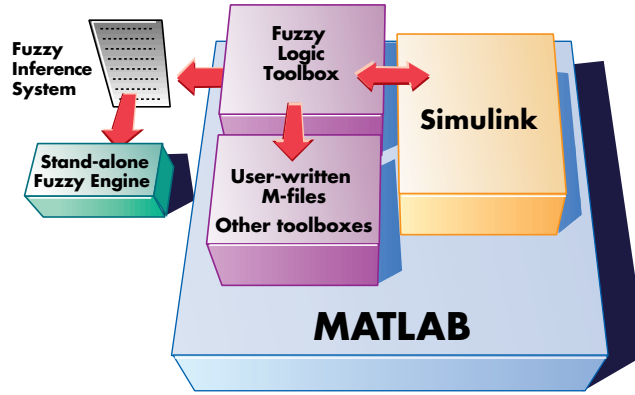
What Can the Fuzzy Logic Toolbox Do?

The Fuzzy Logic Toolbox allows you to do several things, but the most important thing it lets you do is create and edit fuzzy inference systems. You can create these systems using graphical tools or command-line functions, or you can generate them automatically using either clustering or adaptive neuro-fuzzy techniques.

If you have access to Simulink, you can easily test your fuzzy system in a block diagram simulation environment.

The toolbox also lets you run your own stand-alone C programs directly, without the need for Simulink. This is made possible by a stand-alone Fuzzy Inference Engine that reads the fuzzy systems saved from a MATLAB session.

You can customize the stand-alone engine to build fuzzy inference into your own code. All provided code is ANSI compliant.



Because of the integrated nature of MATLAB's environment, you can create your own tools to customize the Fuzzy Logic Toolbox or harness it with another toolbox, such as the Control System, Neural Network, or Optimization Toolbox, to mention only a few of the possibilities.

An Introductory Example: Fuzzy vs. Non-Fuzzy

A specific example would be helpful at this point. To illustrate the value of fuzzy logic, we'll show two different approaches to the same problem: linear and fuzzy. First we will work through this problem the conventional (non-fuzzy) way, writing MATLAB commands that spell out linear and piecewise-linear relations. Then we'll take a quick look at the same system using fuzzy logic.

Consider the tipping problem: what is the "right" amount to tip your waitperson? Here is a clear statement of the problem.

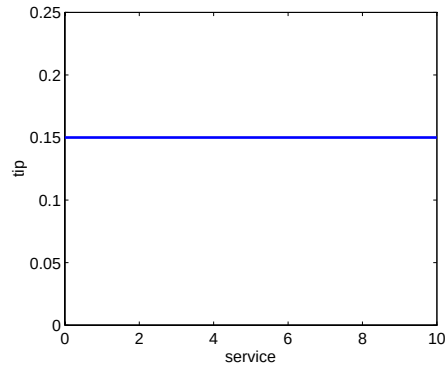
The Basic Tipping Problem. Given a number between 0 and 10 that represents the quality of service at a restaurant (where 10 is excellent), what should the tip be?

Cultural footnote: This problem is based on tipping as it is typically practiced in the United States. An average tip for a meal in the U.S. is 15%, though the actual amount may vary depending on the quality of the service provided.

The Non-Fuzzy Approach

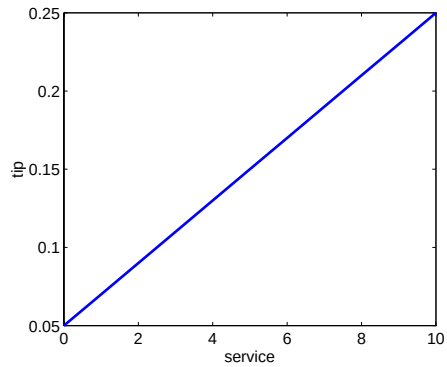
Let's start with the simplest possible relationship. Suppose that the tip always equals 15% of the total bill.

```
tip = 0.15
```



This doesn't really take into account the quality of the service, so we need to add a new term to the equation. Since service is rated on a scale of 0 to 10, we might have the tip go linearly from 5% if the service is bad to 25% if the service is excellent. Now our relation looks like this:

```
tip=0.20/10*service+0.05
```

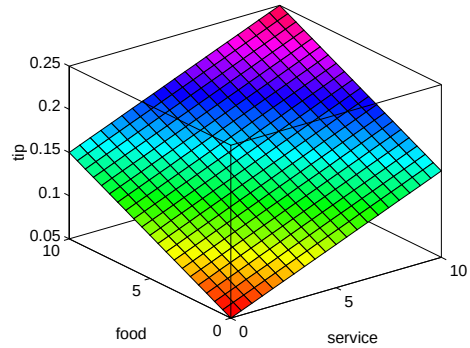


So far so good. The formula does what we want it to do, and it's pretty straightforward. However, we may want the tip to reflect the quality of the food as well. This extension of the problem is defined as follows:

The Extended Tipping Problem. Given two sets of numbers between 0 and 10 (where 10 is excellent) that respectively represent the quality of the service and the quality of the food at a restaurant, what should the tip be?

Let's see how the formula will be affected now that we've added another variable. Suppose we try:

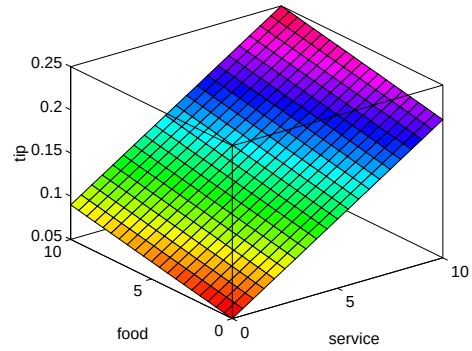
```
tip = 0.20/20*(service+food)+0.05;
```



In this case, the results look pretty, but when you look at them closely, they don't seem quite right. Suppose you want the service to be a more important

factor than the food quality. Let's say that the service will account for 80% of the overall tipping "grade" and the food will make up the other 20%. Try:

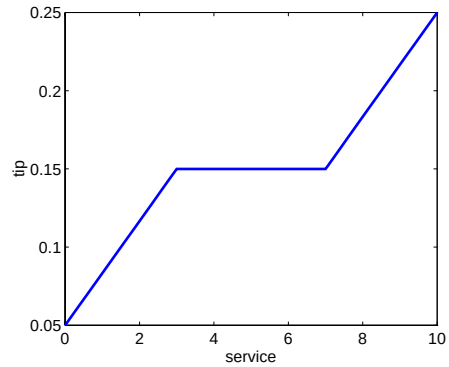
```
servRatio=0.8;
tip=servRatio*(0.20/10*service+0.05) + ...
    (1-servRatio)*(0.20/10*food+0.05);
```



The response is still somehow too uniformly linear. Suppose you want more of a flat response in the middle, i.e., you want to give a 15% tip in general, and will depart from this plateau only if the service is exceptionally good or bad. This, in turn, means that those nice linear mappings no longer apply. We can still salvage things by using a piecewise linear construction. Let's return to the one-dimensional problem of just considering the service. You can string together a simple conditional statement using breakpoints like this:

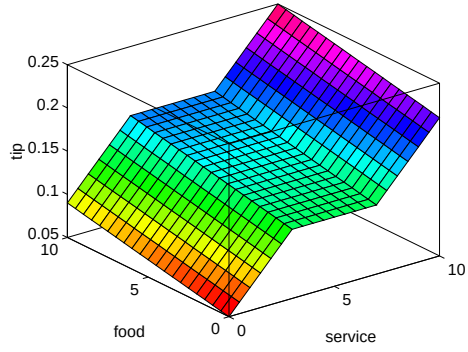
```
if service<3,
    tip=(0.10/3)*service+0.05;
elseif service<7,
    tip=0.15;
elseif service<=10,
    tip=(0.10/3)*(service-7)+0.15;
end
```

The plot looks like this.



If we extend this to two dimensions, where we take food into account again, something like this results:

```
servRatio=0.8;
if service<3,
    tip=((0.10/3)*service+0.05)*servRatio + ...
        (1-servRatio)*(0.20/10*food+0.05);
elseif service<7,
    tip=(0.15)*servRatio + ...
        (1-servRatio)*(0.20/10*food+0.05);
else,
    tip=((0.10/3)*(service-7)+0.15)*servRatio + ...
        (1-servRatio)*(0.20/10*food+0.05);
end
```



Wow! The plot looks good, but the function is surprisingly complicated. It was a little tricky to code this correctly, and it's definitely not easy to modify this code in the future. Moreover, it's even less apparent how the algorithm works to someone who didn't witness the original design process.

The Fuzzy Approach

It would be nice if we could just capture the essentials of this problem, leaving aside all the factors that could be arbitrary. If we make a list of what really matters in this problem, we might end up with the following rule descriptions:

1. *If service is poor, then tip is cheap*
2. *If service is good, then tip is average*
3. *If service is excellent, then tip is generous*

The order in which the rules are presented here is arbitrary. It doesn't matter which rules come first. If we wanted to include the food's effect on the tip, we might add the following two rules:

4. *If food is rancid, then tip is cheap*
5. *If food is delicious, then tip is generous*

In fact, we can combine the two different lists of rules into one tight list of three rules like so:

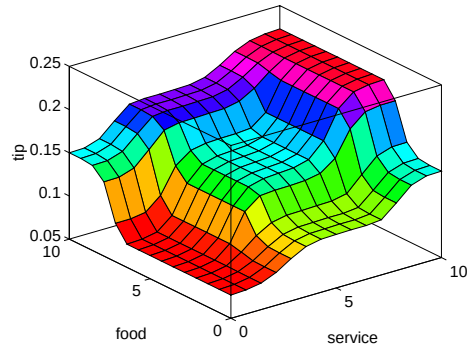
1. *If service is poor or the food is rancid, then tip is cheap*
2. *If service is good, then tip is average*
3. *If service is excellent or food is delicious, then tip is generous*

These three rules are the core of our solution. And coincidentally, we've just defined the rules for a fuzzy logic system. Now if we give mathematical meaning to the linguistic variables (what is an "average" tip, for example?) we would have a complete fuzzy inference system. Of course, there's a lot left to the methodology of fuzzy logic that we're not mentioning right now, things like:

- How are the rules all combined?
- How do I define mathematically what an "average" tip is?

These are questions we provide detailed answers to in the next few chapters. The details of the method don't really change much from problem to problem — the mechanics of fuzzy logic aren't terribly complex. What matters is what

we've shown in this preliminary exposition: fuzzy is adaptable, simple, and easily applied.



Here is the picture associated with the fuzzy system that solves this problem. The picture above was generated by the three rules above. The mechanics of how fuzzy inference works is explained "The Big Picture" on page 2-18, "Foundations of Fuzzy Logic" on page 2-20, and in "Fuzzy Inference Systems" on page 2-36. In the "Building Systems with the Fuzzy Logic Toolbox" on page 2-45, the entire tipping problem is worked through using the graphical tools in the Fuzzy Logic Toolbox.

Some Observations

Here are some observations about the example so far. We found a piecewise linear relation that solved the problem. It worked, but it was something of a nuisance to derive, and once we wrote it down as code, it wasn't very easy to interpret. On the other hand, the fuzzy system is based on some "common sense" statements. Also, we were able to add two more rules to the bottom of the list that influenced the shape of the overall output without needing to undo what had already been done. In other words, the subsequent modification was pretty easy.

Moreover, by using fuzzy logic rules, the maintenance of the structure of the algorithm decouples along fairly clean lines. The notion of an average tip might change from day to day, city to city, country to country, but the underlying logic is the same: if the service is good, the tip should be average. You can recalibrate the method quickly by simply shifting the fuzzy set that defines average without rewriting the fuzzy rules.

You can do this sort of thing with lists of piecewise linear functions, but there is a greater likelihood that recalibration will not be so quick and simple.

For example, here is the piecewise linear tipping problem slightly rewritten to make it more generic. It performs the same function as before, only now the constants can be easily changed.

```
% Establish constants
lowTip=0.05; averTip=0.15; highTip=0.25;
tipRange=highTip-lowTip;
badService=0; okayService=3;
goodService=7; greatService=10;
serviceRange=greatService-badService;
badFood=0; greatFood=10;
foodRange=greatFood-badFood;

% If service is poor or food is rancid, tip is cheap
if service<okayService,
    tip=(((averTip-lowTip)/(okayService-badService)) ...
        *service+lowTip)*servRatio + ...
        (1-servRatio)*(tipRange/foodRange*food+lowTip);
% If service is good, tip is average
elseif service<goodService,
    tip=averTip*servRatio + (1-servRatio)* ...
        (tipRange/foodRange*food+lowTip);
% If service is excellent or food is delicious, tip is generous
else,
    tip=(((highTip-averTip)/ ...
        (greatService-goodService))* ...
        (service-goodService)+averTip)*servRatio + ...
        (1-servRatio)*(tipRange/foodRange*food+lowTip);
end
```

Notice the tendency here, as with all code, for creeping generality to render the algorithm more and more opaque, threatening eventually to obscure it completely. What we're doing here isn't (shouldn't be!) that complicated. True, we can fight this tendency to be obscure by adding still more comments, or perhaps by trying to rewrite it in slightly more self-evident ways, but the medium is not on our side.

The truly fascinating thing to notice is that if we remove everything except for three comments, what remain are exactly the fuzzy rules we wrote down before:

```
% If service is poor or food is rancid, tip is cheap
% If service is good, tip is average
% If service is excellent or food is delicious, tip is generous
```

If, as with a fuzzy system, the comment is identical with the code, think how much more likely your code is to have comments! Fuzzy logic lets the language that's clearest to you, high level comments, also have meaning to the machine, which is why it's a very successful technique for bridging the gap between people and machines.

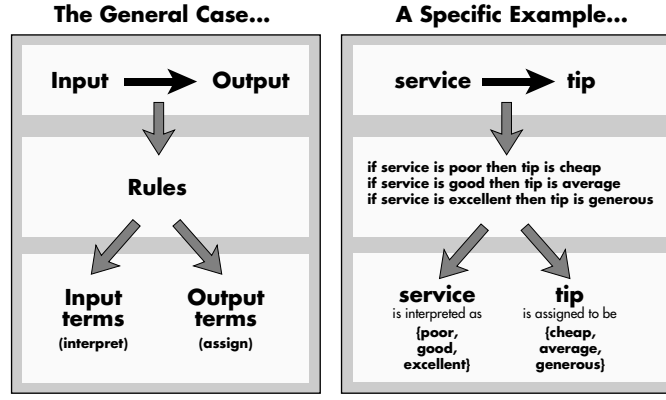
Or think of it this way: by making the equations as simple as possible (linear) we make things simpler for the machine but more complicated for us. But really the limitation is no longer the computer—it's our mental model of what the computer is doing. We all know that computers have the ability to make things hopelessly complex; fuzzy logic is really about reclaiming the middle ground and letting the machine work with our preferences rather than the other way around. It's about time.

2

Tutorial

The Big Picture

We'll start with a little motivation for where we are headed in this chapter. The point of fuzzy logic is to map an input space to an output space, and the primary mechanism for doing this is a list of if-then statements called rules. All rules are evaluated in parallel, and the order of the rules is unimportant. The rules themselves are useful because they refer to variables and the adjectives that describe those variables. Before we can build a system that interprets rules, we have to define all the terms we plan on using and the adjectives that describe them. If we want to talk about how hot the water is, we need to define the range that the water's temperature can be expected to vary over as well as what we mean by the word hot. These are all things we'll be discussing in the next several sections of the manual. The diagram below is something like a roadmap for the fuzzy inference process. It shows the general description of a fuzzy system on the left and a specific fuzzy system (the tipping example from the Introduction) on the right.



To summarize the concept of fuzzy inference depicted in this figure, *fuzzy inference is a method that interprets the values in the input vector and, based on some set of rules, assigns values to the output vector.*

This chapter is designed to guide you through the fuzzy logic process step by step by providing an introduction to the theory and practice of fuzzy logic. The first three sections of this chapter are the most important—they move from

general to specific, first introducing underlying ideas and then discussing implementation details specific to the toolbox. These three areas are

- **Foundations of fuzzy logic**, which is an introduction to the general concepts. If you're already familiar with fuzzy logic, you may want to skip this section.
- **Fuzzy inference systems**, which explains the specific methods of fuzzy inference used in the Fuzzy Logic Toolbox. Since the field of fuzzy logic uses many terms that do not yet have standard interpretations, you should consider reading this section just to become familiar with the fuzzy inference process as it is employed here.
- **Building systems with the Fuzzy Logic Toolbox**, which goes into detail about how you build and edit a fuzzy system using this toolbox. This introduces the graphical user interface tools available in the Fuzzy Logic Toolbox and guides you through the construction of a complete fuzzy inference system from start to finish. If you just want to get up to speed as quickly as possible, start here.

After this there are sections that touch on a variety of topics, such as Simulink use, automatic rule generation, and demonstrations. But from the point of view of getting to know the toolbox, these first three sections are the most crucial.

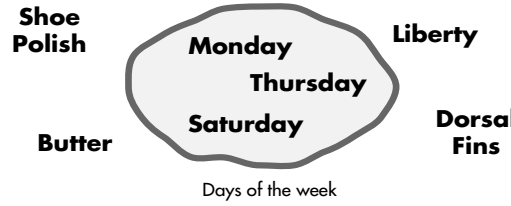
Foundations of Fuzzy Logic

Everything is vague to a degree you do not realize till you have tried to make it precise. —Bertrand Russell

Fuzzy Sets

Fuzzy logic starts with the concept of a fuzzy set. A *fuzzy set* is a set without a crisp, clearly defined boundary. It can contain elements with only a partial degree of membership.

To understand what a fuzzy set is, first consider what is meant by what we might call a *classical set*. A classical set is a container that wholly includes or wholly excludes any given element. For example, the set of days of the week unquestionably includes Monday, Thursday, and Saturday. It just as unquestionably excludes butter, liberty, and dorsal fins, and so on.

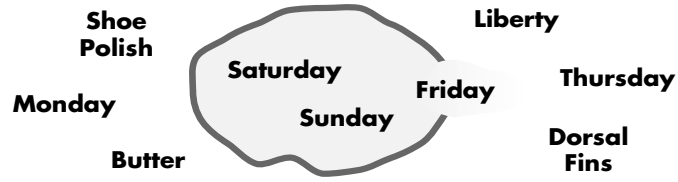


We call this set a classical set because it's been around for such a long time. It was Aristotle who first formulated the Law of the Excluded Middle, which says X must either be in set A or in set not-A. Another version runs like this:

Of any subject, one thing must be either asserted or denied.

Here is a restatement of the law with annotations: "Of any subject (say Monday), one thing (being a day of the week) must be either asserted or denied (I assert that Monday is a day of the week)." This law demands that opposites, the two categories A and not-A, should between them contain the entire universe. Everything falls into either one group or the other. There is no thing that is both a day of the week and not a day of the week.

Now consider the set of days comprising a weekend. The diagram below is one attempt at classifying the weekend days.



Days of the weekend

Most would agree that Saturday and Sunday belong, but what about Friday? It "feels" like a part of the weekend, but somehow it seems like it should be technically excluded. So in the diagram above Friday tries its best to sit on the fence. Classical or "normal" sets wouldn't tolerate this kind of thing. Either you're in or you're out. Human experience suggests something different, though: fence sitting is a part of life.

Of course we're on tricky ground here, because we're starting to take individual perceptions and cultural background into account when we define what constitutes the weekend. But this is exactly the point. Even the dictionary is imprecise, defining the weekend as "the period from Friday night or Saturday to Monday morning." We're entering the realm where sharp edged yes-no logic stops being helpful. Fuzzy reasoning becomes valuable exactly when we're talking about how people really perceive the concept "weekend" as opposed to a simple-minded classification useful for accounting purposes only. More than anything else, the following statement lays the foundations for fuzzy logic:

In fuzzy logic, the truth of any statement becomes a matter of degree.

Any statement can be fuzzy. The tool that fuzzy reasoning gives is the ability to reply to a yes-no question with a not-quite-yes-or-no answer. This is the kind of thing that humans do all the time (think how rarely you get a straight answer to a seemingly simple question) but it's a rather new trick for computers.

How does it work? Reasoning in fuzzy logic is just a matter of generalizing the familiar yes-no (Boolean) logic. If we give "true" the numerical value of 1 and

"false" the numerical value of 0, we're saying that fuzzy logic also permits in-between values like 0.2 and 0.7453. For instance:

Q: Is Saturday a weekend day?

A: 1 (yes, or true)

Q: Is Tuesday a weekend day?

A: 0 (no, or false)

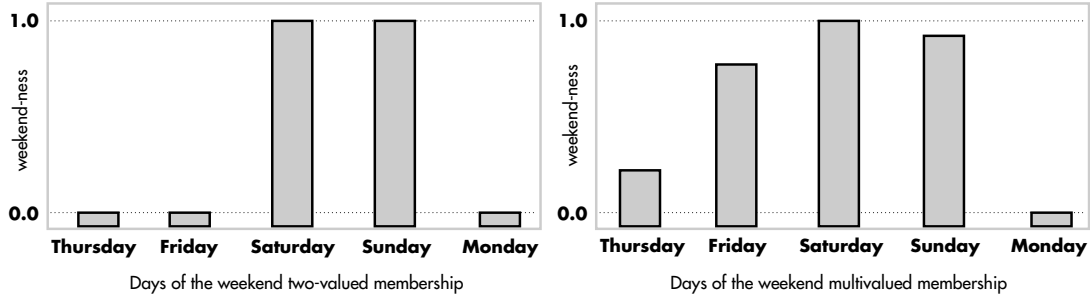
Q: Is Friday a weekend day?

A: 0.8 (for the most part yes, but not completely)

Q: Is Sunday a weekend day?

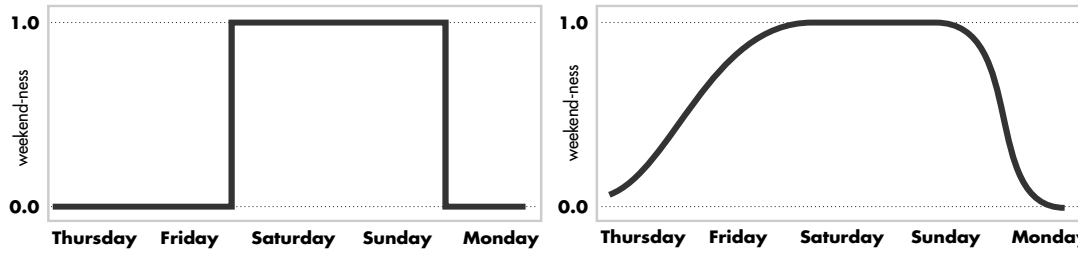
A: 0.95 (yes, but not quite as much as Saturday).

Below on the left is a plot that shows the truth values for "weekend-ness" if we are forced to respond with an absolute yes or no response. On the right is a plot that shows the truth value for weekend-ness if we are allowed to respond with fuzzy in-between values.



Technically, the representation on the right is from the domain of *multivalued logic* (or multivalent logic). If I ask the question "Is X a member of set A?" the answer might be yes, no, or any one of a thousand intermediate values in between. In other words, X might have partial membership in A. Multivalued logic stands in direct contrast to the more familiar concept of two-valued (or bivalent yes-no) logic. Two-valued logic has played a central role in the history of science since Aristotle first codified it, but the time has come for it to share the stage.

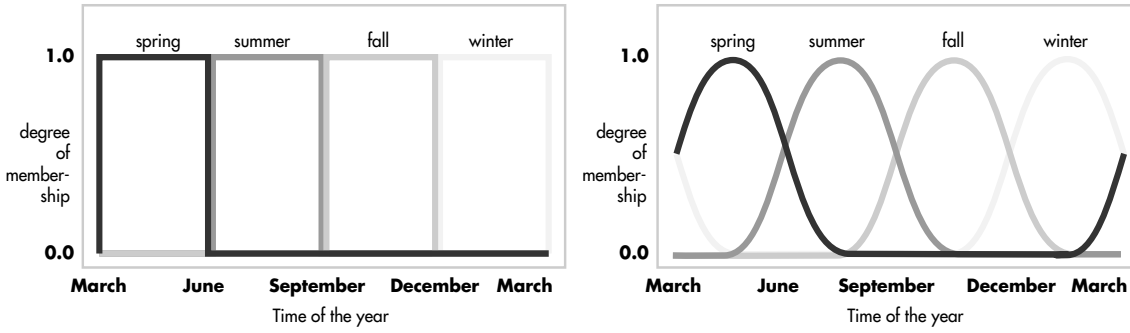
To return to our example, now consider a continuous scale time plot of weekend-ness shown below.



By making the plot continuous, we're defining the degree to which any given instant belongs in the weekend rather than an entire day. In the plot on the left, notice that at midnight on Friday, just as the second hand sweeps past 12, the weekend-ness truth value jumps discontinuously from 0 to 1. This is one way to define the weekend, and while it may be useful to an accountant, it doesn't really connect with our real-world experience of weekend-ness.

The plot on the right shows a smoothly varying curve that accounts for the fact that all of Friday, and, to a small degree, parts of Thursday, partake of the quality of weekend-ness and thus deserve partial membership in the fuzzy set of weekend moments. The curve that defines the weekend-ness of any instant in time is a function that maps the input space (time of the week) to the output space (weekend-ness). Specifically it is known as a *membership function*. We'll discuss this in greater detail in the next section.

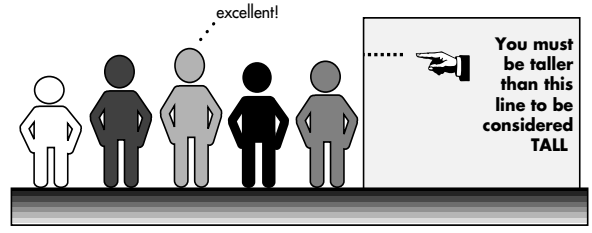
As another example of fuzzy sets, consider the question of seasons. What season is it right now? In the northern hemisphere, summer officially begins at the exact moment in the earth's orbit when the North Pole is pointed most directly toward the sun. It occurs exactly once a year, in late June. Using the astronomical definitions for the season, we get sharp boundaries as shown on the left in the figure on the next page. But what we experience as the seasons varies more or less continuously as shown on the right below (in temperate northern hemisphere climates).



Membership Functions

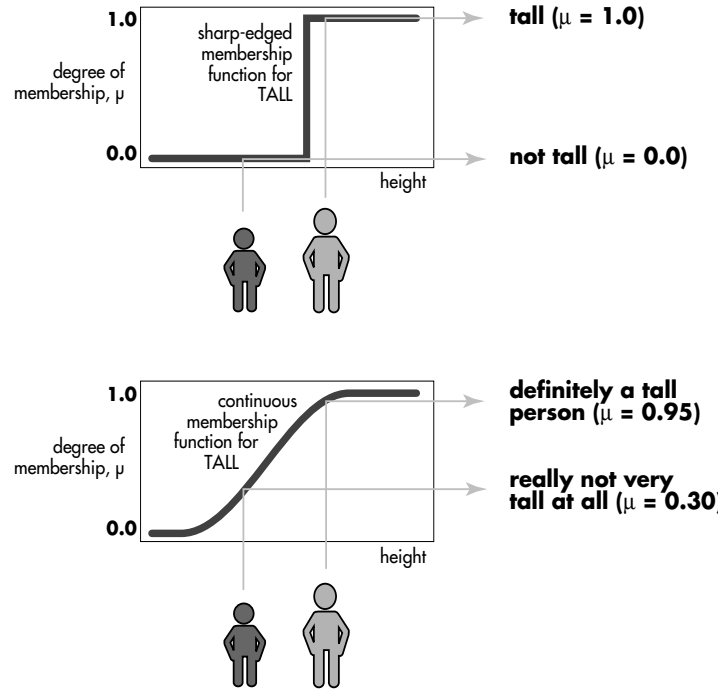
A *membership function* (MF) is a curve that defines how each point in the input space is mapped to a membership value (or degree of membership) between 0 and 1. The input space is sometimes referred to as the *universe of discourse*, a fancy name for a simple concept.

One of the most commonly used examples of a fuzzy set is the set of tall people. In this case the universe of discourse is all potential heights, say from 3 feet to 9 feet, and the word "tall" would correspond to a curve that defines the degree to which any person is tall. If the set of tall people is given the well-defined (crisp) boundary of a classical set, we might say all people taller than six feet are officially considered tall. But such a distinction is clearly absurd. It may make sense to consider the set of all real numbers greater than six because numbers belong on an abstract plane, but when we want to talk about real people, it is unreasonable to call one person short and another one tall when they differ in height by the width of a hair.



But if the kind of distinction shown above is unworkable, then what is the right way to define the set of tall people? Much as with our plot of weekend days, the

figure below shows a smoothly varying curve that passes from not-tall to tall. The output-axis is a number known as the membership value between 0 and 1. The curve is known as a membership function and is often given the designation of μ. This curve defines the transition from not tall to tall. Both people are tall to some degree, but one is significantly less tall than the other.



Subjective interpretations and appropriate units are built right into fuzzy sets. If I say "She's tall," the membership function "tall" should already take into account whether I'm referring to a six-year-old or a grown woman. Similarly, the units are included in the curve. Certainly it makes no sense to say "Is she tall in inches or in meters?"

Membership Functions in the Fuzzy Logic Toolbox

The only condition a membership function must really satisfy is that it must vary between 0 and 1. The function itself can be an arbitrary curve whose

shape we can define as a function that suits us from the point of view of simplicity, convenience, speed, and efficiency.

A classical set might be expressed as

$$A = \{x \mid x > 6\}$$

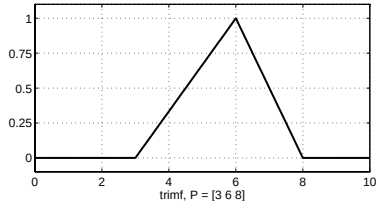
A fuzzy set is an extension of a classical set. If X is the universe of discourse and its elements are denoted by x, then a fuzzy set A in X is defined as a set of ordered pairs:

$$A = \{x, \mu_A(x) \mid x \in X\}$$

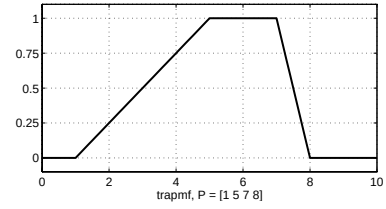
$\mu_A(x)$ is called the membership function (or MF) of x in A. The membership function maps each element of X to a membership value between 0 and 1.

The Fuzzy Logic Toolbox includes 11 built-in membership function types. These 11 functions are, in turn, built from several basic functions: piecewise linear functions, the Gaussian distribution function, the sigmoid curve, and quadratic and cubic polynomial curves. For detailed information on any of the membership functions mentioned below, turn to Chapter 3, “Reference” . By convention, all membership functions have the letters `mf` at the end of their names.

The simplest membership functions are formed using straight lines. Of these, the simplest is the *triangular* membership function, and it has the function name `trimf`. It’s nothing more than a collection of three points forming a triangle. The *trapezoidal* membership function, `trapmf`, has a flat top and really is just a truncated triangle curve. These straight line membership functions have the advantage of simplicity.



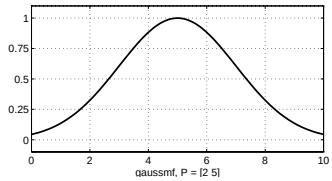
trimf



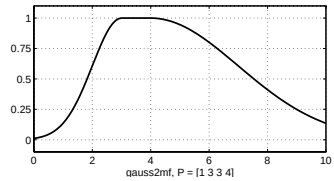
trapmf

Two membership functions are built on the *Gaussian* distribution curve: a simple Gaussian curve and a two-sided composite of two different Gaussian curves. The two functions are `gaussmf` and `gauss2mf`.

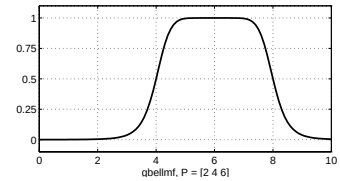
The *generalized bell* membership function is specified by three parameters and has the function name `gbellmf`. The bell membership function has one more parameter than the Gaussian membership function, so it can approach a non-fuzzy set if the free parameter is tuned. Because of their smoothness and concise notation, Gaussian and bell membership functions are popular methods for specifying fuzzy sets. Both of these curves have the advantage of being smooth and nonzero at all points.



gaussmf

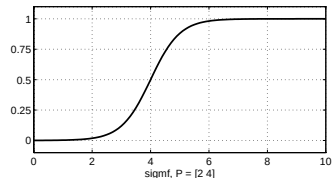


gauss2mf

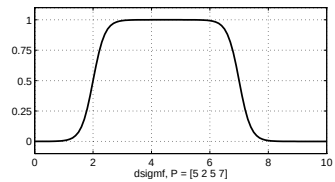


gbellmf

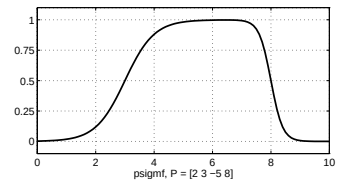
Although the Gaussian membership functions and bell membership functions achieve smoothness, they are unable to specify asymmetric membership functions, which are important in certain applications. Next we define the *sigmoidal* membership function, which is either open left or right. Asymmetric and closed (i.e. not open to the left or right) membership functions can be synthesized using two sigmoidal functions, so in addition to the basic `sigmf`, we also have the difference between two sigmoidal functions, `dsigmf`, and the product of two sigmoidal functions `psigmf`.



sigmf



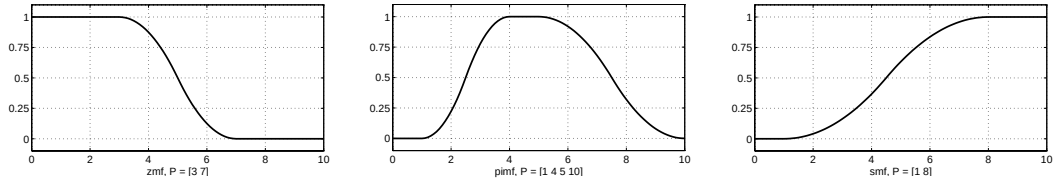
dsigmf



psigmf

Polynomial based curves account for several of the membership functions in the toolbox. Three related membership functions are the *Z, S*, and *Pi curves*, all

named because of their shape. The function `zmf` is the asymmetrical polynomial curve open to the left, `smf` is the mirror-image function that opens to the right, and `pimf` is zero on both extremes with a rise in the middle.



zmf pimf smf

There's a very wide selection to choose from when you're selecting your favorite membership function. And the Fuzzy Logic Toolbox also allows you to create your own membership functions if you find this list too restrictive. On the other hand, if this list seems bewildering, just remember that you could probably get along very well with just one or two types of membership functions, for example the triangle and trapezoid functions. The selection is wide for those who want to explore the possibilities, but exotic membership functions are by no means required for perfectly good fuzzy inference systems. Finally, remember that more details are available on all these functions in the reference section, which makes up the second half of this manual.

Summary of Membership Functions

- Fuzzy sets describe vague concepts (fast runner, hot weather, weekend days).
- A fuzzy set admits the possibility of partial membership in it. (Friday is sort of a weekend day, the weather is rather hot).
- The degree an object belongs to a fuzzy set is denoted by a membership value between 0 and 1. (Friday is a weekend day to the degree 0.8).
- A membership function associated with a given fuzzy set maps an input value to its appropriate membership value.

Logical Operations

We now know what's fuzzy about fuzzy logic, but what about the logic?

The most important thing to realize about fuzzy logical reasoning is the fact that it is a superset of standard Boolean logic. In other words, if we keep the

fuzzy values at their extremes of 1 (completely true), and 0 (completely false), standard logical operations will hold. As an example, consider the standard truth tables below:

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	A or B
0	0	0
0	1	1
1	0	1
1	1	1

OR

A	not A
0	1
1	0

NOT

Now remembering that in fuzzy logic the truth of any statement is a matter of degree, how will these truth tables be altered? The input values can be real numbers between 0 and 1. What function will preserve the results of the AND truth table (for example) and also extend to all real numbers between 0 and 1?

One answer is the *min* operation. That is, resolve the statement *A* AND *B*, where *A* and *B* are limited to the range (0,1), by using the function *min*(*A*,*B*). Using the same reasoning, we can replace the OR operation with the *max* function, so that *A* OR *B* becomes equivalent to *max*(*A*,*B*). Finally, the operation NOT *A* becomes equivalent to the operation $1 - A$. Notice how the truth table above is completely unchanged by this substitution.

A	B	min(A,B)
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	max(A,B)
0	0	0
0	1	1
1	0	1
1	1	1

OR

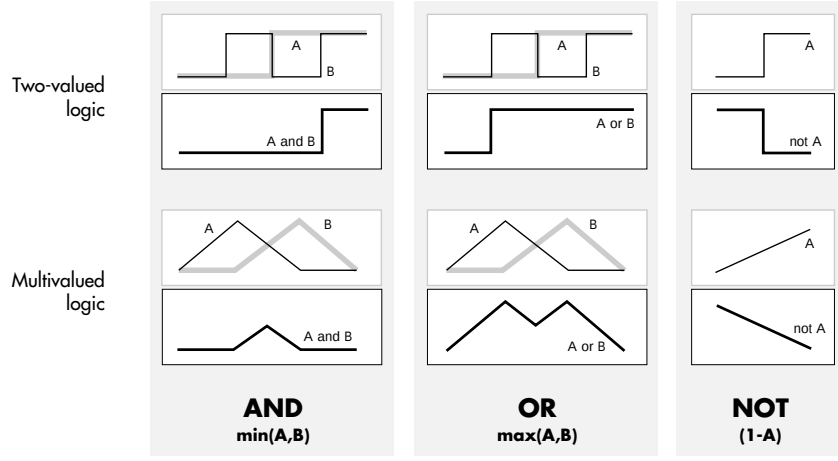
A	1 - A
0	1
1	0

NOT

Moreover, since there is a function behind the truth table rather than just the truth table itself, we can now consider values other than 1 and 0.

The next figure uses a graph to show the same information. We've converted the truth table to a plot of two fuzzy sets applied together to create one fuzzy

set. The upper part of the figure displays plots corresponding to the two-valued truth tables above, while the lower part of the figure displays how the operations work over a continuously varying range of truth values *A* and *B* according to the fuzzy operations we've defined.



Given these three functions, we can resolve any construction using fuzzy sets and the fuzzy logical operation AND, OR, and NOT.

Additional Fuzzy Operators

We've only defined here one particular correspondence between two-valued and multivalued logical operations for AND, OR, and NOT. This correspondence is by no means unique.

In more general terms, we're defining what are known as the fuzzy intersection or conjunction (AND), fuzzy union or disjunction (OR), and fuzzy complement (NOT). We have defined above what we'll call the classical operators for these functions: AND = *min*, OR = *max*, and NOT = additive complement. Typically most fuzzy logic applications make use of these operations and leave it at that. In general, however, these functions are arbitrary to a surprising degree. The Fuzzy Logic Toolbox uses the classical operator for the fuzzy complement as shown above, but also enables you to customize the AND and OR operators.

The intersection of two fuzzy sets *A* and *B* is specified in general by a binary mapping *T*, which aggregates two membership functions as follows:

$$\mu_{A\cap B}(x) = T(\mu_A(x), \mu_B(x))$$

For example, the binary operator *T* may represent the multiplication of $\mu_A(x)$ and $\mu_B(x)$. These fuzzy intersection operators, which are usually referred to as *T*-norm (Triangular norm) operators, meet the following basic requirements.

A *T-norm* operator is a binary mapping $T(\cdot\ ,\ \cdot)$ satisfying:

boundary: $T(0, 0) = 0,\ T(a, 1) = T(1, a) = a$

monotonicity: $T(a, b) <= T(c, d)$ if $a <= c$ and $b <= d$

commutativity: $T(a, b) = T(b, a)$

associativity: $T(a, T(b, c)) = T(T(a, b), c)$

The first requirement imposes the correct generalization to crisp sets. The second requirement implies that a decrease in the membership values in *A* or *B* cannot produce an increase in the membership value in *A* intersection *B*. The third requirement indicates that the operator is indifferent to the order of the fuzzy sets to be combined. Finally, the fourth requirement allows us to take the intersection of any number of sets in any order of pairwise groupings.

Like fuzzy intersection, the fuzzy union operator is specified in general by a binary mapping *S*:

$$\mu_{A\cup B}(x) = S(\mu_A(x), \mu_B(x))$$

For example, the binary operator S can represent the addition of $\mu_A(x)$ and $\mu_B(x)$. These fuzzy union operators, which are often referred to as *T*-conorm (or *S*-norm) operators, must satisfy the following basic requirements.

A *T-conorm* (or *S-norm*) operator is a binary mapping $S(\cdot\ ,\ \cdot)$ satisfying:

boundary: $S(1, 1) = 1,\ S(a, 0) = S(0, a) = a$

monotonicity: $S(a, b) <= S(c, d)$ if $a <= c$ and $b <= d$

commutativity: $S(a, b) = S(b, a)$

associativity: $S(a, S(b, c)) = S(S(a, b), c)$

Several parameterized *T*-norms and dual *T*-conorms have been proposed in the past, such as those of Yager [Yag80], Dubois and Prade [Dub80], Schweizer and Sklar [Sch63], and Sugeno [Sug77]. Each of these provides a way to vary the "gain" on the function so that it can be very restrictive or very permissive.

If-Then Rules

Fuzzy sets and fuzzy operators are the subjects and verbs of fuzzy logic. These if-then rule statements are used to formulate the conditional statements that comprise fuzzy logic.

A single fuzzy if-then rule assumes the form

if *x* is *A* then *y* is *B*

where *A* and *B* are linguistic values defined by fuzzy sets on the ranges (universes of discourse) X and Y, respectively. The if-part of the rule "*x* is *A*" is called the *antecedent* or premise, while the then-part of the rule "*y* is *B*" is called the *consequent* or conclusion. An example of such a rule might be

if service is good then tip is average

Note that *good* is represented as a number between 0 and 1, and so the antecedent is an interpretation that returns a single number between 0 and 1. On the other hand, *average* is represented as a fuzzy set, and so the consequent is an assignment that assigns the entire fuzzy set B to the output variable *y*. In the if-then rule, the word "is" gets used in two entirely different ways depending on whether it appears in the antecedent or the consequent. In MATLAB terms, this is the distinction between a relational test using "==" and a variable assignment using the "=" symbol. A less confusing way of writing the rule would be

if service == good then tip = average

In general, the input to an if-then rule is the current value for the input variable (in this case, *service*) and the output is an entire fuzzy set (in this case, *average*). This set will later be *defuzzified*, assigning one value to the output. The concept of defuzzification is described in the next section, on page 2-41.

Interpreting an if-then rule involves distinct parts: first evaluating the antecedent (which involves *fuzzifying* the input and applying any necessary *fuzzy operators*) and second applying that result to the consequent (known as *implication*). In the case of two-valued or binary logic, if-then rules don't present much difficulty. If the premise is true, then the conclusion is true. If we relax the restrictions of two-valued logic and let the antecedent be a fuzzy statement, how does this reflect on the conclusion? The answer is a simple one:

if the antecedent is true to some degree of membership, then the consequent is also true to that same degree. In other words

in binary logic: $p \rightarrow q$ (p and q are either both true or both false)

in fuzzy logic: $0.5\ p \rightarrow 0.5\ q$ (partial antecedents provide partial implication)

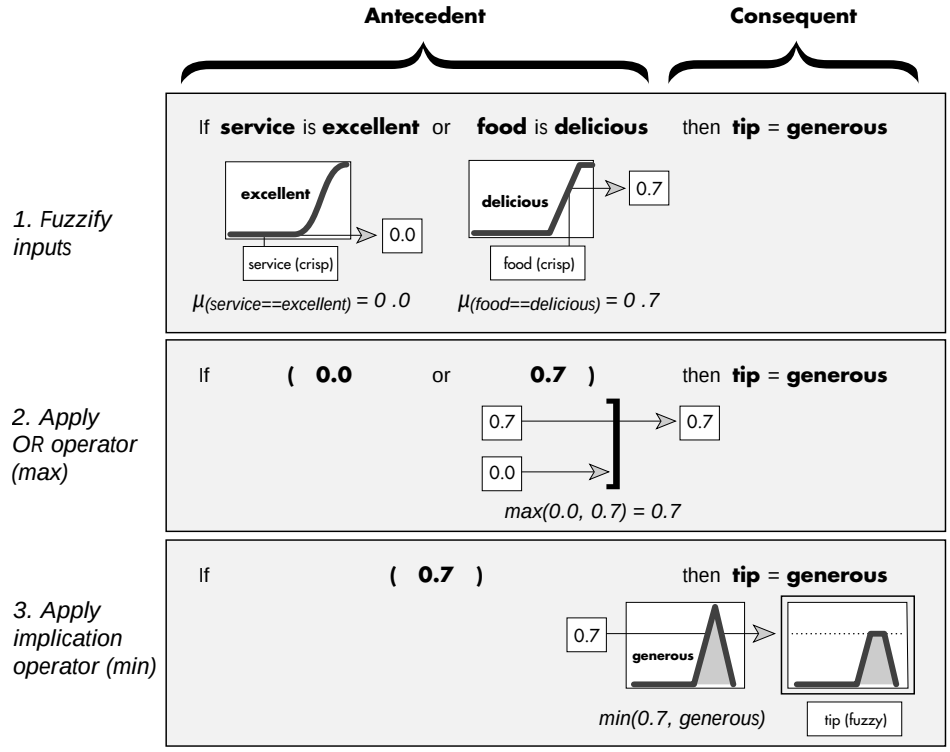
The antecedent of a rule can have multiple parts:

if sky is gray and wind is strong and barometer is falling, then ...

in which case all parts of the antecedent are calculated simultaneously and resolved to a single number using the logical operators described in the preceding section. The consequent of a rule can also have multiple parts:

if temperature is cold then hot water valve is open and cold water valve is shut

in which case all consequents are affected equally by the result of the antecedent. How is the consequent affected by the antecedent? The consequent specifies a fuzzy set be assigned to the output. The *implication function* then modifies that fuzzy set to the degree specified by the antecedent. The most common ways to modify the output fuzzy set are truncation using the *min* function (where the fuzzy set is "chopped off" as shown below) or scaling using the *prod* function (where the output fuzzy set is "squashed"). Both are supported by the Fuzzy Logic Toolbox, but we use truncation for the examples in this section.



Summary of If-Then Rules

Interpreting if-then rules is a three-part process. This process is explained in detail in the next section.

1. *Fuzzify inputs*: Resolve all fuzzy statements in the antecedent to a degree of membership between 0 and 1. If there is only one part to the antecedent, this is the degree of support for the rule.

2. *Apply fuzzy operator to multiple part antecedents*: If there are multiple parts to the antecedent, apply fuzzy logic operators and resolve the antecedent to a single number between 0 and 1. This is the degree of support for the rule.

3. *Apply implication method*: Use the degree of support for the entire rule to shape the output fuzzy set. The consequent of a fuzzy rule assigns an entire fuzzy set to the output. This fuzzy set is represented by a membership function that is chosen to indicate the qualities of the consequent. If the antecedent is only partially true, (i.e., is assigned a value less than 1), then the output fuzzy set is truncated according to the implication method.

In general, one rule by itself doesn't do much good. What's needed are two or more rules that can play off one another. The output of each rule is a fuzzy set. The output fuzzy sets for each rule are then *aggregated* into a single output fuzzy set. Finally the resulting set is *defuzzified*, or resolved to a single number. The next section shows how the whole process works from beginning to end for a particular type of fuzzy inference system called a Mamdani type.

Fuzzy Inference Systems

Fuzzy inference is the process of formulating the mapping from a given input to an output using fuzzy logic. The mapping then provides a basis from which decisions can be made, or patterns discerned. The process of fuzzy inference involves all of the pieces that are described in the previous sections: membership functions, fuzzy logic operators, and if-then rules. There are two types of fuzzy inference systems that can be implemented in the Fuzzy Logic Toolbox: Mamdani-type and Sugeno-type. These two types of inference systems vary somewhat in the way outputs are determined. Descriptions of these two types of fuzzy inference systems can be found in the references, [Jan97, Mam75, Sug85].

Fuzzy inference systems have been successfully applied in fields such as automatic control, data classification, decision analysis, expert systems, and computer vision. Because of its multidisciplinary nature, fuzzy inference systems are associated with a number of names, such as fuzzy-rule-based systems, fuzzy expert systems, fuzzy modeling, fuzzy associative memory, fuzzy logic controllers, and simply (and ambiguously) fuzzy systems. Since the terms used to describe the various parts of the fuzzy inference process are far from standard, we will try to be as clear as possible about the different terms introduced in this section.

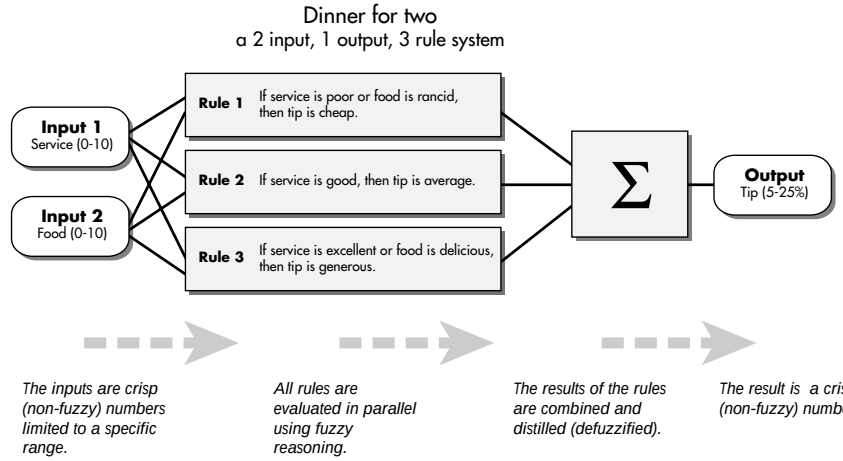
Mamdani's fuzzy inference method is the most commonly seen fuzzy methodology. Mamdani's method was among the first control systems built using fuzzy set theory. It was proposed in 1975 by Ebrahim Mamdani [Mam75] as an attempt to control a steam engine and boiler combination by synthesizing a set of linguistic control rules obtained from experienced human operators. Mamdani's effort was based on Lotfi Zadeh's 1973 paper on fuzzy algorithms for complex systems and decision processes [Zad73]. Although the inference process we describe in the next few sections differs somewhat from the methods described in the original paper, the basic idea is much the same.

Mamdani-type inference, as we have defined it for the Fuzzy Logic Toolbox, expects the output membership functions to be fuzzy sets. After the aggregation process, there is a fuzzy set for each output variable that needs defuzzification. It's possible, and in many cases much more efficient, to use a single spike as the output membership function rather than a distributed fuzzy set. This is sometimes known as a *singleton* output membership function, and it can be thought of as a pre-defuzzified fuzzy set. It enhances the efficiency of the defuzzification process because it greatly simplifies the computation

required by the more general Mamdani method, which finds the centroid of a two-dimensional function. Rather than integrating across the two-dimensional function to find the centroid, we use the weighted average of a few data points. Sugeno-type systems support this type of model. In general, Sugeno-type systems can be used to model any inference system in which the output membership functions are either linear or constant.

Dinner for Two, Reprise

In this section we provide the same two-input one-output three-rule tipping problem that you saw in the introduction, only in more detail. The basic structure of this example is shown in the diagram below.



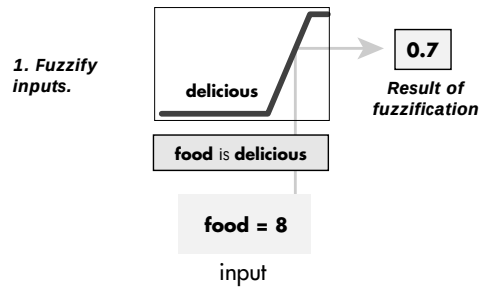
Information flows from left to right, from two inputs to a single output. The parallel nature of the rules is one of the more important aspects of fuzzy logic systems. Instead of sharp switching between modes based on breakpoints, we will glide smoothly from regions where the system's behavior is dominated by either one rule or another.

In the Fuzzy Logic Toolbox, there are five parts of the fuzzy inference process: fuzzification of the input variables, application of the fuzzy operator (AND or OR) in the antecedent, implication from the antecedent to the consequent, aggregation of the consequents across the rules, and defuzzification. These sometimes cryptic and odd names have very specific meaning that we'll define carefully as we step through each of them in more detail below.

Step 1. Fuzzify Inputs

The first step is to take the inputs and determine the degree to which they belong to each of the appropriate fuzzy sets via membership functions. In the Fuzzy Logic Toolbox, the input is always a crisp numerical value limited to the universe of discourse of the input variable (in this case the interval between 0 and 10) and the output is a fuzzy degree of membership in the qualifying linguistic set (always the interval between 0 and 1). Fuzzification of the input amounts to either a table lookup or a function evaluation.

The example we're using in this section is built on three rules, and each of the rules depends on resolving the inputs into a number of different fuzzy linguistic sets: service is poor, service is good, food is rancid, food is delicious, and so on. Before the rules can be evaluated, the inputs must be fuzzified according to each of these linguistic sets. For example, to what extent is the food really delicious? The figure below shows how well the food at our hypothetical restaurant (rated on a scale of 0 to 10) qualifies, (via its membership function), as the linguistic variable "delicious." In this case, we rated the food as an 8, which, given our graphical definition of delicious, corresponds to $\mu = 0.7$ for the "delicious" membership function.



(The compliment to the chef would be "your food is delicious to the degree 0.7.") In this manner, each input is fuzzified over all the qualifying membership functions required by the rules.

Step 2. Apply Fuzzy Operator

Once the inputs have been fuzzified, we know the degree to which each part of the antecedent has been satisfied for each rule. If the antecedent of a given rule has more than one part, the fuzzy operator is applied to obtain one number that represents the result of the antecedent for that rule. This number will then be applied to the output function. The input to the fuzzy operator is two or more

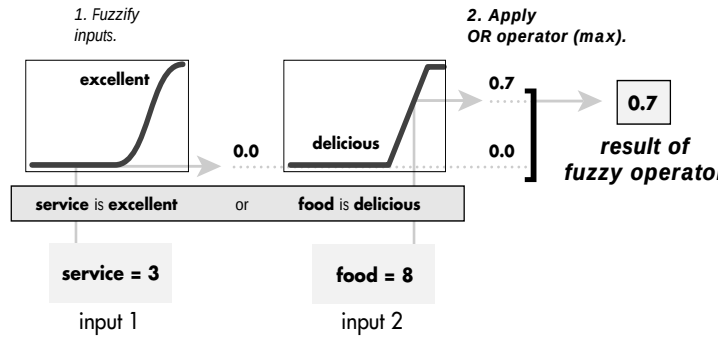
membership values from fuzzified input variables. The output is a single truth value.

As is described in the section on fuzzy logical operations, any number of well-defined methods can fill in for the AND operation or the OR operation. In the Fuzzy Logic Toolbox, two built-in AND methods are supported: *min* (minimum) and *prod* (product). Two built-in OR methods are also supported: *max* (maximum), and the probabilistic OR method *probor*. The probabilistic OR method (also known as the algebraic sum) is calculated according to the equation

$$probor(a,b) = a + b - ab$$

In addition to these built-in methods, you can create your own methods for AND and OR by writing any function and setting that to be your method of choice. There will be more information on how to do this later.

Shown below is an example of the OR operator *max* at work. We're evaluating the antecedent of the rule 3 for the tipping calculation. The two different pieces of the antecedent (service is excellent and food is delicious) yielded the fuzzy membership values 0.0 and 0.7 respectively. The fuzzy OR operator simply selects the maximum of the two values, 0.7, and the fuzzy operation for rule 3 is complete. If we were using the probabilistic OR method, the result would still be 0.7 in this case.

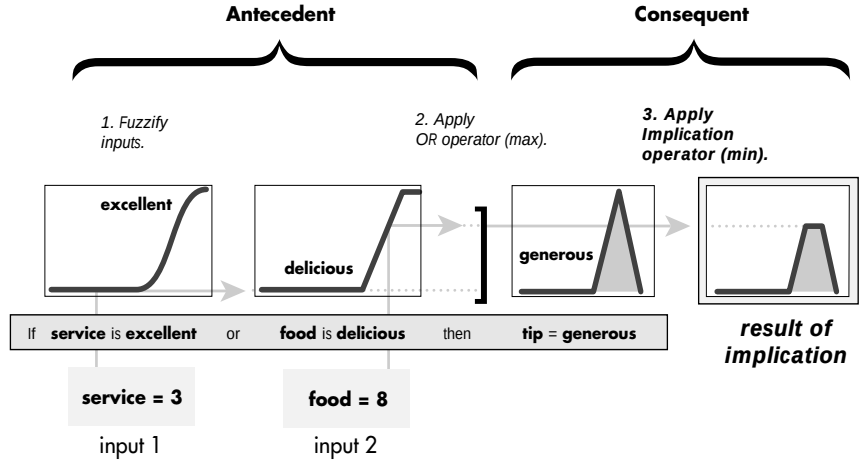


Step 3. Apply Implication Method

Before applying the implication method, we must take care of the rule's weight. Every rule has a *weight* (a number between 0 and 1), which is applied to the number given by the antecedent. Generally this weight is 1 (as it is for this example) and so it has no effect at all on the implication process. From time to

time you may want to weight one rule relative to the others by changing its weight value to something other than 1.

Once proper weighting has been assigned to each rule, the implication method is implemented. A consequent is a fuzzy set represented by a membership function, which weights appropriately the linguistic characteristics that are attributed to it. The consequent is reshaped using a function associated with the antecedent (a single number). The input for the implication process is a single number given by the antecedent, and the output is a fuzzy set. Implication is implemented for each rule. Two built-in methods are supported, and they are the same functions that are used by the AND method: *min* (minimum), which truncates the output fuzzy set, and *prod* (product), which scales the output fuzzy set.

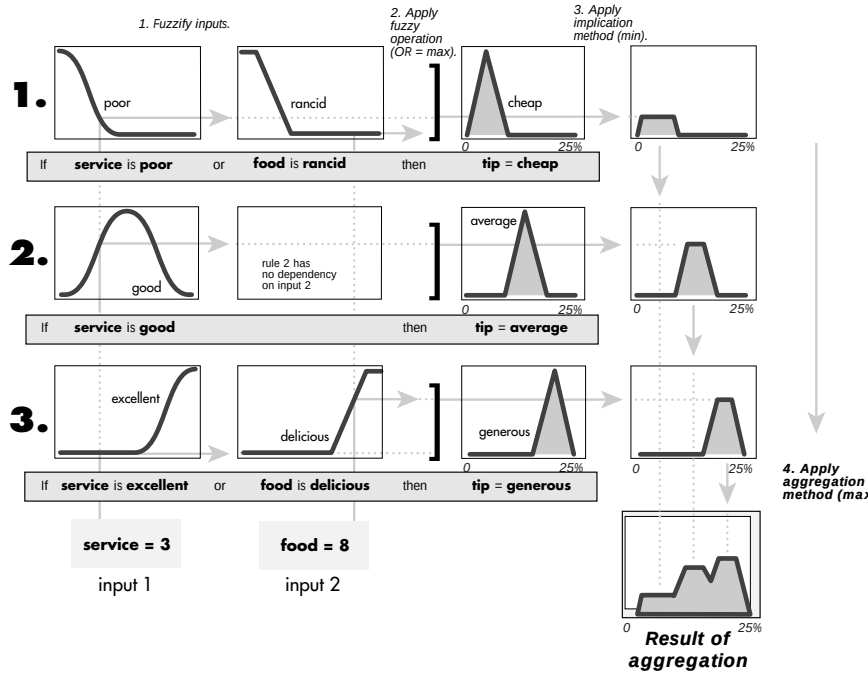


Step 4. Aggregate All Outputs

Since decisions are based on the testing of all of the rules in an FIS, the rules must be combined in some manner in order to make a decision. Aggregation is the process by which the fuzzy sets that represent the outputs of each rule are combined into a single fuzzy set. Aggregation only occurs once for each output variable, just prior to the fifth and final step, defuzzification. The input of the aggregation process is the list of truncated output functions returned by the implication process for each rule. The output of the aggregation process is one fuzzy set for each output variable.

Notice that as long as the aggregation method is commutative (which it always should be), then the order in which the rules are executed is unimportant. Three built-in methods are supported: *max* (maximum), *probor* (probabilistic or), and *sum* (simply the sum of each rule's output set).

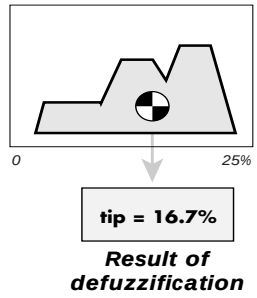
In the diagram below, all three rules have been placed together to show how the output of each rule is combined, or aggregated, into a single fuzzy set whose membership function assigns a weighting for every output (tip) value.



Step 5. Defuzzify

The input for the defuzzification process is a fuzzy set (the aggregate output fuzzy set) and the output is a single number. As much as fuzziness helps the rule evaluation during the intermediate steps, the final desired output for each variable is generally a single number. However, the aggregate of a fuzzy set encompasses a range of output values, and so must be defuzzified in order to resolve a single output value from the set.

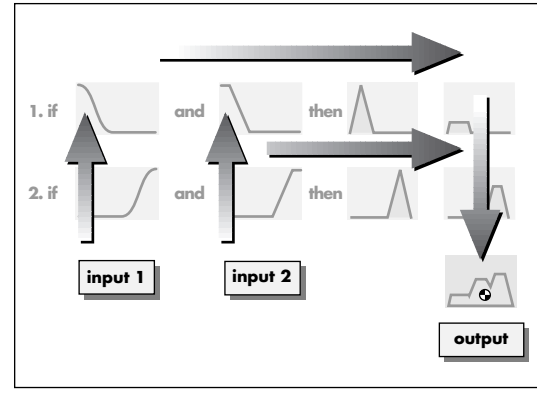
Perhaps the most popular defuzzification method is the centroid calculation, which returns the center of area under the curve. There are five built-in methods supported: centroid, bisector, middle of maximum (the average of the maximum value of the output set), largest of maximum, and smallest of maximum.



The Fuzzy Inference Diagram

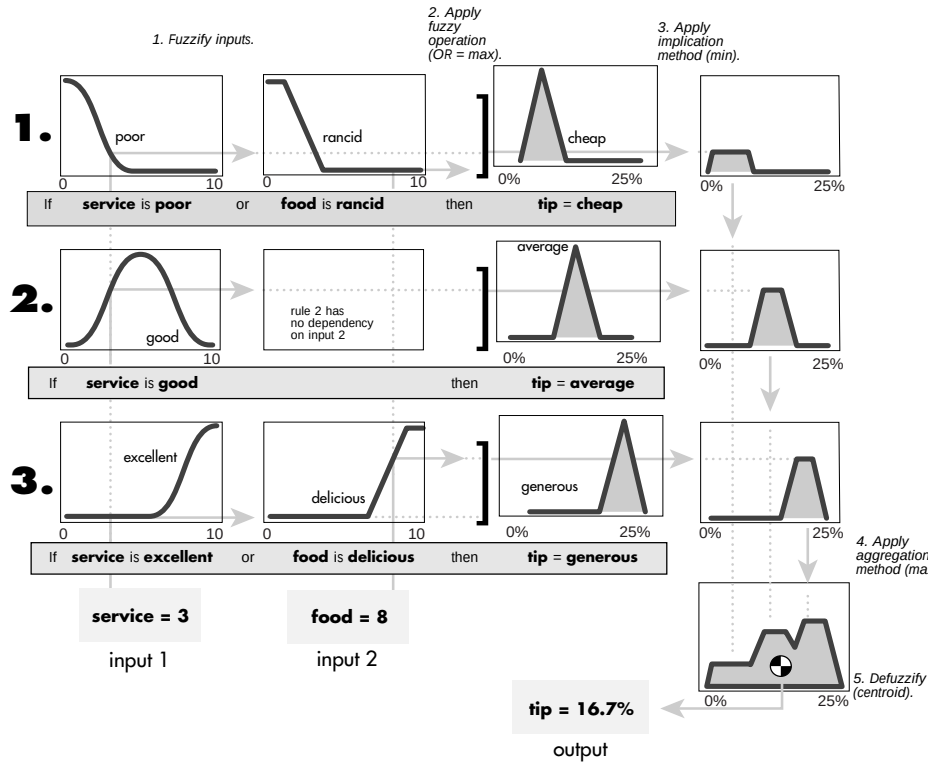
The fuzzy inference diagram is the composite of all the smaller diagrams we've been looking at so far in this section. It simultaneously displays all parts of the fuzzy inference process we've examined. Information flows through the fuzzy inference diagram as shown below.

Interpreting the Fuzzy Inference Diagram



Notice how the flow proceeds up from the inputs in the lower left, then across each row, or rule, and then down the rule outputs to finish in the lower right. This is a very compact way of showing everything at once, from linguistic variable fuzzification all the way through defuzzification of the aggregate output.

Shown below is the real full-size fuzzy inference diagram. There's a lot to see in a fuzzy inference diagram, but once you become accustomed to it, you can learn a lot about a system very quickly. For instance, from this diagram with these particular inputs, you can easily see that the implication method is truncation with the *min* function. The *max* function is being used for the fuzzy OR operation. Rule 3 (the bottom-most row in the diagram shown opposite) is having the strongest influence on the output. And so on. The Rule Viewer described in "The Rule Viewer" on page 2-59 is a MATLAB implementation of the fuzzy inference diagram.



Customization

One of the primary goals of the Fuzzy Logic Toolbox is to have an open and easily modified fuzzy inference system structure. Thus, the Fuzzy Logic Toolbox is designed to give you as much freedom as possible, within the basic

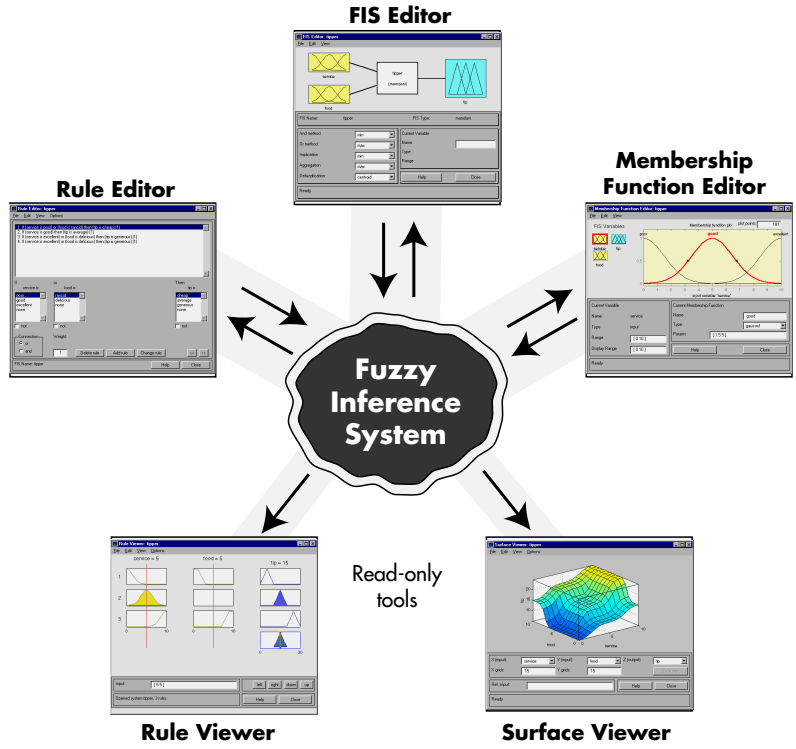
constraints of the process described here, to customize the fuzzy inference process for your application. For example, you can substitute your own MATLAB functions for any of the default functions used in the five steps detailed above: you make your own membership functions, AND methods, OR methods, implication methods, aggregation methods, and defuzzification methods. The next section describes exactly how to build and implement a fuzzy inference system using the tools provided.

Building Systems with the Fuzzy Logic Toolbox

Dinner for Two, from the Top

Now we're going to work through a similar tipping example, only we'll be building it using the graphical user interface (GUI) tools provided by the Fuzzy Logic Toolbox. Although it's possible to use the Fuzzy Logic Toolbox by working strictly from the command line, in general it's much easier to build a system graphically. There are five primary GUI tools for building, editing, and observing fuzzy inference systems in the Fuzzy Logic Toolbox: the Fuzzy Inference System or FIS Editor, the Membership Function Editor, the Rule Editor, the Rule Viewer, and the Surface Viewer. These GUIs are dynamically linked, in that changes you make to the FIS using one of them, can affect what you see on any of the other open GUIs. You can have any or all of them open for any given system.

In addition to these five primary GUIs, the toolbox includes the graphical ANFIS Editor GUI, which is used for building and analyzing Sugeno-type adaptive neural fuzzy inference systems. The ANFIS Editor GUI is discussed later in this chapter, in the section, “Sugeno-Type Fuzzy Inference” on page 2-86.



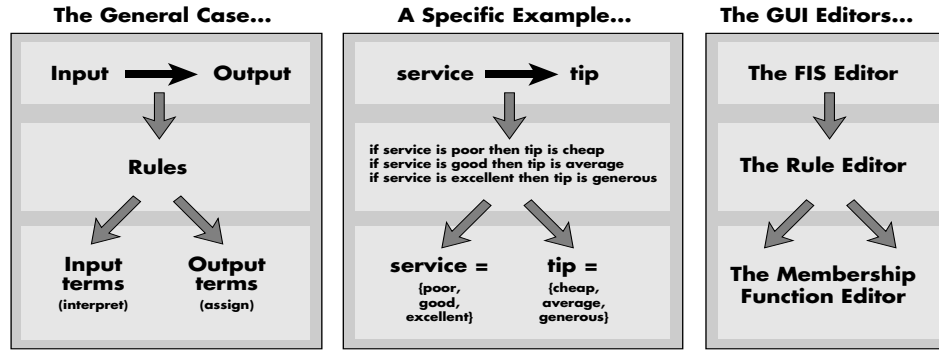
The FIS Editor handles the high level issues for the system: How many input and output variables? What are their names? The Fuzzy Logic Toolbox doesn't limit the number of inputs. However, the number of inputs may be limited by the available memory of your machine. If the number of inputs is too large, or the number of membership functions is too big, then it may also be difficult to analyze the FIS using the other GUI tools.

The Membership Function Editor is used to define the shapes of all the membership functions associated with each variable.

The Rule Editor is for editing the list of rules that defines the behavior of the system.

The Rule Viewer and the Surface Viewer are used for looking at, as opposed to editing, the FIS. They are strictly read-only tools. The Rule Viewer is a MATLAB-based display of the fuzzy inference diagram shown at the end of the last section. Used as a diagnostic, it can show (for example) which rules are active, or how individual membership function shapes are influencing the results. The Surface Viewer is used to display the dependency of one of the outputs on any one or two of the inputs—that is, it generates and plots an output surface map for the system.

This chapter began with an illustration similar to the one below describing the main parts of a fuzzy inference system, only the one below shows how the three Editors fit together. The two Viewers examine the behavior of the entire system.



The five primary GUIs can all interact and exchange information. Any one of them can read and write both to the workspace and to the disk (the *read-only* viewers can still exchange plots with the workspace and/or the disk). For any fuzzy inference system, any or all of these five GUIs may be open. If more than one of these editors is open for a single system, the various GUI windows are aware of the existence of the others, and will, if necessary, update related windows. Thus if the names of the membership functions are changed using the Membership Function Editor, those changes are reflected in the rules shown in the Rule Editor. The editors for any number of different FIS systems may be open simultaneously. The FIS Editor, the Membership Function Editor, and the Rule Editor can all read and modify the FIS data, but the Rule Viewer and the Surface Viewer do not modify the FIS data in any way.

Getting Started

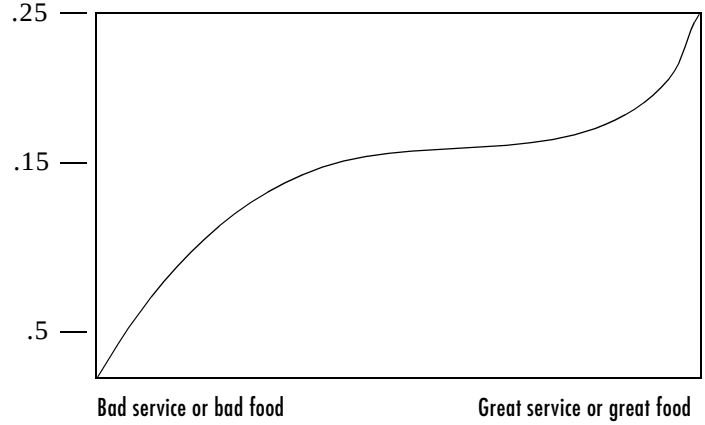
We'll start with a basic description of a two-input, one-output tipping problem (based on tipping practices in the U.S.).

The Basic Tipping Problem. Given a number between 0 and 10 that represents the quality of service at a restaurant (where 10 is excellent), and another number between 0 and 10 that represents the quality of the food at that restaurant (again, 10 is excellent), what should the tip be?

The starting point is to write down the three golden rules of tipping, based on years of personal experience in restaurants.

1. *If the service is poor or the food is rancid, then tip is cheap.*
2. *If the service is good, then tip is average.*
3. *If the service is excellent or the food is delicious, then tip is generous.*

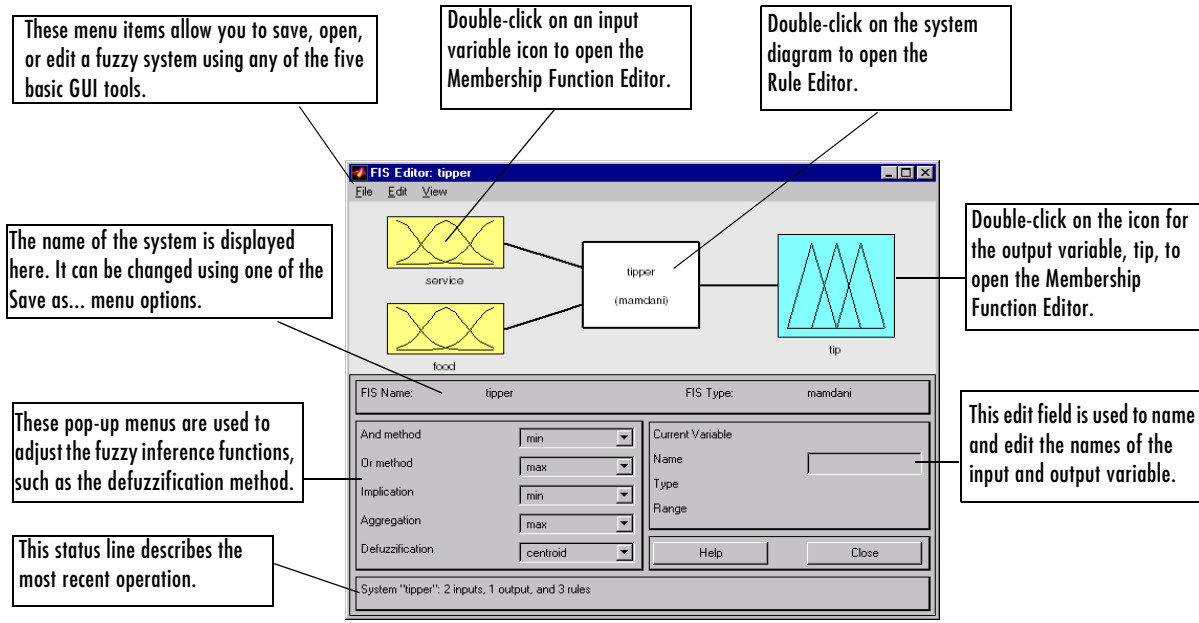
We'll assume that an average tip is 15%, a generous tip is 25%, and a cheap tip is 5%. It's also useful to have a vague idea of what the tipping function should look like.



Obviously the numbers and the shape of the curve are subject to local traditions, cultural bias, and so on, but the three rules are pretty universal.

Now we know the rules, and we have an idea of what the output should look like. Let's begin working with the GUI tools to construct a fuzzy inference system for this decision process.

The FIS Editor



The following discussion walks you through building a new fuzzy inference system from scratch. If you want to save time and follow along quickly, you can load the already built system by typing

```
fuzzy tipper
```

This will load the FIS associated with the file `tipper.fis` (the `.fis` is implied) and launch the FIS Editor. However, if you load the pre-built system, you will not be building rules and constructing membership functions.

The FIS Editor displays general information about a fuzzy inference system. There's a simple diagram at the top that shows the names of each input variable on the left, and those of each output variable on the right. The sample membership functions shown in the boxes are just icons and do not depict the actual shapes of the membership functions.

Below the diagram is the name of the system and the type of inference used. The default, Mamdani-type inference, is what we've been describing so far and

what we'll continue to use for this example. Another slightly different type of inference, called Sugeno-type inference, is also available. This method is explained in "Sugeno-Type Fuzzy Inference" on page 2-86. Below the name of the fuzzy inference system, on the left side of the figure, are the pop-up menus that allow you to modify the various pieces of the inference process. On the right side at the bottom of the figure is the area that displays the name of either an input or output variable, its associated membership function type, and its range. The latter two fields are specified only after the membership functions have been. Below that region are the **Help** and **Close** buttons that call up online help and close the window, respectively. At the bottom is a status line that relays information about the system.

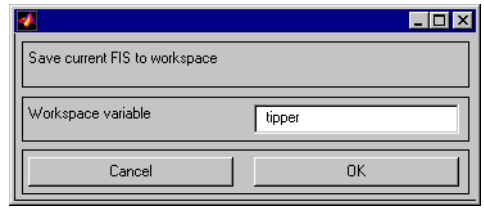
To start this system from scratch, type

```
fuzzy
```

at the MATLAB prompt. The generic untitled FIS Editor opens, with one input, labeled **input1**, and one output, labeled **output1**. For this example, we will construct a two-input, one output system, so go to the **Edit** menu and select **Add input**. A second yellow box labeled **input2** will appear. The two inputs we will have in our example are **service** and **food**. Our one output is **tip**. We'd like to change the variable names to reflect that, though.

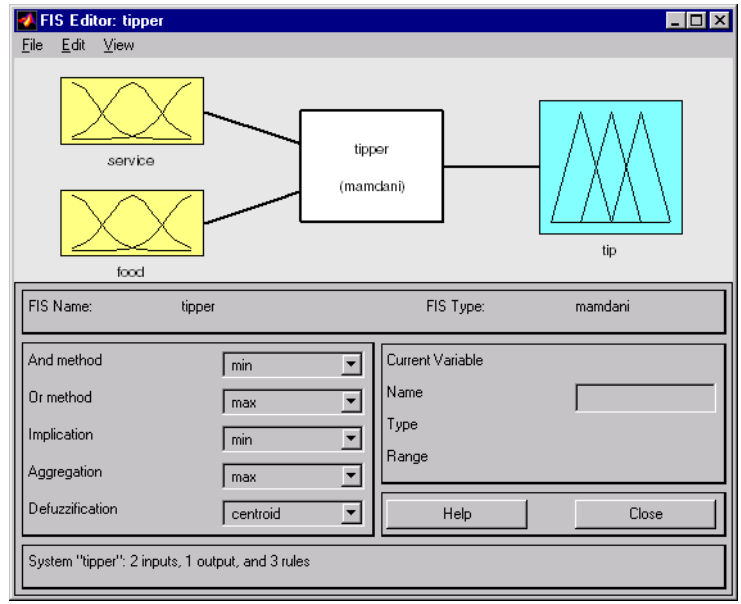
1. Click once on the left-hand (yellow) box marked **input1** (the box will be highlighted in red).
2. In the white edit field on the right, change `input1` to `service` and press **Return.**
3. Click once on the left-hand (yellow) box marked **input2** (the box will be highlighted in red).
4. In the white edit field on the right, change `input2` to `food` and press **Return**.
5. Click once on the right-hand (blue) box marked **output1**.
6. In the white edit field on the right, change `output1` to `tip`.

7 From the **File** menu select **Save to workspace as...**.



8 Enter the variable name `tipper` and click on **OK.**

You will see the diagram updated to reflect the new names of the input and output variables. There is now a new variable in the workspace called `tipper` that contains all the information about this system. By saving to the workspace with a new name, you also rename the entire system. Your window will look something like this.

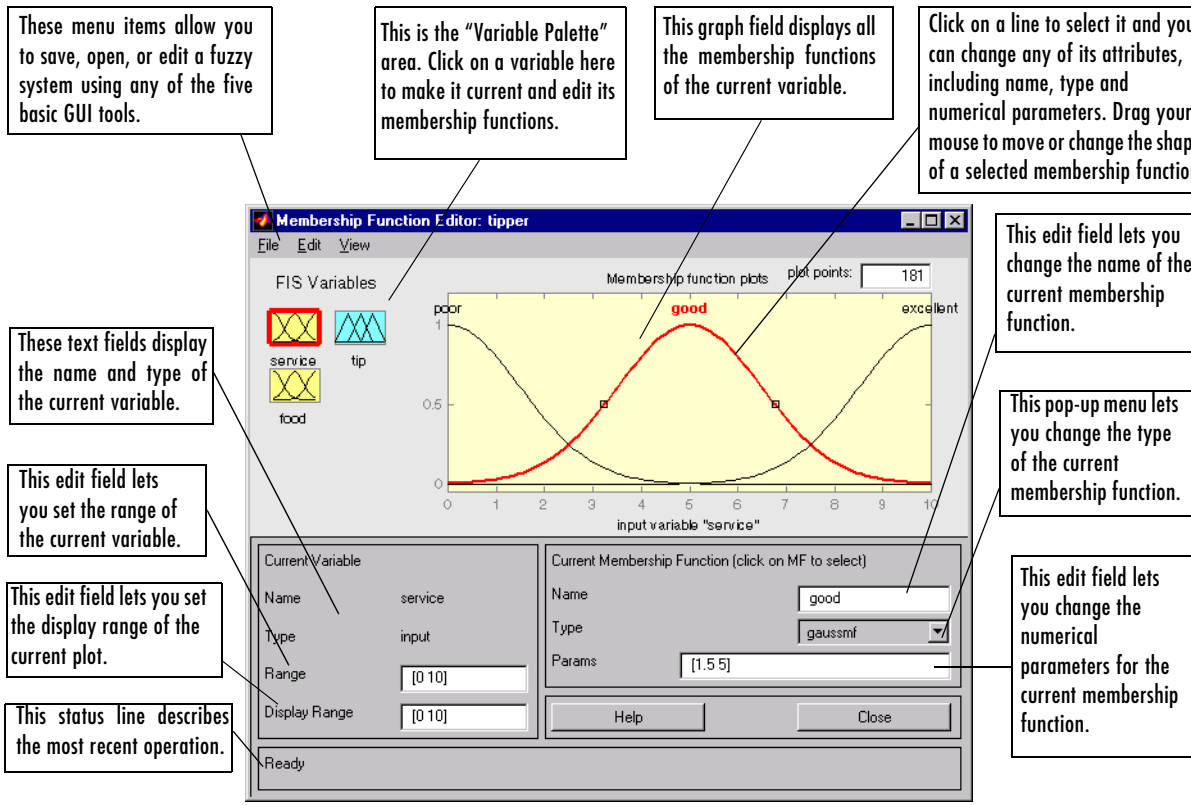


Leave the inference options in the lower left in their default positions for now. You've entered all the information you need for this particular GUI. Next define the membership functions associated with each of the variables. To do

this, open the Membership Function Editor. You can open the Membership Function Editor in one of three ways:

- Pull down the **View** menu item and select **Edit Membership Functions...**.
- Double-click on the icon for the output variable, **tip**.
- Type `mfedit` at the command line.

The Membership Function Editor



The Membership Function Editor shares some features with the FIS Editor. In fact, all of the five basic GUI tools have similar menu options, status lines, and **Help** and **Close** buttons. The Membership Function Editor is the tool that lets

you display and edit all of the membership functions associated with all of the input and output variables for the entire fuzzy inference system.

When you open the Membership Function Editor to work on a fuzzy inference system that does not already exist in the workspace, there are not yet any membership functions associated with the variables that you have just defined with the FIS Editor.

On the upper left side of the graph area in the Membership Function Editor is a "Variable Palette" that lets you set the membership functions for a given variable.To set up your membership functions associated with an input or an output variable for the FIS, select an FIS variable in this region by clicking on it.

Next select the **Edit** pull-down menu, and choose **Add MFs...**. A new window will appear, which allows you to select both the membership function type and the number of membership functions associated with the selected variable. In the lower right corner of the window are the controls that let you change the name, type, and parameters (shape), of the membership function, once it has been selected.

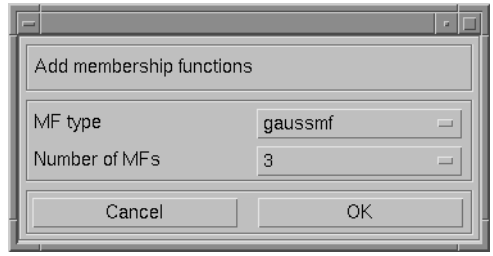
The membership functions from the current variable are displayed in the main graph. These membership functions can be manipulated in two ways. You can first use the mouse to select a particular membership function associated with a given variable quality, (such as poor, for the variable, service), and then drag the membership function from side to side. This will affect the mathematical description of the quality associated with that membership function for a given variable. The selected membership function can also be tagged for dilation or contraction by clicking on the small square drag points on the membership function, and then dragging the function with the mouse toward the *outside*, for dilation, or toward the *inside*, for contraction. This will change the parameters associated with that membership function.

Below the Variable Palette is some information about the type and name of the current variable. There is a text field in this region that lets you change the limits of the current variable's range (universe of discourse) and another that lets you set the limits of the current plot (which has no real effect on the system).

The process of specifying the input membership functions for this two input tipper problem is as follows:

1 Select the input variable, **service**, by double-clicking on it. Set both the **Range** and the **Display Range** to the vector `[0 10]`.

2 Select **Add MFs...** from the **Edit** menu. The window below pops open.



3 Use the pull-down tab to choose **gaussmf** for **MF Type** and **3** for **Number of MFs**. This adds three Gaussian curves to the input variable service

4 Click once on the curve with the leftmost *hump*. Change the name of the curve to `poor`. To adjust the shape of the membership function, either use the mouse, as described above, or type in a desired parameter change, and then click on the membership function. The default parameter listing for this curve is `[1.5 0]`.

5 Name the curve with the middle hump, `good`, and the curve with the rightmost hump, `excellent`. Reset the associated parameters if desired.

6 Select the input variable, **food**, by clicking on it. Set both the **Range** and the **Display Range** to the vector `[0 10]`.

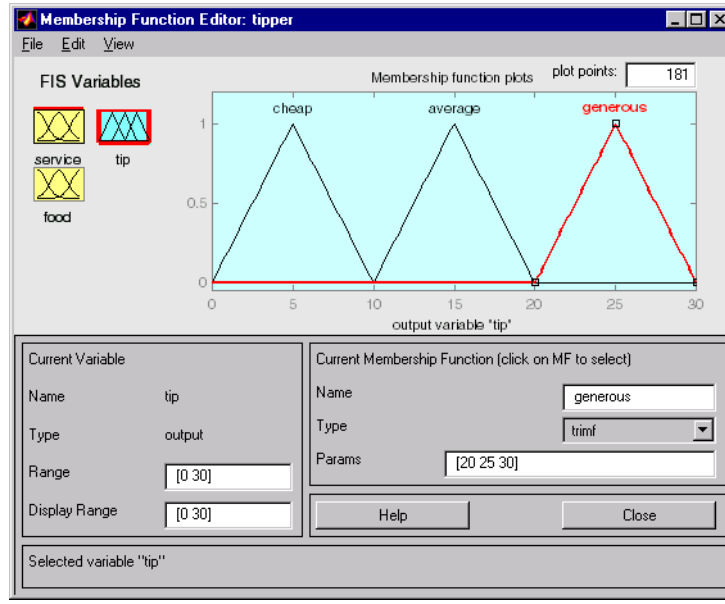
7 Select **Add MFs...** from the **Edit** menu and add two `trapmf` curves to the input variable `food`.

8 Click once directly on the curve with the leftmost trapazoid. Change the name of the curve to `rancid`. To adjust the shape of the membership function, either use the mouse, as described above, or type in a desired parameter change, and then click on the membership function. The default parameter listing for this curve is `[0 0 1 3]`.

9 Name the curve with the rightmost trapazoid, `delicious`, and reset the associated parameters if desired

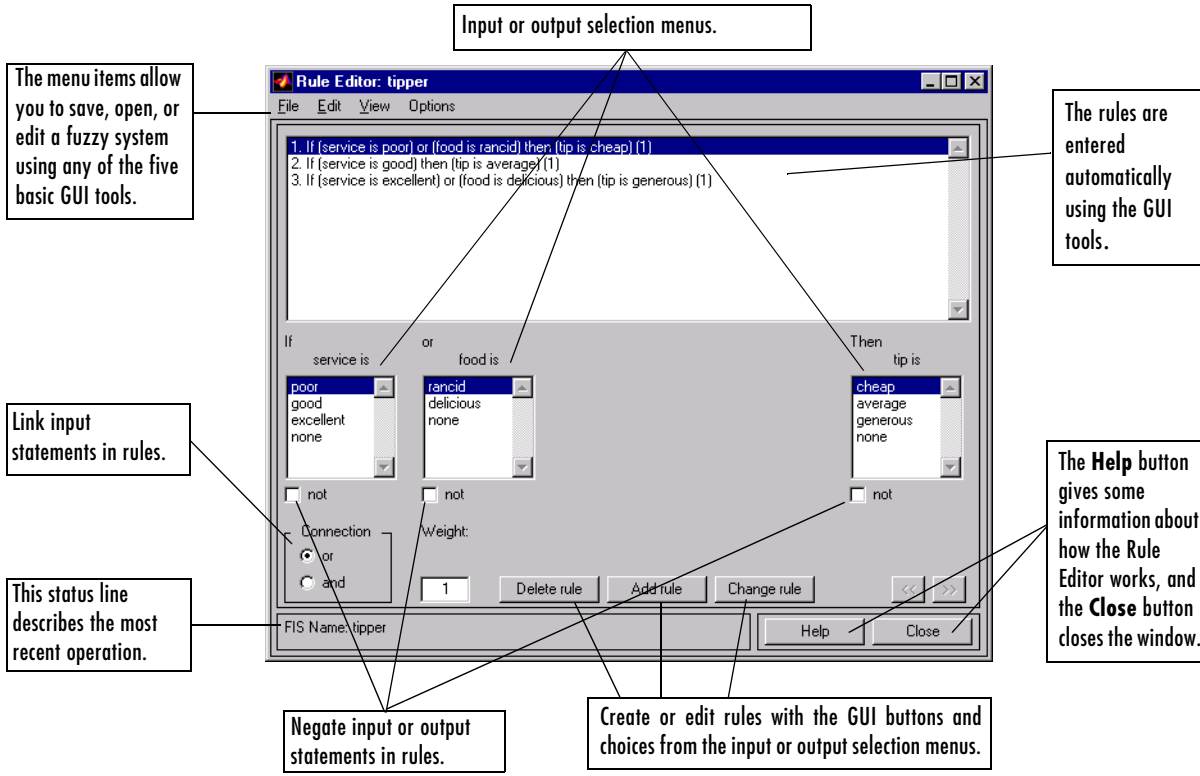
Next you need to create the membership functions for the output variable, **tip**. To create the output variable membership functions, use the Variable Palette on the left, selecting the output variable, **tip**. The inputs ranged from 0 to 10, but the output scale is going to be a tip between 5 and 25 percent.

Use triangular membership function types for the output. First, set the **Range** (and the **Display Range**) to `[0 30]`, to cover the output range. Initially, the *cheap* membership function will have the parameters `[0 5 10]`, the *average* membership function will be `[10 15 20]`, and the *generous* membership function will be `[20 25 30]`. Your system should look something like this.



Now that the variables have been named, and the membership functions have appropriate shapes and names, you're ready to write down the rules. To call up the Rule Editor, go to the **View** menu and select **Edit rules...**, or type `ruleedit` at the command line.

The Rule Editor



Constructing rules using the graphical Rule Editor interface is fairly self-evident. Based on the descriptions of the input and output variables defined with the FIS Editor, the Rule Editor allows you to construct the rule statements automatically, by clicking on and selecting one item in each input variable box, one item in each output box, and one connection item. Choosing **none** as one of the variable qualities will exclude that variable from a given rule. Choosing **not** under any variable name will negate the associated quality. Rules may be changed, deleted, or added, by clicking on the appropriate button.

The Rule Editor also has some familiar landmarks, similar to those in the FIS Editor and the Membership Function Editor, including the menu bar and the status line. The **Format** pop-up menu is available from the **Options** pull-down menu from the top menu bar—this is used to set the format for the display.

Similarly, **Language** can be set from under **Options** as well. The **Help** button will bring up a MATLAB Help window.

To insert the first rule in the Rule Editor, select the following:

- **poor** under the variable **service**
- **rancid** under the variable **food**
- the radio button, **or**, in the **Connection** block
- **cheap**, under the output variable, **tip**.

The resulting rule is:

1. *If (service is poor) or (food is rancid) then (tip is cheap) (1)*

The numbers in the parentheses represent weights that can be applied to each rule if desired. You can specify the weights by typing in a desired number between zero and one under the **Weight:** setting. If you do not specify them, the weights are assumed to be unity (1).

Follow a similar procedure to insert the second and third rules in the Rule Editor to get:

1. *If (service is poor) or (food is rancid) then (tip is cheap) (1)*
2. *If (service is good) then (tip is average) (1)*
3. *If (service is excellent) or (food is delicious) then (tip is generous) (1)*

To change a rule, first click on the rule to be changed. Next make the desired changes to that rule, and then click on **Change rule**. For example, to change the first rule to

1. *If (service not poor) or (food not rancid) then (tip is not cheap) (1)*

click **not** under each variable, and then click **Change rule**.

The **Format** pop-up menu from the **Options** menu indicates that you're looking at the verbose form of the rules. Try changing it to **symbolic**. You will see

1. *(service==poor) => (tip=cheap) (1)*
2. *(service==good) => (tip=average) (1)*
3. *(service==excellent) => (tip=generous) (1)*

There is not much difference in the display really, but it's slightly more language neutral, since it doesn't depend on terms like "if" and "then." If you change the format to indexed, you'll see an extremely compressed version of the rules that has squeezed all the language out.

1, 1 (1) : 1

2, 2 (1) : 1

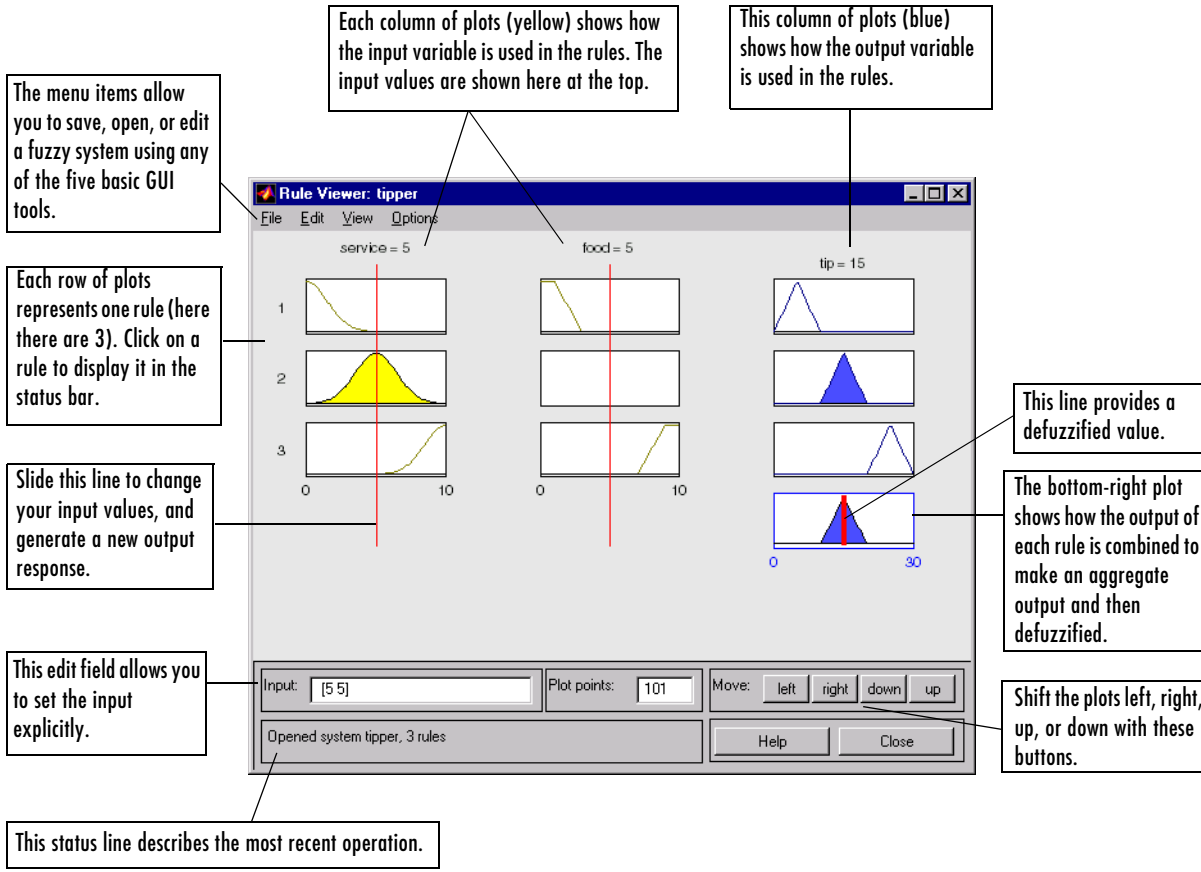
3, 3 (1) : 1

This is the version that the machine deals with. The first column in this structure corresponds to the input variable, the second column corresponds to the output variable, the third column displays the weight applied to each rule, and the fourth column is shorthand that indicates whether this is an OR (2) rule or an AND (1) rule. The numbers in the first two columns refer to the index number of the membership function. A literal interpretation of rule 1 is: "if input 1 is MF1 (the first membership function associated with input 1) then output 1 should be MF1 (the first membership function associated with output 1) with the weight 1." Since there is only one input for this system, the AND connective implied by the 1 in the last column is of no consequence.

The symbolic format doesn't bother with the terms, *if*, *then*, and so on. The indexed format doesn't even bother with the names of your variables. Obviously the functionality of your system doesn't depend on how well you have named your variables and membership functions. The whole point of naming variables descriptively is, as always, making the system easier for you to interpret. Thus, unless you have some special purpose in mind, it will probably be easier for you to stick with the **verbose** format.

At this point, the fuzzy inference system has been completely defined, in that the variables, membership functions, and the rules necessary to calculate tips are in place. It would be nice, at this point, to look at a fuzzy inference diagram like the one presented at the end of the previous section and verify that everything is behaving the way we think it should. This is exactly the purpose of the Rule Viewer, the next of the GUI tools we'll look at. From the **View** menu, select **View rules...**.

The Rule Viewer



The Rule Viewer displays a roadmap of the whole fuzzy inference process. It's based on the fuzzy inference diagram described in the previous section. You see a single figure window with 10 small plots nested in it. The three small plots across the top of the figure represent the antecedent and consequent of the first rule. Each rule is a row of plots, and each column is a variable. The first two columns of plots (the six yellow plots) show the membership functions referenced by the antecedent, or the if-part of each rule. The third column of plots (the three blue plots) shows the membership functions referenced by the consequent, or the then-part of each rule. If you click once on a rule number, the corresponding rule will be displayed at the bottom of the figure. Notice that

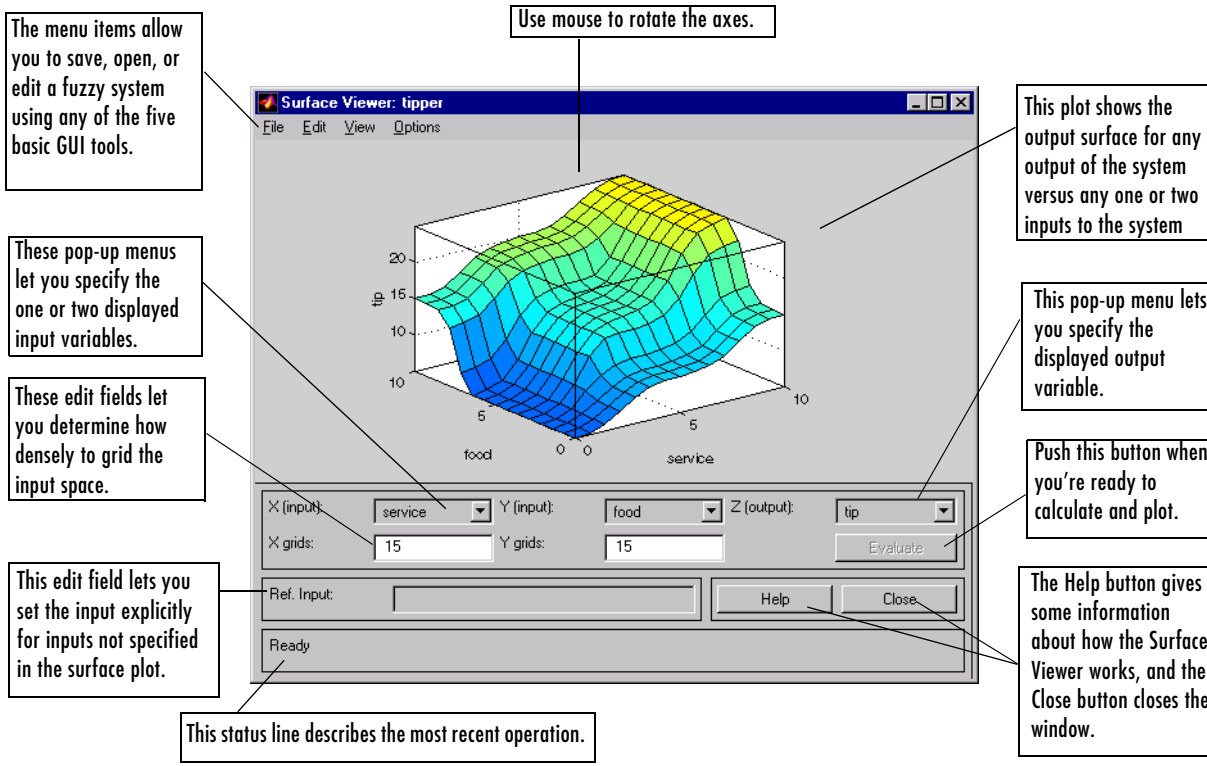
under **food**, there is a plot which is blank. This corresponds to the characterization of **none** for the variable **food** in the second rule. The fourth plot in the third column of plots represents the aggregate weighted decision for the given inference system. This decision will depend on the input values for the system.

There are also the now familiar items like the status line and the menu bar. In the lower right there is a text field into which you can enter specific input values. For the two-input system, you will enter an input vector, `[9 8]`, for example, and then click on **input**. You can also adjust these input values by clicking anywhere on any of the three plots for each input. This will move the red index line horizontally, to the point where you have clicked. You can also just click and drag this line in order to change the input values. When you release the line, (or after manually specifying the input), a new calculation is performed, and you can see the whole fuzzy inference process take place. Where the index line representing service crosses the membership function line "service is poor" in the upper left plot will determine the degree to which rule one is activated. A yellow patch of color under the actual membership function curve is used to make the fuzzy membership value visually apparent. Each of the characterizations of each of the variables is specified with respect to the input index line in this manner. If we follow rule 1 across the top of the diagram, we can see the consequent "tip is cheap" has been truncated to exactly the same degree as the (composite) antecedent—this is the implication process in action. The aggregation occurs down the third column, and the resultant aggregate plot is shown in the single plot to be found in the lower right corner of the plot field. The defuzzified output value is shown by the thick line passing through the aggregate fuzzy set.

The Rule Viewer allows you to interpret the entire fuzzy inference process at once. The Rule Viewer also shows how the shape of certain membership functions influences the overall result. Since it plots every part of every rule, it can become unwieldy for particularly large systems, but, for a relatively small number of inputs and outputs, it performs well (depending on how much screen space you devote to it) with up to 30 rules and as many as 6 or 7 variables.

The Rule Viewer shows one calculation at a time and in great detail. In this sense, it presents a sort of micro view of the fuzzy inference system. If you want to see the entire output surface of your system, that is, the entire span of the output set based on the entire span of the input set, you need to open up the Surface Viewer. This is the last of our five basic GUI tools in the Fuzzy Logic Toolbox, and you open it by selecting **View surface...** from the **View** menu.

The Surface Viewer



Upon opening the Surface Viewer, we are presented with a two-dimensional curve that represents the mapping from service quality to tip amount. Since this is a one-input one-output case, we can see the entire mapping in one plot. Two-input one-output systems also work well, as they generate three-dimensional plots that MATLAB can adeptly manage. When we move beyond three dimensions overall, we start to encounter trouble displaying the results. Accordingly, the Surface Viewer is equipped with pop-up menus that let you select any two inputs and any one output for plotting. Just below the pop-up menus are two text input fields that let you determine how many *x-axis* and *y-axis* grid lines you want to include. This allows you to keep the calculation time reasonable for complex problems. Pushing the **Evaluate** button initiates the calculation, and the plot comes up soon after the calculation is complete. To change the *x-axis* or *y-axis* grid after the surface is

in view, simply change the appropriate text field, and click on either **X-grids:** or **Y-grids:**, according to which text field you changed, to redraw the plot.

The Surface Viewer has a special capability that is very helpful in cases with two (or more) inputs and one output: you can actually grab the axes and reposition them to get a different three-dimensional view on the data. The **Ref. Input:** field is used in situations when there are more inputs required by the system than the surface is mapping. Suppose you have a four-input one-output system and would like to see the output surface. The Surface Viewer can generate a three-dimensional output surface where any two of the inputs vary, but two of the inputs must be held constant since computer monitors cannot display a five-dimensional shape. In such a case the input would be a four-dimensional vector with `NaN`s holding the place of the varying inputs while numerical values would indicate those values that remain fixed. An `NaN` is the IEEE symbol for "not a number."

This concludes the quick walk-through of each of the main GUI tools. Notice that for the tipping problem, the output of the fuzzy system matches our original idea of the shape of the fuzzy mapping from service to tip fairly well. In hindsight, you might say, "Why bother? I could have just drawn a quick lookup table and been done an hour ago!" However, if you are interested in solving an entire class of similar decision-making problems, fuzzy logic may provide an appropriate tool for the solution, given its ease with which a system can be quickly modified.

Importing and Exporting from the GUI Tools

When you save a fuzzy system to disk, you're saving an ASCII text FIS file representation of that system with the file suffix `.fis`. This text file can be edited and modified and is simple to understand. When you save your fuzzy system to the MATLAB workspace, you're creating a variable (whose name you choose) that will act as a MATLAB structure for the FIS system. FIS files and FIS structures represent the same system.

Note: If you do not save your FIS to your disk, but only save it to the MATLAB workspace, you will not be able to recover it for use in a new MATLAB session.

Customizing Your Fuzzy System

If you want to include customized functions as part of your use of the Fuzzy Logic Toolbox, there are a few guidelines you need to follow. The AND method, OR method, aggregation method, and defuzzification method functions you provide need to work in a similar way to `max`, `min`, or `prod` in MATLAB. That is, they must be able to operate down the columns of a matrix. For example, the implication method does an element by element matrix operation, similar to the `min` function, as in

```
a=[1 2; 3 4];
b=[2 2; 2 2];
min(a,b)
ans =
    1     2
    2     2
```

Custom Membership Functions

You can create your own membership functions using an M-file. The values these functions can take must be between 0 and 1. There is a limitation on customized membership functions in that they cannot use more than 16 parameters.

To define a custom membership function named *`custmf`*:

1. Create an M-file for a function, *`custmf.m`*, that takes values between 0 and 1, and depends on at most 16 parameters.

2. Choose the **Add Custom MF** item in the **Edit** menu on the Membership Function Editor GUI.

3. Enter your custom membership function M-file name, *`custmf`*, in the **M-file function name** text box.

4. Enter the vector of parameters you want to use to parameterize your customized membership function in the text box next to **Parameter list**.

5. Give the custom membership function a name different from any other membership function name you will use in your FIS.

6. Select **OK**.

Here is some sample code for a custom membership function, `testmf1`, that depends on eight parameters between 0 and 10.

```
function out = testmf1(x, params)
for i=1:length(x)
if x(i)<params(1)
 y(i)=params(1);
elseif x(i)<params(2)
 y(i)=params(2);
elseif x(i)<params(3)
 y(i)=params(3);
elseif x(i)<params(4)
 y(i)=params(4);
elseif x(i)<params(5)
 y(i)=params(5);
elseif x(i)<params(6)
 y(i)=params(6);
elseif x(i)<params(7)
 y(i)=params(7);
elseif x(i)<params(8)
 y(i)=params(8);
else
 y(i)=0;
end
end
out=.1*y';
```

You can try naming this file `testmf1.m` and loading it into the Membership Function Editor using the parameters of your choice.

Working from the Command Line

The tipping example system is one of many example fuzzy inference systems provided with the Fuzzy Logic Toolbox. The FIS is always cast as a MATLAB structure. To load this system (rather than bothering with creating it from scratch), type

```
a = readfis('tipper.fis')
```

MATLAB will respond with

```
a =
            name: 'tipper'
            type: 'mamdani'
       andMethod: 'min'
        orMethod: 'max'
    defuzzMethod: 'centroid'
       impMethod: 'min'
       aggMethod: 'max'
           input: [1x2 struct]
          output: [1x1 struct]
            rule: [1x3 struct]
```

The labels on the left of this listing represent the various components of the MATLAB structure associated with `tipper.fis`. You can access the various components of this structure by typing the component name after typing `a`. At the MATLAB command line, type

```
a.type
```

for example. MATLAB will respond with

```
ans =
mamdani
```

The function

```
getfis(a)
```

returns almost the same structure information that typing `a`, alone does.

`getfis(a)` returns

```
Name      = tipper
      Type      = mamdani
```

```
NumInputs = 2
InLabels  =
      service
      food
NumOutputs = 1
OutLabels =
      tip
NumRules = 3
AndMethod = min
OrMethod = max
ImpMethod = min
AggMethod = max
DefuzzMethod = centroid
```

Notice that some of these fields are not part of the structure, `a`. Thus, you cannot get information by typing `a.Inlabels`, but you can get it by typing:

```
getfis(a,'Inlabels')
```

Similarly, you can obtain structure information using `getfis` in this manner.

```
getfis(a,'input',1)
getfis(a,'output',1)
getfis(a,'input',1,'mf',1)
```

The *`structure.field`* syntax also generates this information. For more information on the syntax for MATLAB structures, see Chapter 13, "Structures and Cell Arrays," in *Using MATLAB*.

For example, type

```
a.input
```

or

```
a.input(1).mf(1)
```

The function `getfis` is loosely modeled on the Handle Graphics® function `get`. There is also a function called `setfis` that acts as the reciprocal to `getfis`. It allows you to change any property of an FIS. For example, if you wanted to change the name of this system, you could type

```
a = setfis(a,'name','gratuity');
```

However, since a is already a MATLAB structure, you can set this information more simply by typing

```
a.name = 'gratuity';
```

Now the FIS structure a has been changed to reflect the new name. If you want a little more insight into this FIS structure, try

```
showfis(a)
```

This returns a printout listing all the information about a. This function is intended more for debugging than anything else, but it shows all the information recorded in the FIS structure

Since the variable, a, designates the fuzzy tipping system, you can call up any of the GUIs for the tipping system directly from the command line. Any of the following will bring up the tipping system with the associated GUI.

- `fuzzy(a)`: brings up the FIS Editor
- `mfedit(a)`: brings up the Membership Function Editor
- `ruleedit(a)`: brings up the Rule Editor
- `ruleview(a)`: brings up the Rule Viewer
- `surfview(a)`: brings up the Surface Viewer

If, in addition, a is a Sugeno-type FIS, then `anfisedit(a)` will bring up the ANFIS Editor GUI.

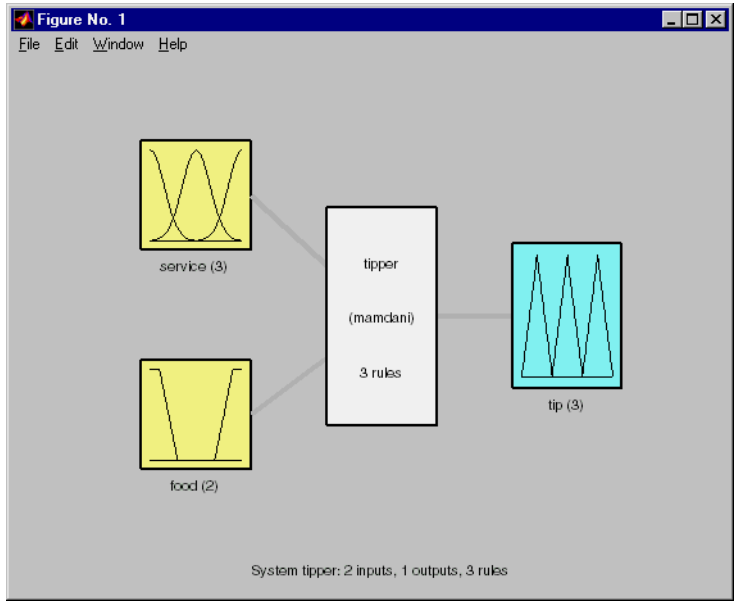
Once any of these GUIs has been opened, you can access any of the other GUIs using the pull-down menu rather than the command line.

System Display Functions

There are three functions designed to give you a high-level view of your fuzzy inference system from the command line: `plotfis`, `plotmf`, and `gensurf`. The

first of these displays the whole system as a block diagram much as it would appear on the FIS Editor.

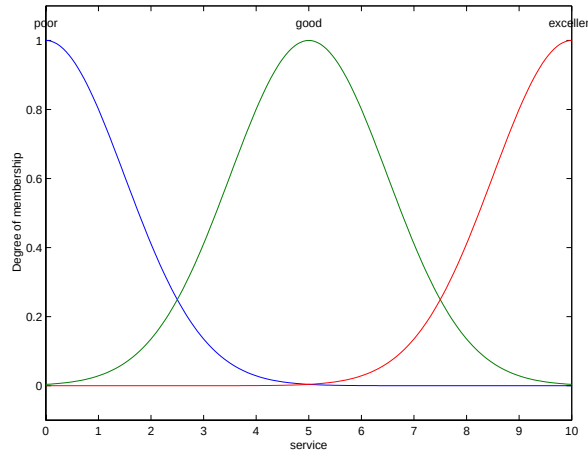
```
plotfis(a)
```



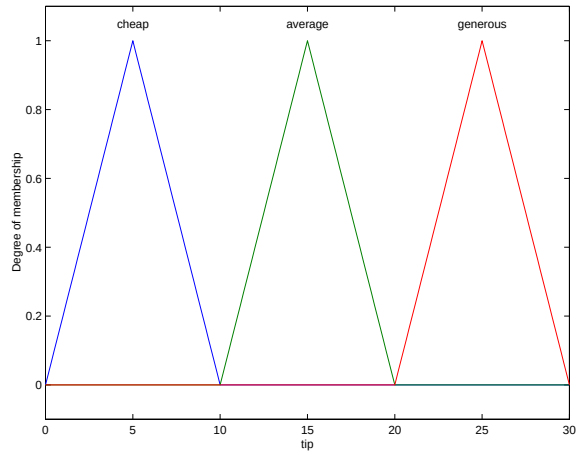
After closing any open MATLAB figures or GUI windows, the function `plotmf` plots all the membership functions associated with a given variable as follows:

```
plotmf(a,'input',1)
```

returns



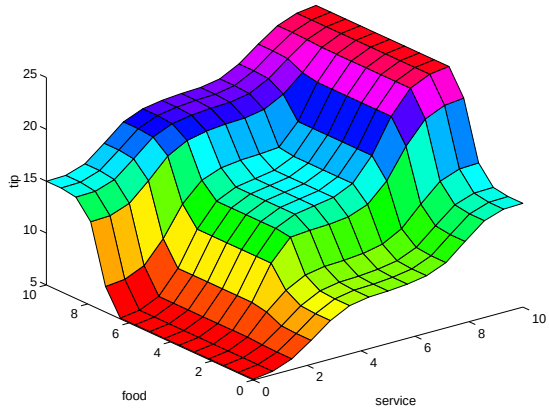

```
plotmf(a,'output',1)
```



These plots will appear in the Membership Function Editor GUI, or in an open MATLAB figure, if `plotmf` is called while either of these is open.

Finally, the function `gensurf` will plot any one or two inputs versus any one output of a given system. The result is either a two-dimensional curve, or a three-dimensional surface. Note that when there are three or more inputs, `gensurf` must be generated with all but two inputs fixed, as is described in the description of `genfis` in Chapter 3, “Reference” .

```
gensurf(a)
```



Building a System from Scratch

It is possible to use the Fuzzy Logic Toolbox without bothering with the GUI tools at all. For instance, to build the tipping system entirely from the command line, you would use the commands `newfis`, `addvar`, `addmf`, and `addrule`.

Probably the trickiest part of this process is learning the shorthand that the fuzzy inference systems use for building rules. This is accomplished using the command line function, `addrule`.

Each variable, input, or output, has an index number, and each membership function has an index number. The rules are built from statements like this

```
if input1 is MF1 or input2 is MF3 then output1 is MF2 (weight = 0.5)
```

This rule is turned into a structure according to the following logic: If there are *m* inputs to a system and *n* outputs, then the first *m* vector entries of the rule structure correspond to inputs 1 through *m*. The entry in column 1 is the index number for the membership function associated with input 1. The entry in

column 2 is the index number for the membership function associated with input 2. And so on. The next *n* columns work the same way for the outputs. Column *m* + *n* + 1 is the weight associated with that rule (typically 1) and column *m* + *n* + 2 specifies the connective used (where AND = 1 and OR = 2). The structure associated with the rule shown above is

```
1 3 2 0.5 2
```

Here is one way you can build the entire tipping system from the command line, using the MATLAB structure syntax.

```
a=newfis('tipper');
a.input(1).name='service';
a.input(1).range=[0 10];
a.input(1).mf(1).name='poor';
a.input(1).mf(1).type='gaussmf';
a.input(1).mf(1).params=[1.5 0];
a.input(1).mf(2).name='good';
a.input(1).mf(2).type='gaussmf';
a.input(1).mf(2).params=[1.5 5];
a.input(1).mf(3).name='excellent';
a.input(1).mf(3).type='gaussmf';
a.input(1).mf(3).params=[1.5 10];
a.input(2).name='food';
a.input(2).range=[0 10];
a.input(2).mf(1).name='rancid';
a.input(2).mf(1).type='trapmf';
a.input(2).mf(1).params=[-2 0 1 3];
a.input(2).mf(2).name='delicious';
a.input(2).mf(2).type='trapmf';
a.input(2).mf(2).params=[7 9 10 12];
a.output(1).name='tip';
a.output(1).range=[0 30];
a.output(1).mf(1).name='cheap'
a.output(1).mf(1).type='trimf';
a.output(1).mf(1).params=[0 5 10];
a.output(1).mf(2).name='average';
a.output(1).mf(2).type='trimf';
a.output(1).mf(2).params=[10 15 20];
a.output(1).mf(3).name='generous';
a.output(1).mf(3).type='trimf';
```

```
a.output(1).mf(3).params=[20 25 30];
a.rule(1).antecedent=[1 1];
a.rule(1).consequent=[1];
a.rule(1).weight=1;
a.rule(1).connection=2;
a.rule(2).antecedent=[2 0];
a.rule(2).consequent=[2];
a.rule(2).weight=1;
a.rule(2).connection=1;
a.rule(3).antecedent=[3 2];
a.rule(3).consequent=[3];
a.rule(3).weight=1;
a.rule(3).connection=2
```

Alternatively, here is how you can build the entire tipping system from the command line using Fuzzy Logic Toolbox commands.

```
a=newfis('tipper');
a=addmf(a,'input',1,'service',[0 10]);
a=addmf(a,'input',1,'poor','gaussmf',[1.5 0]);
a=addmf(a,'input',1,'good','gaussmf',[1.5 5]);
a=addmf(a,'input',1,'excellent','gaussmf',[1.5 10]);
a=addvar(a,'input','food',[0 10]);
a=addmf(a,'input',2,'rancid','trapmf',[-2 0 1 3]);
a=addmf(a,'input',2,'delicious','trapmf',[7 9 10 12]);
a=addvar(a,'output','tip',[0 30]);
a=addmf(a,'output',1,'cheap','trimf',[0 5 10]);
a=addmf(a,'output',1,'average','trimf',[10 15 20]);
a=addmf(a,'output',1,'generous','trimf',[20 25 30]);
ruleList=[ ...
1 1 1 1 2
2 0 2 1 1
3 2 3 1 2 ];
a=addrule(a,ruleList);
```

FIS Evaluation

To evaluate the output of a fuzzy system for a given input, use the function `evalfis`. For example, the following script evaluates `tipper` at the input, [1 2].

```
a = readfis('tipper');
evalfis([1 2], a)
ans =
    5.5586
```

This function can also be used for multiple collections of inputs, since different input vectors are represented in different parts of the input structure. By doing multiple evaluations at once, you get a tremendous boost in speed.

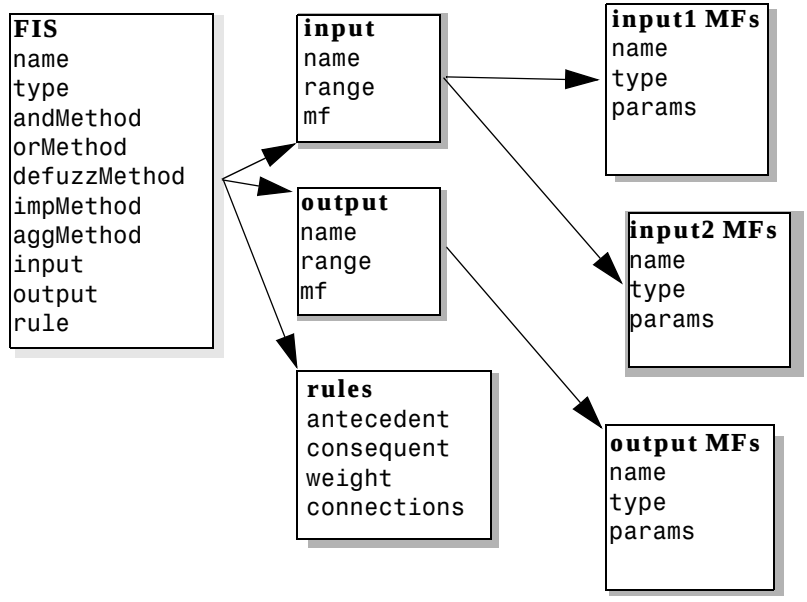
```
evalfis([3 5; 2 7], a)
ans =
    12.2184
     7.7885
```

The FIS Structure

The FIS structure is the MATLAB object that contains all the fuzzy inference system information. This structure is stored inside each GUI tool. Access functions such as `getfis` and `setfis` make it easy to examine this structure. You can also access the FIS structure information using the *`structure.field`* syntax (see the section, “Working from the Command Line” on page 2-65).

All the information for a given fuzzy inference system is contained in the FIS structure, including variable names, membership function definitions, and so

on. This structure can itself be thought of as a hierarchy of structures, as shown in the diagram below:



You can generate a listing of information on the FIS using the `showfis` command, as shown below.

```
showfis(a)
1.  Name             tipper
2.  Type             mamdani
3.  Inputs/Outputs   [ 2 1 ]
4.  NumInputMFs      [ 3 2 ]
5.  NumOutputMFs     3
6.  NumRules         3
7.  AndMethod        min
8.  OrMethod         max
9.  ImpMethod        min
10. AggMethod        max
11. DefuzzMethod     centroid
12. InLabels         service
13.                  food
14. OutLabels        tip
```

```
15. InRange          [ 0 10 ]
16.                  [ 0 10 ]
17. OutRange         [ 0 30 ]
18. InMFLabels       poor
19.                  good
20.                  excellent
21.                  rancid
22.                  delicious
23. OutMFLabels      cheap
24.                  average
25.                  generous
26. InMFTypes        gaussmf
27.                  gaussmf
28.                  gaussmf
29.                  trapmf
30.                  trapmf
31. OutMFTypes       trimf
32.                  trimf
33.                  trimf
34. InMFParams       [ 1.5 0 0 0 ]
35.                  [ 1.5 5 0 0 ]
36.                  [ 1.5 10 0 0 ]
37.                  [ 0 0 1 3 ]
38.                  [ 7 9 10 10 ]
39. OutMFParams      [ 0 5 10 0 ]
40.                  [ 10 15 20 0 ]
41.                  [ 20 25 30 0 ]
42. Rule Antecedent  [ 1 1 ]
43.                  [ 2 0 ]
44.                  [ 3 2 ]
42. Rule Consequent  1
43.                  2
44.                  3
42. Rule Weigth      1
43.                  1
44.                  1
42. Rule Connection  2
43.                  1
44.                  2
```

The list of command line functions associated with FIS construction include `getfis`, `setfis`, `showfis`, `addvar`, `addmf`, `addrule`, `rmvar`, and `rmmf`.

Saving FIS Files on Disk

A specialized text file format is used for saving fuzzy inference systems to a disk. The functions `readfis` and `writefis` are used for reading and writing these files.

If you prefer, you can modify the FIS by editing its `.fis` text file rather than using any of the GUIs. You should be aware, however, that changing one entry may oblige you to change another. For example, if you delete a membership function using this method, you also need to make certain that any rules requiring this membership function are also deleted.

The rules appear in "indexed" format in a `.fis` text file. Here is the file `tipper.fis`.

```
[System]
Name='tipper'
Type='mamdani'
NumInputs=2
NumOutputs=1
NumRules=3
AndMethod='min'
OrMethod='max'
ImpMethod='min'
AggMethod='max'
DefuzzMethod='centroid'

[Input1]
Name='service'
Range=[0 10]
NumMFs=3
MF1='poor':'gaussmf',[1.5 0]
MF2='good':'gaussmf',[1.5 5]
MF3='excellent':'gaussmf',[1.5 10]

[Input2]
Name='food'
Range=[0 10]
NumMFs=2
```

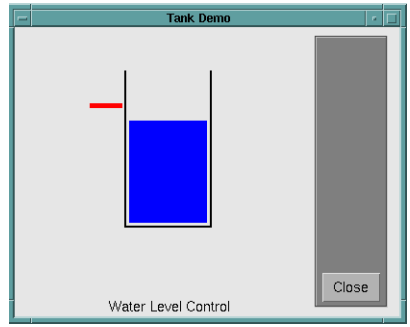
```
MF1='rancid':'trapmf',[0 0 1 3]
MF2='delicious':'trapmf',[7 9 10 10]

[Output1]
Name='tip'
Range=[0 30]
NumMFs=3
MF1='cheap':'trimf',[0 5 10]
MF2='average':'trimf',[10 15 20]
MF3='generous':'trimf',[20 25 30]

[Rules]
1 1, 1 (1) : 2
2 0, 2 (1) : 1
3 2, 3 (1) : 2
```

Working with Simulink

The Fuzzy Logic Toolbox is designed to work seamlessly with Simulink, the simulation software available from The MathWorks. Once you've created your fuzzy system using the GUI tools or some other method, you're ready to embed your system directly into a simulation.

An Example: Water Level Control

Picture a tank with a pipe flowing in and a pipe flowing out. You can change the valve controlling the water that flows in, but the outflow rate depends on the diameter of the outflow pipe (which is constant) and the pressure in the tank (which varies with the water level). The system has some very nonlinear characteristics.



A controller for the water level in the tank needs to know the current water level and it needs to be able to set the valve. Our controller's input will be the water level error (desired water level minus actual water level) and its output will be the rate at which the valve is opening or closing. A first pass at writing a fuzzy controller for this system might be the following.

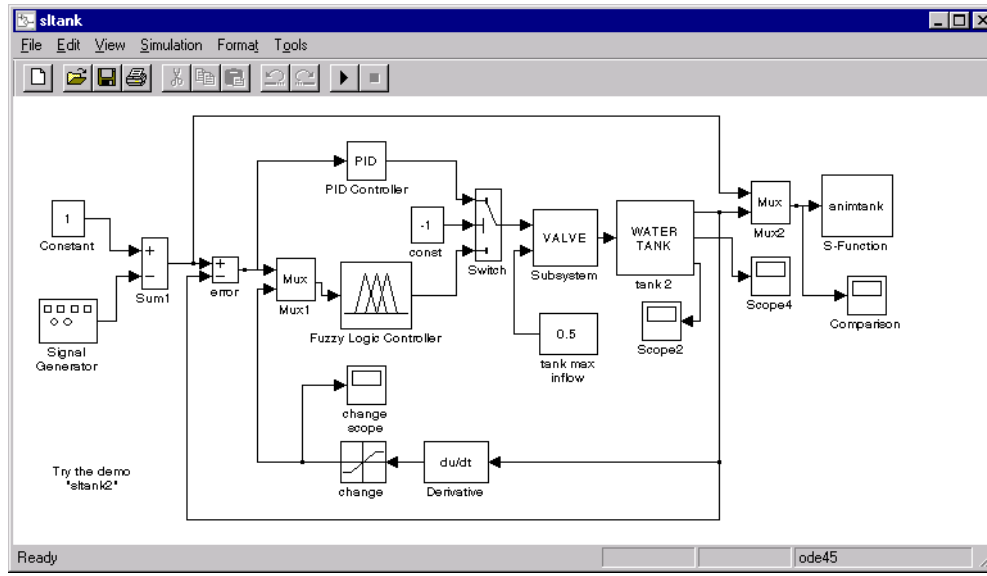
1. *If (level is okay) then (valve is no_change) (1)*
2. *If (level is low) then (valve is open_fast) (1)*
3. *If (level is high) then (valve is close_fast) (1)*

One of the great advantages of the Fuzzy Logic Toolbox is the ability to take fuzzy systems directly into Simulink and test them out in a simulation environment. A Simulink block diagram for this system is shown below. It

contains a Simulink block called the Fuzzy Logic Controller block. The Simulink block diagram for this system is `sltank`. Typing

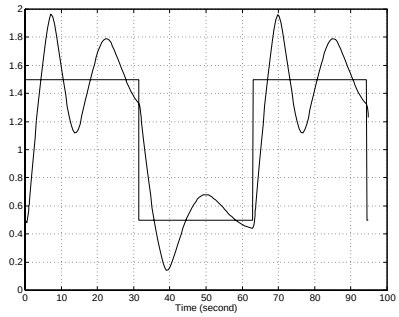
```
sltank
```

at the command line, causes the system to appear.



At the same time, the file `tank.fis` is loaded into the FIS structure `tank`.

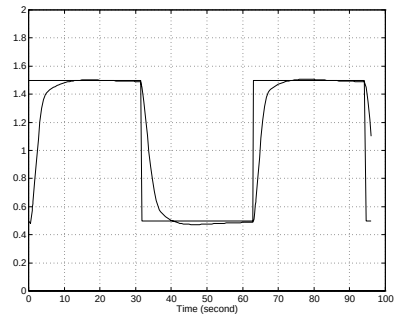
Some experimentation shows that three rules are not sufficient, since the water level tends to oscillate around the desired level. This is seen from the plot below.



We need to add another input, the water level's rate of change, to slow down the valve movement when we get close to the right level.

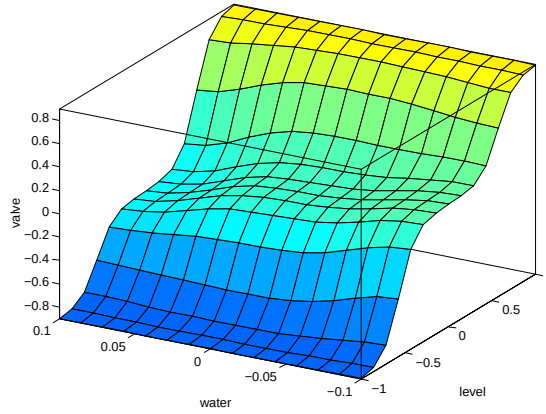
4. *If (level is good) and (rate is negative) then (valve is close_slow) (1)*
5. *If (level is good) and (rate is positive) then (valve is open_slow) (1)*

The demo, `sltank` is built with these five rules. You can examine With all five rules in operations, the step response by simulating this system. This is done by clicking on **Start** from the pull-down menu under **Simulate**, and clicking on the **Comparison** block. The result looks like this



One interesting feature of the water tank system is that the tank empties much more slowly than it fills up because of the specific value of the outflow diameter pipe. We can deal with this by setting the close_slow valve membership function to be slightly different from the open_slow setting. A PID controller does not have this capability. The valve command versus the water level

change rate (depicted as *water*) and the relative water level change (depicted as *level*) surface looks like this. If you look closely, you can see a slight asymmetry to the plot.



Because the MATLAB technical computing environment supports so many tools (like the Control System Toolbox, the Neural Network Toolbox, the Nonlinear Control Design Blockset, and so on), you can, for example, easily make a comparison of a fuzzy controller versus a linear controller or a neural network controller.

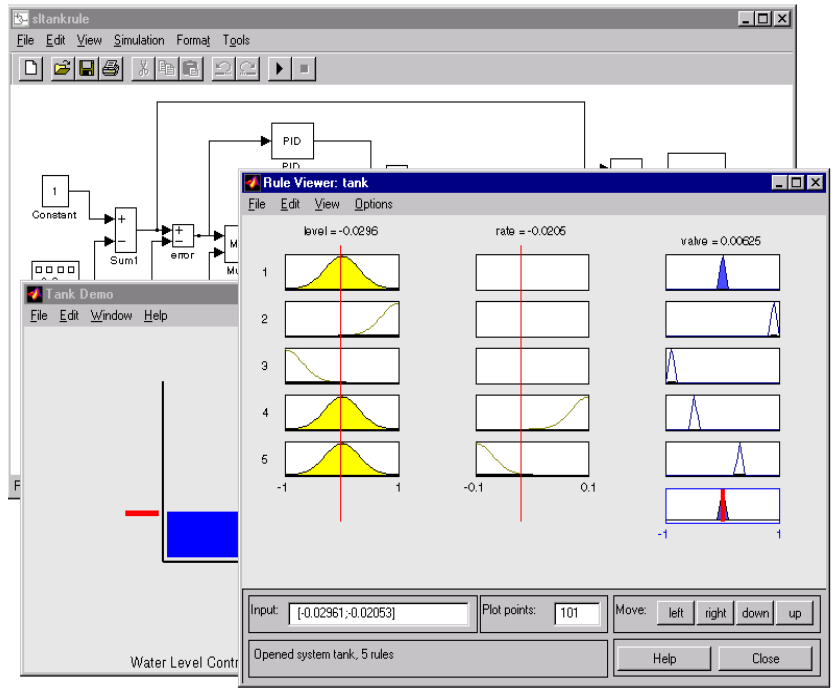
For a demonstration of how the Rule Viewer can be used to interact with a Fuzzy Logic Controller block in a Simulink model, type

```
sltankrule
```

This demo contains a block called the Fuzzy Controller With Rule Viewer block.

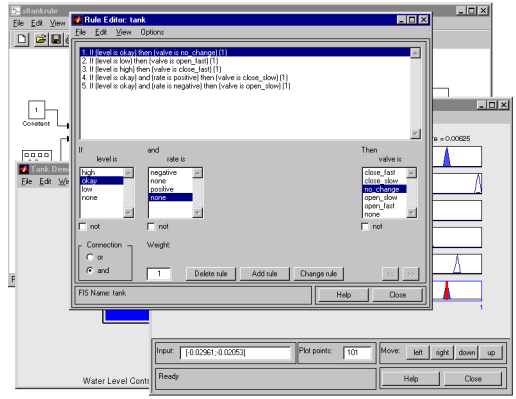
In this demo, the Rule Viewer opens when you start the Simulink simulation. This Rule Viewer provides an animation of how the rules are fired during the

water tank simulation. The windows that open when you simulate the `sltankrule` demo are depicted as follows:



The Rule Viewer that opens during the simulation can be used to access the Membership Function Editor, the Rule Editor, or any of the other GUIs, (see “The Membership Function Editor” on page 2-52, or “The Rule Editor” on page 2-56, for more information).

For example, you may want to open the Rule Editor to change one of your rules. To do so, select the **Edit rules** menu item under the **View** menu of the open Rule Viewer. Now you can view or edit the rules for this Simulink model:



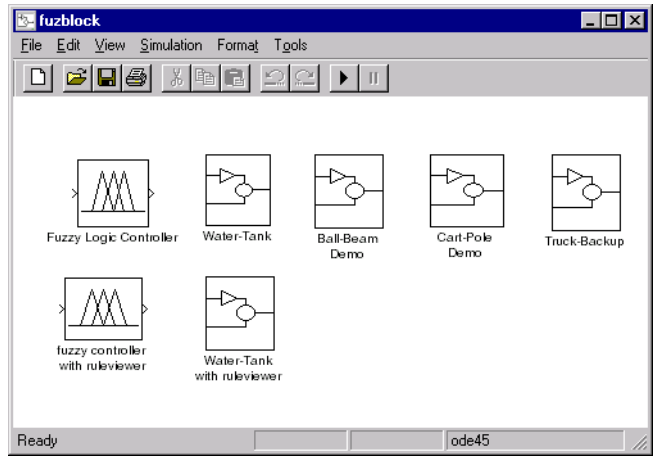
It's best if you stop the simulation prior to selecting any of these editors to change your FIS. Remember to save any changes you make to your FIS to the workspace before you restart the simulation.

Building Your Own Fuzzy Simulink Models

To build your own Simulink systems that use fuzzy logic, simply copy the Fuzzy Logic Controller block out of `sltank` (or any of the other Simulink demo systems available with the toolbox) and place it in your own block diagram. You can also open the Simulink library called `fuzblock`, which contains the Fuzzy Logic Controller block, the Fuzzy Controller With Rule Viewer block, and several demo blocks. To access these blocks, type

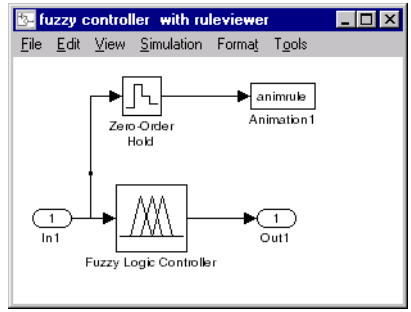
```
fuzblock
```

at the MATLAB prompt. The following library appears:



When you use these blocks, make sure that the fuzzy inference system (FIS) structure corresponding to your fuzzy system is both in the MATLAB workspace, and referred to by name in the dialog box associated with the Fuzzy Logic Controller block.

Double-click on the Fuzzy Controller With Rule Viewer block, and the following appears:



This block uses the zero-order hold method for sampling.

The Fuzzy Logic Controller block is a masked Simulink block based on the S-function `sffis.mex`. This function is itself based on the same algorithms as the function `evalfis`, but it has been tailored to work optimally within the

Simulink environment. For more descriptions of these, see `fuzblock` on page 3-27, and `sffis` on page 3-65.

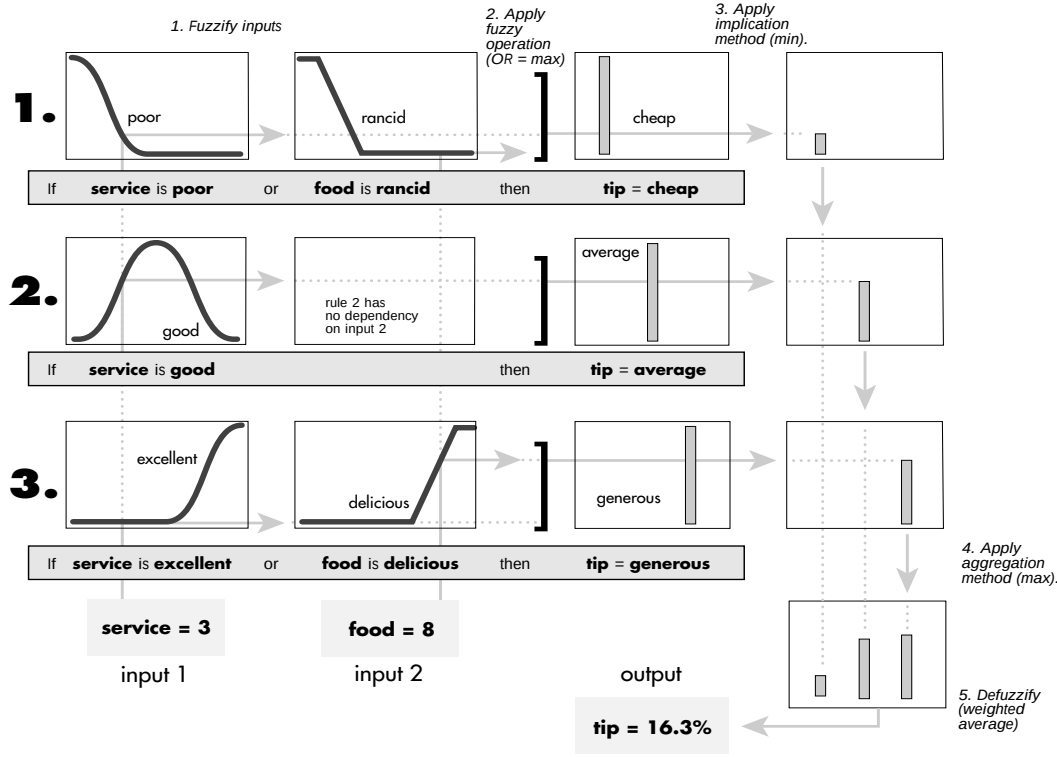
Sugeno-Type Fuzzy Inference

The fuzzy inference process we've been referring to so far is known as Mamdani's fuzzy inference method. It's the most commonly seen fuzzy methodology. In this section we discuss the so-called Sugeno, or Takagi-Sugeno-Kang method of fuzzy inference first introduced in 1985 [Sug85]. It is similar to the Mamdani method in many respects. In fact the first two parts of the fuzzy inference process, fuzzifying the inputs and applying the fuzzy operator, are exactly the same. The main difference between Mamdani-type of fuzzy inference and Sugeno-type is that the output membership functions are only linear or constant for Sugeno-type fuzzy inference.

A typical fuzzy rule in a *zero-order Sugeno fuzzy model* has the form

if x is A and y is B then $z = k$

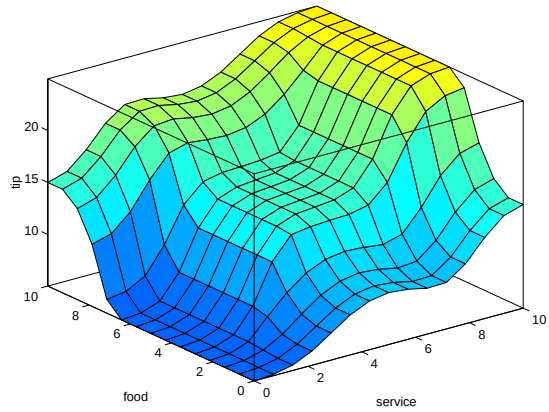
where A and B are fuzzy sets in the antecedent, while k is a crisply defined constant in the consequent. When the output of each rule is a constant like this, the similarity with Mamdani's method is striking. The only distinctions are the fact that all output membership functions are singleton spikes, and the implication and aggregation methods are fixed and can not be edited. The implication method is simply multiplication, and the aggregation operator just includes all of the singletons.



The figure above shows the fuzzy tipping model developed in previous sections of this manual adapted for use as a zero-order Sugeno system. Fortunately it is frequently the case that singleton output functions are completely sufficient for a given problem's needs. As an example, the system `tippersg.fis` is the Sugeno-type representation of the now-familiar tipping model. If you load the

system and plot its output surface, you will see it is almost the same as the Mamdani system we've been looking at.

```
a = readfis('tippersg');
gensurf(a)
```



The more general *first-order Sugeno fuzzy model* has rules of the form

if *x* is *A* and *y* is *B* then *z* = *p*x* + *q*y* + *r*

where *A* and *B* are fuzzy sets in the antecedent, while *p*, *q*, and *r* are all constants. The easiest way to visualize the first-order system is to think of each rule as defining the location of a "moving singleton." That is, the singleton output spikes can move around in a linear fashion in the output space, depending on what the input is. This also tends to make the system notation very compact and efficient. Higher order Sugeno fuzzy models are possible, but they introduce significant complexity with little obvious merit. Sugeno fuzzy models whose output membership functions are greater than first order are not supported by the Fuzzy Logic Toolbox.

Because of the linear dependence of each rule on the system's input variables, the Sugeno method is ideal for acting as an interpolating supervisor of multiple linear controllers that are to be applied, respectively, to different operating conditions of a dynamic nonlinear system. For example, the performance of an aircraft may change dramatically with altitude and Mach number. Linear controllers, though easy to compute and well-suited to any given flight condition, must be updated regularly and smoothly to keep up with the changing state of the flight vehicle. A Sugeno fuzzy inference system is

extremely well suited to the task of smoothly interpolating the linear gains that would be applied across the input space; it's a natural and efficient gain scheduler. Similarly, a Sugeno system is suited for modeling nonlinear systems by interpolating multiple linear models.

An Example: Two Lines

To see a specific example of a system with linear output membership functions, consider the one input one output system stored in `sugeno1.fis`.

```
fismat = readfis('sugeno1');
getfis(fismat,'output',1)
                        Name = output
                        NumMFs = 2
                        MFLabels =
                            line1
                            line2
                        Range = [0 1]
```

The output variable has two membership functions:

```
getfis(fismat,'output',1,'mf',1)
                        Name = line1
                        Type = linear
                        Params =
                            -1    -1
getfis(fismat,'output',1,'mf',2)
                        Name = line2
                        Type = linear
                        Params =
                            1    -1
```

Further, these membership functions are linear functions of the input variable. The membership function `line1` is defined by the equation

$$output = (-1)*input + (-1)$$

and the membership function `line2` is defined by the equation

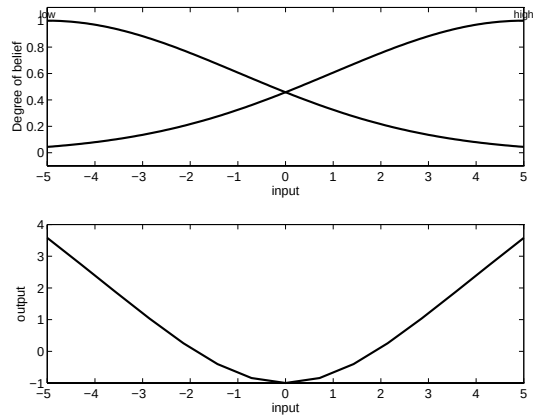
$$output = (1)*input + (-1)$$

The input membership functions and rules define which of these output functions will be expressed and when.

```
showrule(fismat)
ans =
1. If (input is low) then (output is line1) (1)
2. If (input is high) then (output is line2) (1)
```

The function `plotmf` shows us that the membership function `low` generally refers to input values less than zero, while `high` refers to values greater than zero. The function `gensurf` shows how the overall fuzzy system output switches smoothly from the line called `line1` to the line called `line2`.

```
subplot(2,1,1), plotmf(fismat,'input',1)
subplot(2,1,2), gensurf(fismat)
```



This is just one example of how a Sugeno-type system gives you the freedom to incorporate linear systems into your fuzzy systems. By extension, you could build a fuzzy system that switches between several optimal linear controllers as a highly nonlinear system moves around in its operating space.

Conclusion

Because it is a more compact and computationally efficient representation than a Mamdani system, the Sugeno system lends itself to the use of adaptive techniques for constructing fuzzy models. These adaptive techniques can be used to customize the membership functions so that the fuzzy system best models the data.

Note on FIS Conversion: The MATLAB command line function `mam2sug` can be used to convert a Mamdani system into a Sugeno system (not necessarily with a single output) with constant output membership functions. It uses the centroid associated with all of the output membership functions of the Mamdani system. See Chapter 3 for details.

Here are some final considerations about the two different methods:

Advantages of the Sugeno method

- It's computationally efficient.
- It works well with linear techniques (e.g., PID control).
- It works well with optimization and adaptive techniques.
- It has guaranteed continuity of the output surface.
- It's well-suited to mathematical analysis.

Advantages of the Mamdani method

- It's intuitive.
- It has widespread acceptance.
- It's well-suited to human input.

anfis and the ANFIS Editor GUI

The basic structure of the type of fuzzy inference system that we've seen thus far is a model that maps input characteristics to input membership functions, input membership function to rules, rules to a set of output characteristics, output characteristics to output membership functions, and the output membership function to a single-valued output or a decision associated with the output. We have only considered membership functions that have been fixed, and somewhat arbitrarily chosen. Also, we've only applied fuzzy inference to modeling systems whose rule structure is essentially predetermined by the user's interpretation of the characteristics of the variables in the model.

In this section we discuss the use of the function `anfis` and the ANFIS Editor GUI in the Fuzzy Logic Toolbox. These tools apply fuzzy inference techniques to data modeling. As you have seen from the other fuzzy inference GUIs, the shape of the membership functions depends on parameters, and changing these parameters will change the shape of the membership function. Instead of just looking at the data to choose the membership function parameters, we will see how membership function parameters can be chosen automatically using these Fuzzy Logic Toolbox applications.

A Modeling Scenario

Suppose you want to apply fuzzy inference to a system for which you already have a collection of input/output data that you would like to use for modeling, model-following, or some similar scenario. You don't necessarily have a predetermined model structure based on characteristics of variables in your system.

There will be some modeling situations in which you can't just look at the data and discern what the membership functions should look like. Rather than choosing the parameters associated with a given membership function arbitrarily, these parameters could be chosen so as to tailor the membership functions to the input/output data in order to account for these types of variations in the data values. This is where the so-called *neuro-adaptive* learning techniques incorporated into `anfis` in the Fuzzy Logic Toolbox can help.

Model Learning and Inference Through ANFIS

The basic idea behind these neuro-adaptive learning techniques is very simple. These techniques provide a method for the fuzzy modeling procedure to *learn* information about a data set, in order to compute the membership function parameters that best allow the associated fuzzy inference system to track the given input/output data. This learning method works similarly to that of neural networks. The Fuzzy Logic Toolbox function that accomplishes this membership function parameter adjustment is called `anfis`. `anfis` can be accessed either from the command line, or through the ANFIS Editor GUI. Since the functionality of the command line function `anfis` and the ANFIS Editor GUI is similar, they are used somewhat interchangeably in this discussion, until we distinguish them through the description of the GUI.

What Is ANFIS?

The acronym ANFIS derives its name from *adaptive neuro-fuzzy inference system*. Using a given input/output data set, the toolbox function `anfis` constructs a fuzzy inference system (FIS) whose membership function parameters are tuned (adjusted) using either a backpropagation algorithm alone, or in combination with a least squares type of method. This allows your fuzzy systems to learn from the data they are modeling.

FIS Structure and Parameter Adjustment

A network-type structure similar to that of a neural network, which maps inputs through input membership functions and associated parameters, and then through output membership functions and associated parameters to outputs, can be used to interpret the input/output map.

The parameters associated with the membership functions will change through the learning process. The computation of these parameters (or their adjustment) is facilitated by a gradient vector, which provides a measure of how well the fuzzy inference system is modeling the input/output data for a given set of parameters. Once the gradient vector is obtained, any of several optimization routines could be applied in order to adjust the parameters so as to reduce some error measure (usually defined by the sum of the squared difference between actual and desired outputs). `anfis` uses either back propagation or a combination of least squares estimation and backpropagation for membership function parameter estimation.

Familiarity Breeds Validation: Know Your Data

The modeling approach used by `anfis` is similar to many system identification techniques. First, you hypothesize a parameterized model structure (relating inputs to membership functions to rules to outputs to membership functions, and so on). Next, you collect input/output data in a form that will be usable by `anfis` for training. You can then use `anfis` to *train* the FIS model to emulate the training data presented to it by modifying the membership function parameters according to a chosen error criterion.

In general, this type of modeling works well if the training data presented to `anfis` for training (estimating) membership function parameters is fully representative of the features of the data that the trained FIS is intended to model. This is not always the case, however. In some cases, data is collected using noisy measurements, and the training data cannot be representative of all the features of the data that will be presented to the model. This is where *model validation* comes into play.

Model Validation Using Checking and Testing Data Sets

Model validation is the process by which the input vectors from input/output data sets on which the FIS was not trained, are presented to the trained FIS model, to see how well the FIS model predicts the corresponding data set output values. This is accomplished with the ANFIS Editor GUI using the so-called *testing data set*, and its use is described in a subsection that follows. You can also use another type of data set for model validation in `anfis`. This other type of validation data set is referred to as the *checking data set* and this set is used to control the potential for the model overfitting the data. When checking data is presented to `anfis` as well as training data, the FIS model is selected to have parameters associated with the minimum checking data model error.

One problem with model validation for models constructed using adaptive techniques is selecting a data set that is both representative of the data the trained model is intended to emulate, yet sufficiently distinct from the training data set so as not to render the validation process trivial. If you have collected a large amount of data, hopefully this data contains all the necessary representative features, so the process of selecting a data set for checking or testing purposes is made easier. However, if you expect to be presenting noisy measurements to your model, it's possible the training data set does not include all of the representative features you want to model.

The basic idea behind using a checking data set for model validation is that after a certain point in the training, the model begins overfitting the training data set. In principle, the model error for the checking data set tends to decrease as the training takes place up to the point that overfitting begins, and then the model error for the checking data suddenly increases. In the first example in the following section, two similar data sets are used for checking and training, but the checking data set is corrupted by a small amount of noise. This example illustrates of the use of the ANFIS Editor GUI with checking data to reduce the effect of model overfitting. In the second example, a training data set that is presented to `anfis` is sufficiently different than the applied checking data set. By examining the checking error sequence over the training period, it is clear that the checking data set is not good for model validation purposes. This example illustrates the use of the ANFIS Editor GUI to compare data sets.

Some Constraints of anfis

`anfis` is much more complex than the fuzzy inference systems discussed so far, and is not available for all of the fuzzy inference system options. Specifically, `anfis` only supports Sugeno-type systems, and these must be:

- First or zeroth order Sugeno-type systems
- Single output, obtained using weighted average defuzzification (linear or constant output membership functions)
- Of unity weight for each rule

An error occurs if your FIS structure does not comply with these constraints.

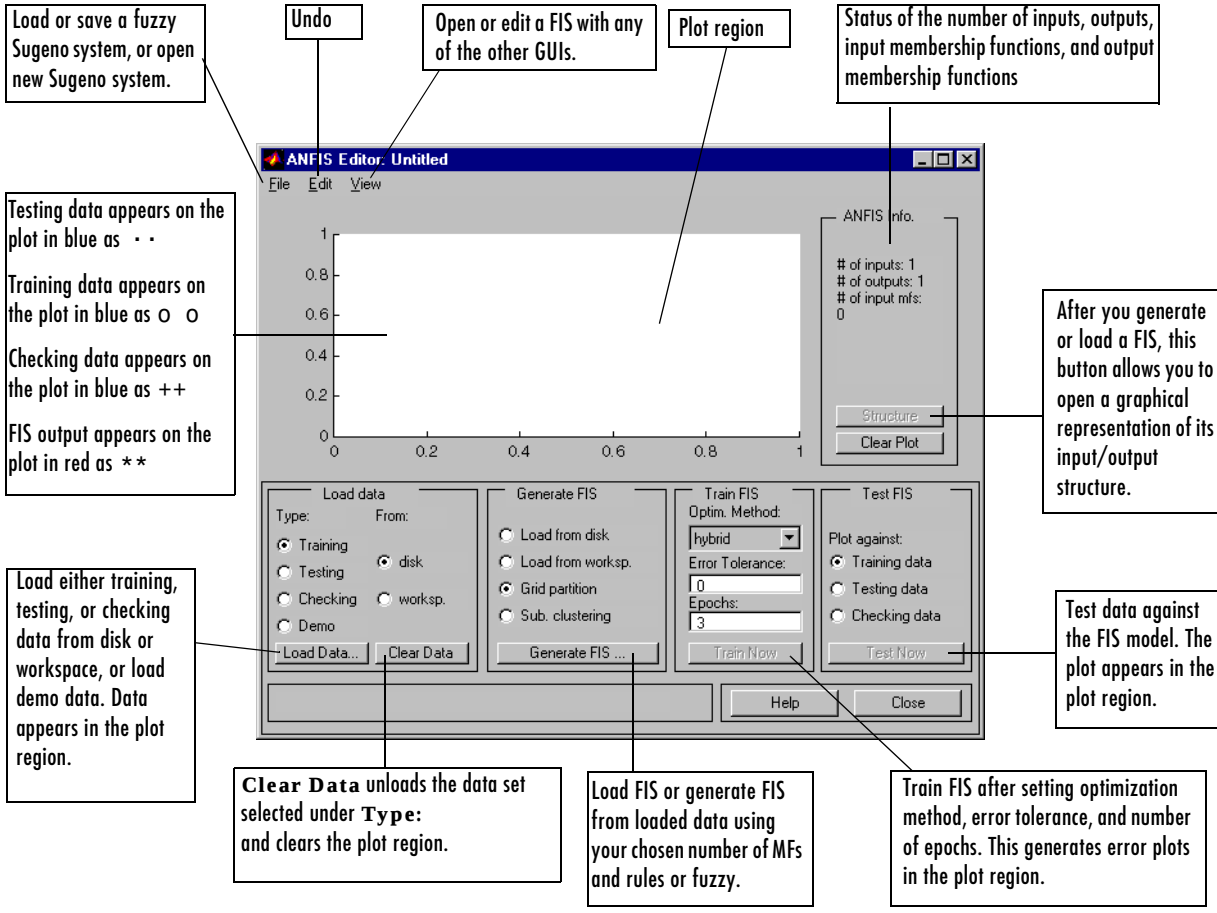
Moreover, `anfis` cannot accept all the customization options that basic fuzzy inference allows. That is, you cannot make your own membership functions and defuzzification functions; you'll have to use the ones provided.

The ANFIS Editor GUI

To get started with the ANFIS Editor GUI, type

```
anfisedit
```

The following GUI will appear on your screen.



From this GUI you can

- Load data (training, testing, and checking) by selecting appropriate radio buttons in the **Load data** portion of the GUI and then selecting **Load Data...**
  The loaded data is plotted on the plot region.
- Generate an initial FIS model or load an initial FIS model using the options in the **Generate FIS** portion of the GUI
- View the FIS model structure once an initial FIS has been generated or loaded by selecting the **Structure** button

- Choose the FIS model parameter optimization method: backpropagation or a mixture of backpropagation and least squares (hybrid method)
- Choose the number of training epochs and the training error tolerance
- Train the FIS model by selecting the **Train** Now button

  This training adjusts the membership function parameters and plots the training (and/or checking data) error plot(s) in the plot region.
- View the FIS model output versus the training, checking, or testing data output by selecting the **Test Now** button

  This function plots the test data against the FIS output in the plot region.

You can also use the ANFIS Editor GUI menu bar to load an FIS training initialization, save your trained FIS, open a new Sugeno system, or open any of the other GUIs to interpret the trained FIS model.

Data Formalities and the ANFIS Editor GUI: Checking and Training

To start training an FIS using either `anfis` or the ANFIS Editor GUI, first you need to have a training data set that contains desired input/output data pairs of the target system to be modeled. Sometimes you also want to have the optional testing data set that can check the generalization capability of the resulting fuzzy inference system, and/or a checking data set that helps with model overfitting during the training. The use of a testing data set and a checking data set for model validation is discussed in “Model Validation Using Checking and Testing Data Sets” on page 2-94. As we mentioned previously, overfitting is accounted for by testing the FIS trained on the training data against the checking data, and choosing the membership function parameters to be those associated with the minimum checking error if these errors indicate model overfitting. You will have to examine your training error plots fairly closely in order to determine this. These issues are discussed later in an example. Usually these training and checking data sets are collected based on observations of the target system and are then stored in separate files.

Note on Data Format: Any data set you load into the ANFIS Editor GUI, (or that is applied to the command line function `anfis`) must be a matrix with the input data arranged as vectors in all but the last column. The output data must be in the last column.

ANFIS Editor GUI Example 1: Checking Data Helps Model Validation

In this section we look at an example that loads similar training and checking data sets, only the checking data set is corrupted by noise.

Loading Data

To work both of the following examples, you load the training data sets (`fuzex1trnData` and `fuzex2trnData`) and the checking data sets (`fuzex1chkData` and `fuzex2chkData`), into the ANFIS Editor GUI from the workspace. You may also substitute your own data sets.

To load these data sets from the directory fuzzydemos into the MATLAB workspace, type

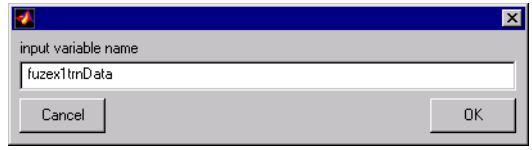
```
load fuzex1trnData.dat
load fuzex2trnData.dat
load fuzex1chkData.dat
load fuzex2chkData.dat
```

from the command line.

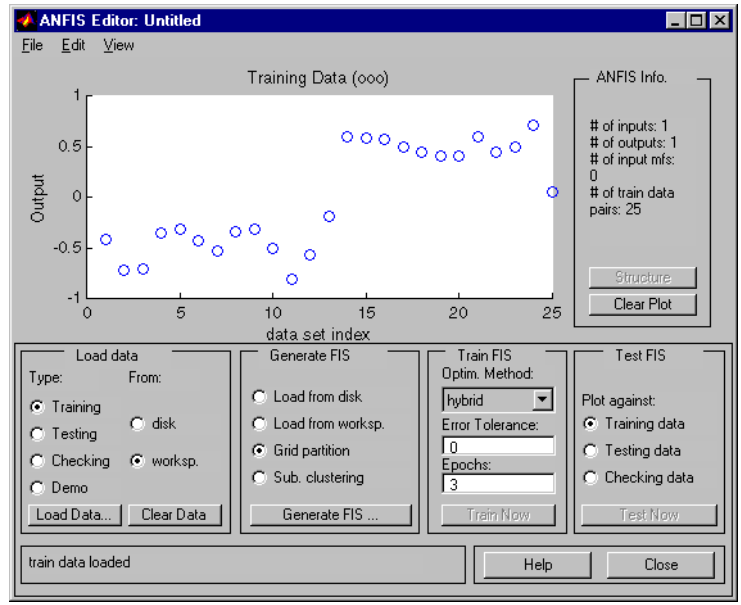
Note on loading data: You may also want to load your data set from the `fuzzydemos` or any other directory on the disk, using the ANFIS Editor GUI, directly.

Open the ANFIS Editor GUI by typing `anfisedit`. To load the training data set, click on **Training, worksp.** and then **Load Data...**.

The small GUI window that pops up allows you to type in a variable name from the workspace. Type in `fuzex1trnData`, as shown below.

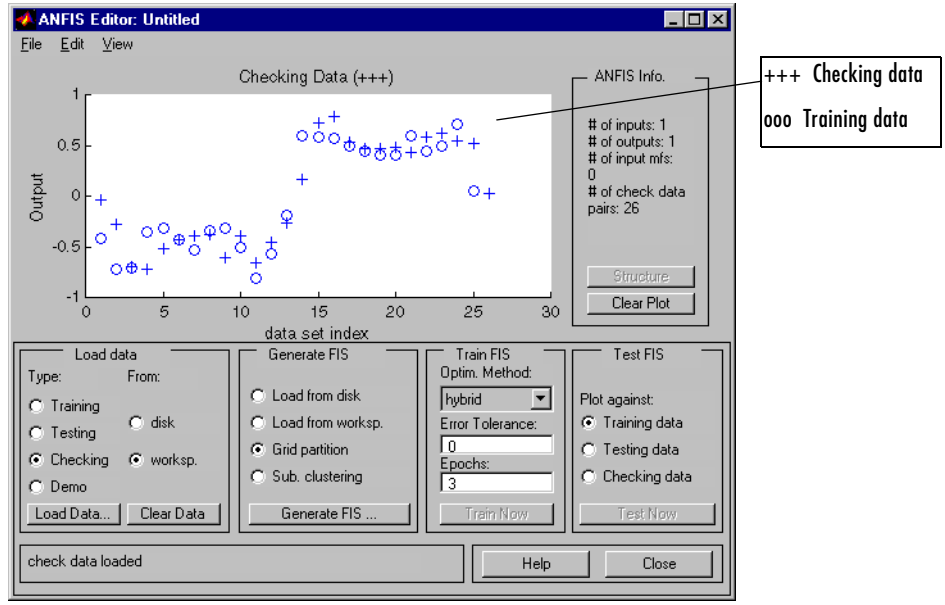


The training data appears in the plot in the center of the GUI as a set of *circles*.



Notice the horizontal axis is marked **data set index**. This index indicates the row from which that input data value was obtained (whether or not the input is a vector or a scalar). Next click on **Checking** in the **Type** column of the **Load**

data portion of the GUI to load `fuzex1chkData` from the workspace. This data appears in the GUI plot as *plusses* superimposed on the training data.



This data set will be used to train a fuzzy system by adjusting the membership function parameters that best model this data. The next step is to specify an initial fuzzy inference system for `anfis` to train.

Initializing and Generating Your FIS

You can either initialize the FIS parameters to your own preference, or if you do not have any preference for how you want the initial membership functions to be parameterized, you can let `anfis` do this for you.

Automatic FIS Structure Generation with ANFIS

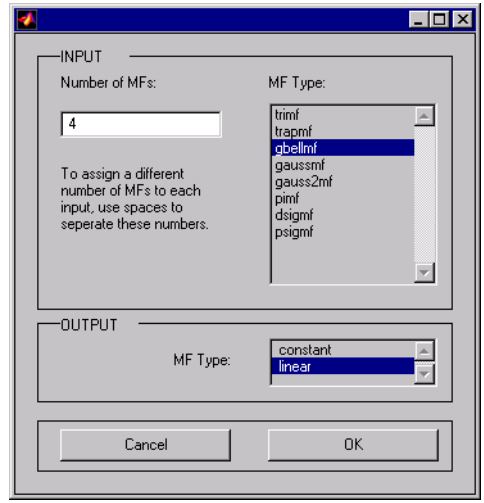
To initialize your FIS using `anfis`:

1. Choose **Grid partition,** the default partitioning method. (The two partition methods, grid partitioning and subtractive clustering, are described later in

"Fuzzy C-Means Clustering" on page 2-120, and in "Subtractive Clustering" on page 2-123.

2 Click on the **Generate FIS** button. This brings up a menu from which you can choose the number of membership functions, **MFs**, and the type of input and output membership functions. Notice there are only two choices for the output membership function: **constant** and **linear**. This limitation of output membership function choices is because `anfis` only operates on Sugeno-type systems.

3 Fill in the entries as we've done below, and click on **OK.**



You can also implement this FIS generation from the command line using the command `genfis1` (for grid partitioning) or `genfis2` (for subtractive clustering). A command line language example illustrating the use of `genfis1` and `anfis` is provided later.

Specifying Your Own Membership Functions for ANFIS

Although we don't expect you to do this for this example, you can choose your own preferred membership functions with specific parameters to be used by `anfis` as an initial FIS for training.

To define your own FIS structure and parameters:

1. Open the **Edit membership functions** menu item from the **View** menu.

2. Add your desired membership functions (the custom membership option will be disabled for `anfis`). The output membership functions must either be all constant or all linear. For carrying out this and the following step, see “The FIS Editor” on page 2-49 and “The Membership Function Editor” on page 2-52.

3. Select the **Edit rules** menu item in the **View** menu. Use the Rule Editor to generate the rules (see“The Rule Editor” on page 2-56).

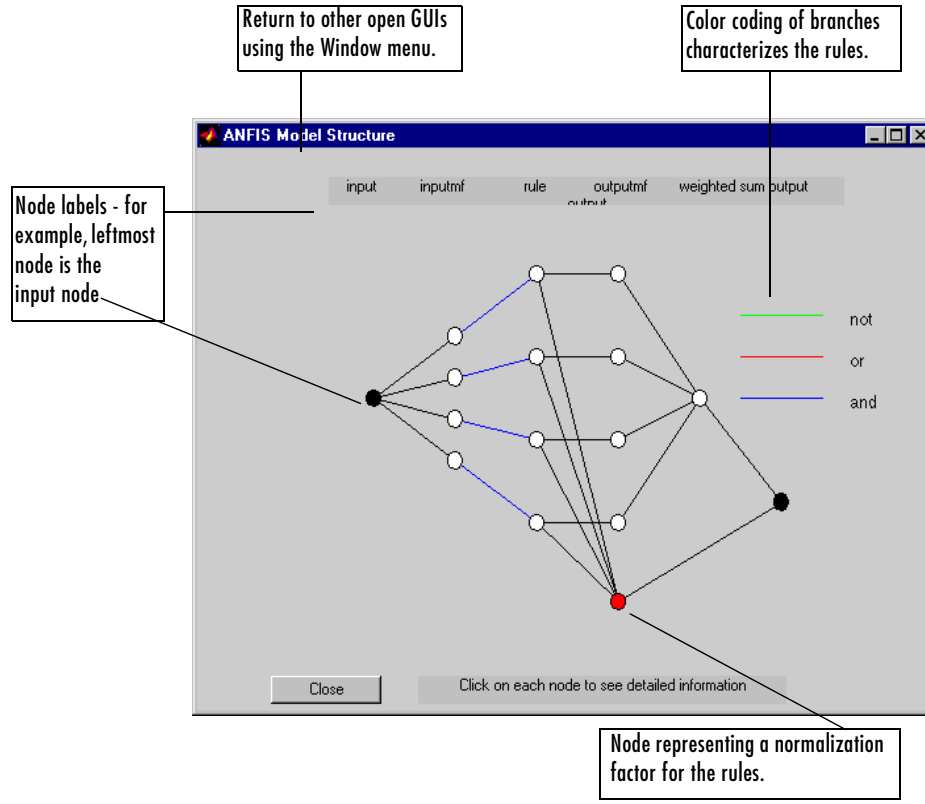
4. Select the **Edit FIS properties** menu item from the **View** menu. Name your FIS, and save it to either the workspace or the disk.

5. Use the **View** menu to return to the ANFIS Editor GUI to train the FIS.

To load an existing FIS for ANFIS initialization, in the **Generate FIS** portion of the GUI, click on **Load from worksp.** or **Load from disk**. You will load your FIS from the disk if you have saved an FIS previously that you would like to use. Otherwise you will be loading your FIS from the workspace. Either of these radio buttons toggle the **Generate FIS** button to **Load...**. Load your FIS by clicking on this button.

Viewing Your FIS Structure

After you generate the FIS, you can view the model structure by clicking on the **Structure** button in the middle of the right-hand side of the GUI. A new GUI appears, as follows:



The branches in this nodal graph are color coded to indicate whether or not *and*, *not*, or *or*, are used in the rules. Clicking on the nodes indicates information about the structure.

You can view the membership functions or the rules by opening either the Membership Function Editor, or the Rule Editor from the **View** menu.

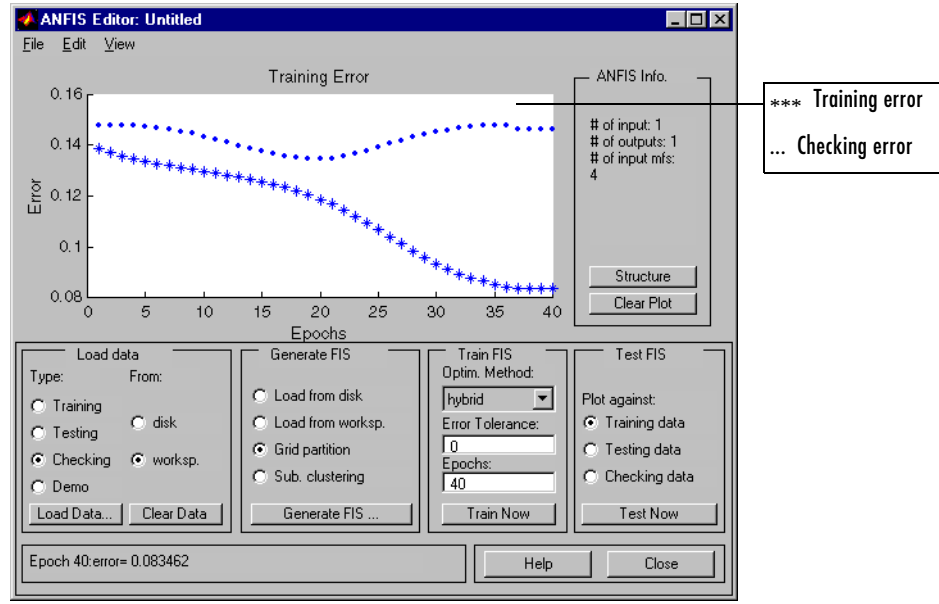
ANFIS Training

The two `anfis` parameter optimization method options available for FIS training are **hybrid** (the default, mixed least squares and backpropagation) and **backpropa** (backpropagation). The **Error Tolerance** is used to create a training stopping criterion, which is related to the error size. The training will stop after the training data error remains within this tolerance. This is best left set to 0 if you don't know how your training error is going to behave.

To start the training:

- Leave the optimization method at **hybrid**.
- Set the number of training epochs to 40, under the **Epochs** listing on the GUI (the default value is 3).
- Select **Train Now**.

The following should appear on your screen:

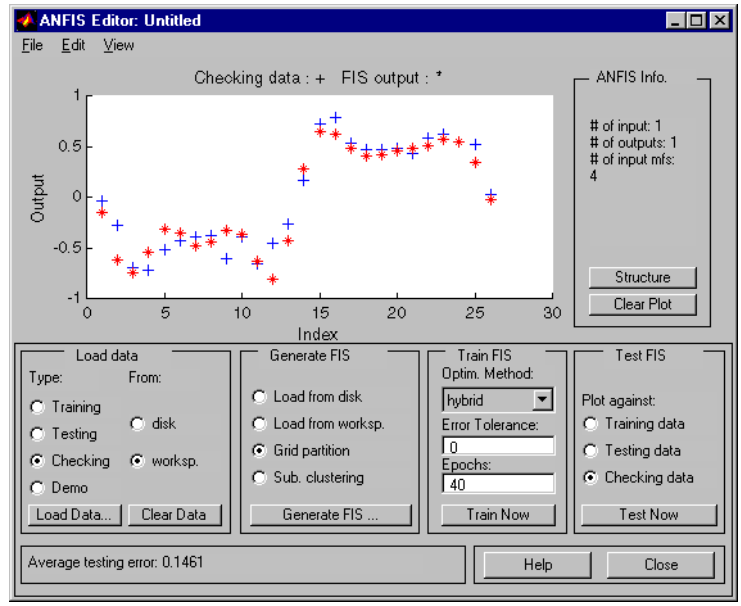


Notice how the checking error decreases up to a certain point in the training and then it increases. This increase represents the point of model overfitting. `anfis` chooses the model parameters associated with the minimum checking

error (just prior to this jump point). This is an example for which the checking data option of `anfis` is useful.

Testing Your Data Against the Trained FIS

To test your FIS against the checking data, click on **Checking data** in the **Test FIS** portion of the GUI, and click on **Test Now**. Now when you test the checking data against the FIS it looks pretty good:



Note on loading more data with anfis: If you are ever loading data into `anfis` after clearing previously loaded data, you must make sure that the newly loaded data sets have the same number of inputs as the previously loaded ones did. Otherwise you will have to start a new `anfisedit` session from the command line.

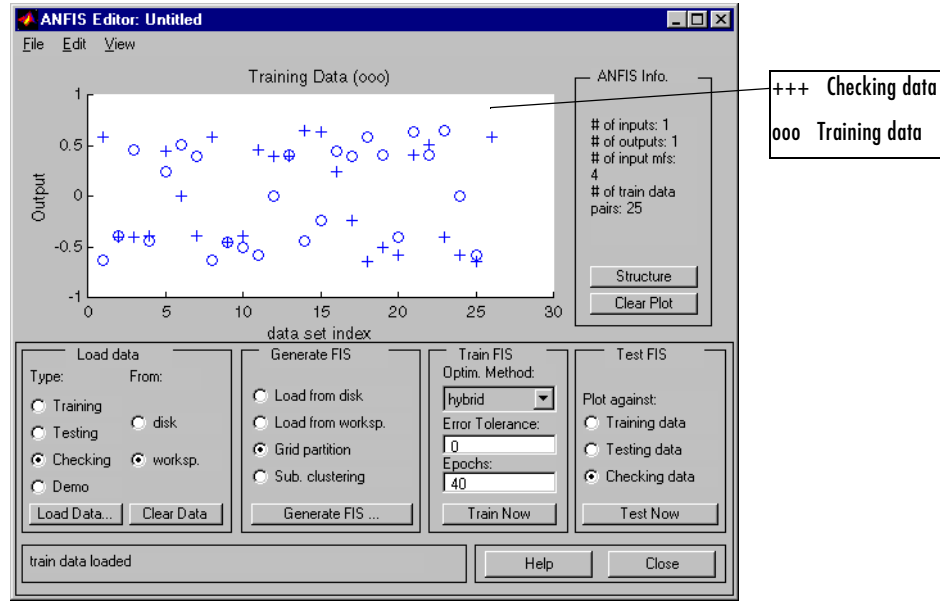
Note on the Checking Data option and Clearing Data: If you don't want to use the checking data option of `anfis`, don't load any checking data before you train the FIS. If you decide to retrain your FIS with no checking data, you can unload the checking data in one of two ways. One method is to click on the **Checking** radio button in the **Load data** portion of the GUI and then click on **Clear Data** to unload the checking data. The other method you can use is to close the GUI and go to the command line and retype `anfisedit`. In this case you will have to reload the training data. After clearing the data, you will need to regenerate your FIS. Once the FIS is generated you can use your first training experience to decide on the number of training epochs you want for the second round of training.

ANFIS Editor GUI Example 2: Checking Data Doesn't Validate Model

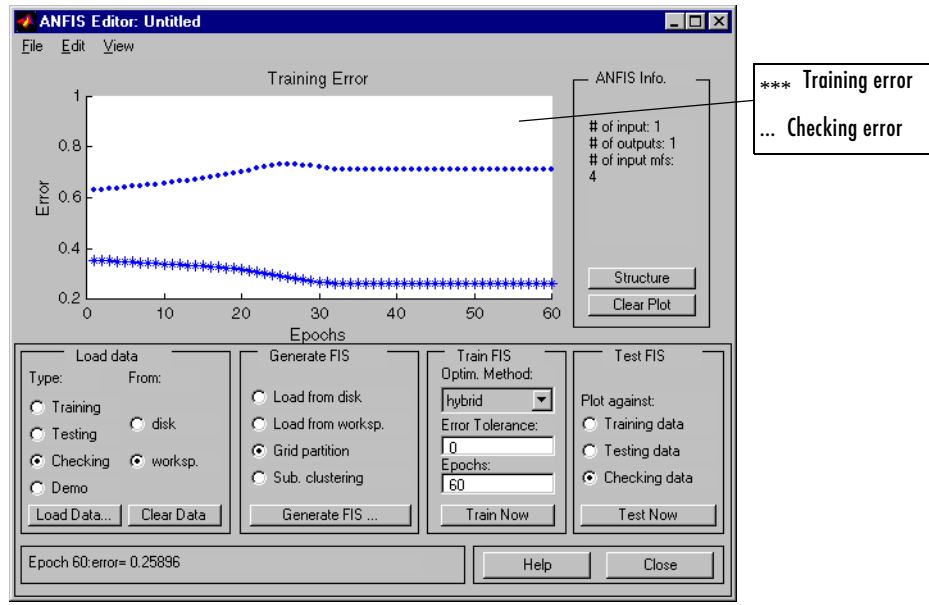
In this example, we examine what happens when the training and checking data sets are sufficiently different. We see how the ANFIS Editor GUI can be used to learn something about data sets and how they differ.

1. Clear both the training and checking data.
2. You can press the **Clear Plot** button on the right, although you don't have to.
3. Load `fuzex2trnData` and `fuzex2chkData` (respectively, the training data and checking data) from the MATLAB workspace just as you did in the previous example.

You should get something that looks like this:



Train the FIS for this system exactly as you did in the previous example, except now choose 60 **Epochs** before training. You should get the following:

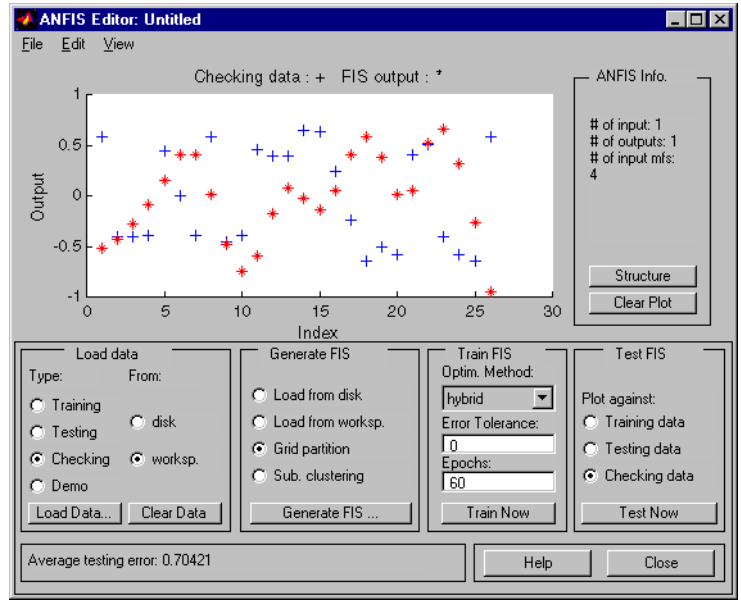


Notice the checking error is quite large. It appears that the minimum checking error occurs within the first epoch. Recall that using the checking data option with `anfis` automatically sets the FIS parameters to be those associated with the minimum checking error. Clearly this set of membership functions would not be the best choice for modeling the training data.

What's wrong here? This example illustrates the problem discussed earlier wherein the checking data set presented to `anfis` for training was sufficiently different from the training data set. As a result, the trained FIS did not capture the features of this data set very well. This illustrates the importance of knowing the features of your data set well enough when you select your training and checking data. When this is not the case, you can analyze the checking error plots to see whether or not the checking data performed sufficiently well with the trained model. In this example, the checking error is sufficiently large to indicate that either more data needs to be selected for training, or you may want to modify your membership function choices (both the number of membership functions and the type). Otherwise the system can

be retrained without the checking data, if you think the training data captures sufficiently the features you are trying to represent.

To complete this example, let's test the trained FIS model against the checking data. To do so, click on **Checking data** in the **Test FIS** portion of the GUI, and click on **Test Now**. The following plot in the GUI indicates that there is quite a discrepancy between the checking data output and the FIS output.



anfis from the Command Line

As you can see, generating an FIS using the ANFIS Editor GUI is quite simple. However, as you saw in the last example, you need to be cautious about implementing the checking data validation feature of `anfis`. You must check that the checking data error does what is supposed to. Otherwise you need to retrain the FIS.

In this section we describe how to carry out the command line features of `anfis` on a chaotic times-series prediction example.

Using anfis for Chaotic Time Series Prediction

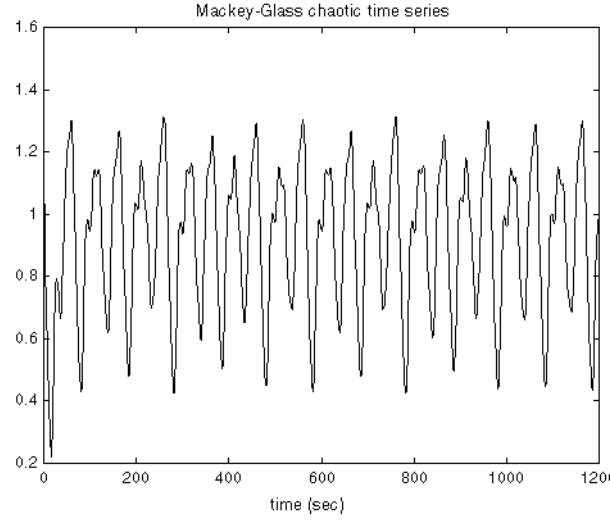
The demo `mgtsdemo` uses `anfis` to predict a time series that is generated by the following Mackey-Glass (MG) time-delay differential equation:

$$\dot{x}(t) = \frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)} - 0.1x(t)$$

This time series is chaotic, and so there is no clearly defined period. The series will not converge or diverge, and the trajectory is highly sensitive to initial conditions. This is a benchmark problem in the neural network and fuzzy modeling research communities.

To obtain the time series value at integer points, we applied the fourth-order Runge-Kutta method to find the numerical solution to the above MG equation; the result was saved in the file `mgdata.dat`. Here we assume $x(0) = 1.2$, $\tau = 17$, and $x(t) = 0$ for $t < 0$. To plot the MG time series, type

```
load mgdata.dat
t = mgdata(:, 1); x = mgdata(:, 2); plot(t, x);
```



In time-series prediction we want to use known values of the time series up to the point in time, say, t, to predict the value at some point in the future, say, $t+P$. The standard method for this type of prediction is to create a mapping

from D sample data points, sampled every Δ units in time, ($x(t-(D-1)\Delta)$,..., $x(t-\Delta)$, x(t)), to a predicted future value x(t+P). Following the conventional settings for predicting the MG time series, we set $D = 4$ and $\Delta = P = 6$. For each `t`, the input training data for `anfis` is a four dimensional vector of the following form:

```
w(t) = [x(t–18) x(t–12) x(t–6) x(t)]
```

The output training data corresponds to the trajectory prediction:

```
s(t) = x(t+6)
```

For each *t*, ranging in values from 118 to 1117, the training input/output data will be a structure whose first component is the four-dimensional input *w*, and whose second component is the output `s`. There will be 1000 input/output data values. We use the first 500 data values for the `anfis` training (these become the training data set), while the others are used as checking data for validating the identified fuzzy model. This results in two 500-point data structures: `trnData` and `chkData`.

Here is the code that generates this data:

```
for t=118:1117,
Data(t-117,:)=[x(t-18) x(t-12) x(t-6) x(t) x(t+6)];
end

trnData=Data(1:500, :);
chkData=Data(501:end, :);
```

To start the training, we need an FIS structure that specifies the structure and initial parameters of the FIS for learning. This is the task of `genfis1`:

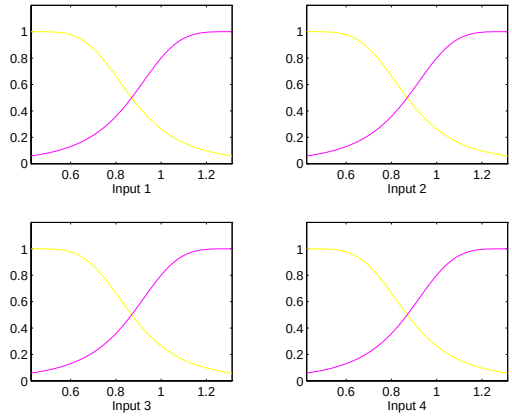
```
fismat = genfis1(trnData);
```

Since we did not specify numbers and types of membership functions used in the FIS, default values are assumed. These defaults provide two generalized bell membership functions on each of the four inputs, eight altogether. The generated FIS structure contains 16 fuzzy rules with 104 parameters. In order to achieve good generalization capability, it is important to have the number of training data points be several times larger than the number parameters being estimated. In this case, the ratio between data and parameters is about five (500/104).

The function genfis1 generates initial membership functions that are equally spaced and cover the whole input space. You can plot the input membership functions using the following commands.

```
subplot(2,2,1)
plotmf(fismat, 'input', 1)
subplot(2,2,2)
plotmf(fismat, 'input', 2)
subplot(2,2,3)
plotmf(fismat, 'input', 3)
subplot(2,2,4)
plotmf(fismat, 'input', 4)
```

These initial membership functions are shown below.



To start the training, type

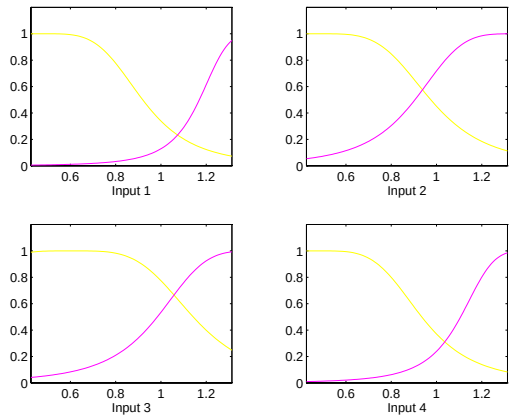
```
[fismat1,error1,ss,fismat2,error2] = ...
                        anfis(trnData,fismat,[],[],chkData);
```

This takes about four minutes on a Sun SPARCstation 2 for 10 epochs of training. Because the checking data option of anfis was invoked, the final FIS you choose would ordinarily be the one associated with the minimum checking

error. This is stored in `fismat2`. The following code will plot these new membership functions:

```
subplot(2,2,1)
plotmf(fismat2, 'input', 1)
subplot(2,2,2)
plotmf(fismat2, 'input', 2)
subplot(2,2,3)
plotmf(fismat2, 'input', 3)
subplot(2,2,4)
plotmf(fismat2, 'input', 4)
```

Here is the result:.



To plot the error signals type

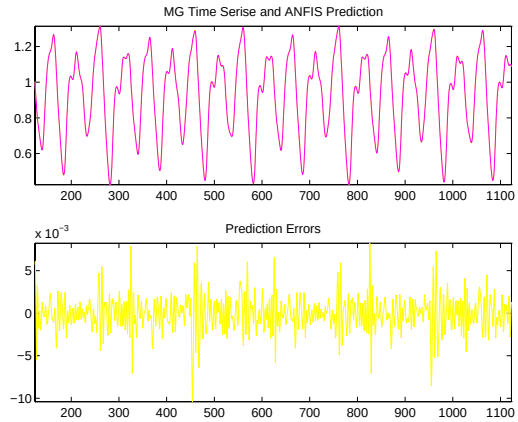
```
plot([error1; error2]);
```

Here `error1` and `error2` are the root mean squared error for the training and checking data, respectively.

In addition to these error plots, you may want to plot the FIS output versus the training or checking data. To compare the original MG time series and the fuzzy prediction side by side, try

```
anfis_output = evalfis([trnData; chkData], fismat2);
index = 125:1124;
```

```
subplot(211), plot(t(index), [x(index) anfis_output]);
subplot(212), plot(t(index), x(index) – anfis_output);
```



Note that the difference between the original MG time series and the `anfis` estimated values is very small. This is why you can only see one curve in the first plot. The prediction error is shown in the second plot with a much finer scale. Note that we have only trained for 10 epochs. Better performance is expected if we apply more extensive training.

More on anfis and the ANFIS Editor GUI

The command `anfis` takes at least two and at most six input arguments. The general format is

```
[fismat1,trnError,ss,fismat2,chkError] = ...
    anfis(trnData,fismat,trnOpt,dispOpt,chkData,method);
```

where `trnOpt` (training options), `dispOpt` (display options), `chkData` (checking data), and `method` (training method), are optional. All of the output arguments are also optional. In this section we discuss the arguments and range components of the command line function `anfis`, as well as the analogous functionality of the ANFIS Editor GUI.

When the ANFIS Editor GUI is invoked using `anfisedit`, only the training data set must exist prior to implementing `anfis`. In addition, the step-size will be fixed when the adaptive neuro-fuzzy system is trained using this GUI tool.

Training Data

The training data, `trnData`, is a required argument to `anfis`, as well as to the ANFIS Editor GUI. Each row of `trnData` is a desired input/output pair of the target system to be modeled. Each row starts with an input vector and is followed by an output value. Therefore, the number of rows of `trnData` is equal to the number of training data pairs, and, since there is only one output, the number of columns of `trnData` is equal to the number of inputs plus one.

Input FIS Structure

The input FIS structure, `fismat`, can be obtained either from any of the fuzzy editors: the FIS Editor, the Membership Function Editor, and the Rule Editor from the ANFIS Editor GUI, (which allows an FIS structure to be loaded from the disk or the workspace), or from the command line function, `genfis1` (for which you only need to give numbers and types of membership functions). The FIS structure contains both the model structure, (which specifies such items as the number of rules in the FIS, the number of membership functions for each input, etc.), and the parameters, (which specify the shapes of membership functions). There are two *methods* that `anfis` learning employs for updating membership function parameters: backpropagation for all parameters (a steepest descent method), and a hybrid method consisting of backpropagation for the parameters associated with the input membership functions, and least squares estimation for the parameters associated with the output membership functions. As a result, the training error decreases, at least locally, throughout the learning process. Therefore, the more the initial membership functions resemble the optimal ones, the easier it will be for the model parameter training to converge. Human expertise about the target system to be modeled may aid in setting up these initial membership function parameters in the FIS structure.

Note that `genfis1` produces an FIS structure based on a fixed number of membership functions. This invokes the so-called *curse of dimensionality*, and causes an explosion of the number of rules when the number of inputs is moderately large, that is, more than four or five. The Fuzzy Logic Toolbox offers a method that will provide for some dimension reduction in the fuzzy inference system: you can generate an FIS structure using the clustering algorithm discussed in “Subtractive Clustering” on page 2-123. From the ANFIS Editor GUI, this algorithm is selected with a radio button before the FIS is generated. This subtractive clustering method partitions the data into groups called clusters, and generates an FIS with the minimum number rules required to distinguish the fuzzy qualities associated with each of the clusters.

Training Options

The ANFIS Editor GUI tool allows you to choose your desired error tolerance and number of training epochs.

Training option `trnOpt` for the command line `anfis` is a vector that specifies the stopping criteria and the step-size adaptation strategy:

- `trnOpt(1)`: number of training epochs, default = 10.
- `trnOpt(2)`: error tolerance, default = 0.
- `trnOpt(3)`: initial step-size, default= 0.01.
- `trnOpt(4)`: step-size decrease rate, default = 0.9.
- `trnOpt(5)`: step-size increase rate, default = 1.1.

If any element of `trnOpt` is an `NaN` or missing, then the default value is taken. The training process stops if the designated epoch number is reached or the error goal is achieved, whichever comes first.

Usually we want the step-size profile to be a curve that increases initially, reaches some maximum, and then decreases for the remainder of the training. This ideal step-size profile is achieved by adjusting the initial step-size and the increase and decrease rates (`trnOpt(3)` - `trnOpt(5)`). The default values are set up to cover a wide range of learning tasks. For any specific application, you may want to modify these step-size options in order to optimize the training. However, as we mentioned previously, there are no user-specified step-size options for training the adaptive neuro fuzzy inference system generated using the ANFIS Editor GUI.

Display Options

Display options only apply to the command line function, `anfis`.

For the command line `anfis`, the display options *argument*, `dispOpt`, is a vector of either ones or zeros that specifies what information to display, (print in the MATLAB command line window), before, during, and after the training process. One is used to denote *print this option*, whereas zero denotes *don't print this option*.

- `dispOpt(1)`: display ANFIS information, default = 1.
- `dispOpt(2)`: display error (each epoch), default = 1.
- `dispOpt(3)`: display step-size (each epoch), default = 1.
- `dispOpt(4)`: display final results, default = 1.

The default mode is displays all available information. If any element of `dispOpt` is `NaN` or missing, the default value will be taken.

Method

Both the ANFIS Editor GUI and the command line `anfis` apply either a backpropagation form of the steepest descent method for membership function parameter estimation, or a combination of backpropagation and the least-squares method to estimate membership function parameters. The choices for this argument are `hybrid` or `backpropagation`. These method choices are designated in the command line function, `anfis`, by 1 and 0, respectively.

Output FIS Structure for Training Data

`fismat1` is the output FIS structure corresponding to a minimal training error. This is the FIS structure that you will use to represent the fuzzy system when there is no checking data used for model crossvalidation. This data also represents the FIS structure that is saved by the ANFIS Editor GUI when the checking data option is not used.

When the checking data option is used, the output saved is that associated with the minimum checking error.

Training Error

The training error is the difference between the training data output value, and the output of the fuzzy inference system corresponding to the same training data input value, (the one associated with that training data output value). The training error `trnError` records the root mean squared error (RMSE) of the training data set at each epoch. `fismat1` is the snapshot of the FIS structure when the training error measure is at its minimum. The ANFIS Editor GUI will plot the training error vs. epochs curve as the system is trained.

Step-size

You cannot control the step-size options with the ANFIS Editor GUI. Using the command line `anfis`, the step-size array `ss` records the step-size during the training. Plotting `ss` gives the step-size profile, which serves as a reference for adjusting the initial step-size and the corresponding decrease and increase

rates. The step-size (ss) for the command line function anfis is updated according to the following guidelines:

- If the error undergoes four consecutive reductions, increase the step-size by multiplying it by a constant (ssinc) greater than one.
- If the error undergoes two consecutive combinations of one increase and one reduction, decrease the step-size by multiplying it by a constant (ssdec) less than one.

The default value for the initial step-size is 0.01; the default values for ssinc and ssdec are 1.1 and 0.9, respectively. All the default values can be changed via the training option for the command line anfis.

Checking Data

The checking data, chkData, is used for testing the generalization capability of the fuzzy inference system at each epoch. The checking data has the same format as that of the training data, and its elements are generally distinct from those of the training data.

The checking data is important for learning tasks for which the input number is large, and/or the data itself is noisy. In general we want a fuzzy inference system to track a given input/output data set well. Since the model structure used for anfis is fixed, there is a tendency for the model to overfit the data on which is it trained, especially for a large number of training epochs. If overfitting does occur, we cannot expect the fuzzy inference system to respond well to other independent data sets, especially if they are corrupted by noise. A validation or checking data set can be useful for these situations. This data set is used to crossvalidate the fuzzy inference model. This crossvalidation is accomplished by applying the checking data to the model, and seeing how well the model responds to this data.

When the checking data option is used with anfis, either via the command line, or using the ANFIS Editor GUI, the checking data is applied to the model at each training epoch. When the command line anfis is invoked, the model parameters that correspond to the minimum checking error are returned via the output argument fismat2. The FIS membership function parameters computed using the ANFIS Editor GUI when both training and checking data are loaded are associated with the training epoch that has a minimum checking error.

The use of the minimum checking data error epoch to set the membership function parameters assumes

- The checking data is similar enough to the training data that the checking data error will decrease as the training begins
- The checking data increases at some point in the training, after which data overfitting has occurred.

As discussed in "ANFIS Editor GUI Example 2: Checking Data Doesn't Validate Model" on page 2-106, depending on the behavior of the checking data error, the resulting FIS may or may not be the one you should be using.

Output FIS Structure for Checking Data

The output of the command line `anfis`, `fismat2`, is the output FIS structure with the minimum checking error. This is the FIS structure that should be used for further calculation if checking data is used for cross validation.

Checking Error

The checking error is the difference between the checking data output value, and the output of the fuzzy inference system corresponding to the same checking data input value, (the one associated with that checking data output value). The checking error `chkError` records the RMSE for the checking data at each epoch. `fismat2` is the snapshot of the FIS structure when the checking error is at its minimum. The ANFIS Editor GUI will plot the checking error vs. epochs curve as the system is trained.

Fuzzy Clustering

Clustering of numerical data forms the basis of many classification and system modeling algorithms. The purpose of clustering is to identify natural groupings of data from a large data set to produce a concise representation of a system's behavior. The Fuzzy Logic Toolbox is equipped with some tools that allow you to find clusters in input-output training data. You can use the cluster information to generate a Sugeno-type fuzzy inference system that best models the data behavior using a minimum number of rules. The rules partition themselves according to the fuzzy qualities associated with each of the data clusters. This type of FIS generation can be accomplished automatically using the command line function, `genfis2`.

Fuzzy C-Means Clustering

Fuzzy c-means (FCM) is a data clustering technique wherein each data point belongs to a cluster to some degree that is specified by a membership grade. This technique was originally introduced by Jim Bezdek in 1981 [Bez81] as an improvement on earlier clustering methods. It provides a method of how to group data points that populate some multidimensional space into a specific number of different clusters?

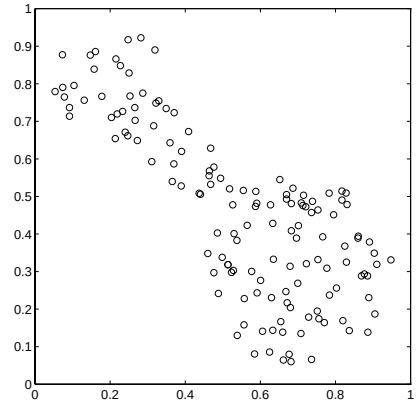
The Fuzzy Logic Toolbox command line function `fcm` starts with an initial guess for the cluster centers, which are intended to mark the mean location of each cluster. The initial guess for these cluster centers is most likely incorrect. Additionally, `fcm` assigns every data point a membership grade for each cluster. By iteratively updating the cluster centers and the membership grades for each data point, `fcm` iteratively moves the cluster centers to the "right" location within a data set. This iteration is based on minimizing an objective function that represents the distance from any given data point to a cluster center weighted by that data point's membership grade.

`fcm` is a command line function whose output is a list of cluster centers and several membership grades for each data point. You can use the information returned by `fcm` to help you build a fuzzy inference system by creating membership functions to represent the fuzzy qualities of each cluster.

An Example: 2-D Clusters

Let's use some quasi-random two-dimensional data to illustrate how FCM clustering works. Load a data set and take a look at it.

```
load fcmdata.dat
plot(fcmdata(:,1),fcmdata(:,2),'o')
```



Now we invoke the command line function, `fcm`, and ask it to find two clusters in this data set

```
[center,U,objFcn] = fcm(fcmdata,2);
    Iteration count = 1, obj. fcn = 8.941176
    Iteration count = 2, obj. fcn = 7.277177
```

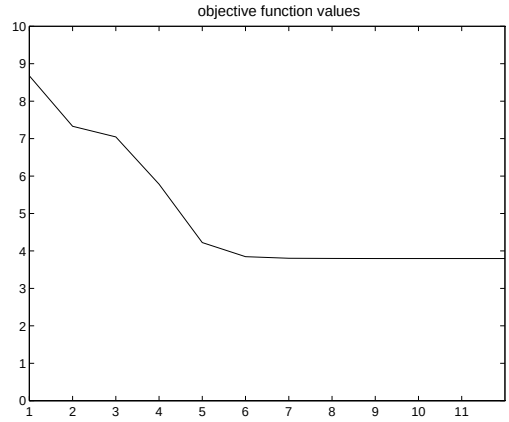
and so on until the objective function is no longer decreasing much at all.

The variable `center` contains the coordinates of the two cluster centers, `U` contains the membership grades for each of the data points, and `objFcn` contains a history of the objective function across the iterations.

The `fcm` function is an iteration loop built on top of several other routines, namely `initfcm`, which initializes the problem, `distfcm`, which is used for distance calculations, and `stepfcm`, which steps through one iteration.

Plotting the objective function shows the progress of the clustering.

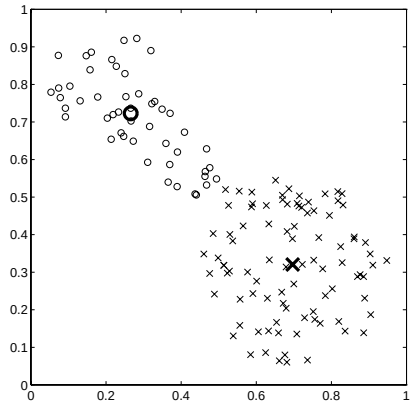
```
plot(objFcn)
```



Finally here is a plot displaying the two separate clusters as classified by the `fcm` routine. The following figure is generated using:

```
load fcmdata.dat
[center, U, obj_fcn] = fcm(fcmdata, 2);
maxU = max(U);
index1 = find(U(1, :) == maxU);
index2 = find(U(2, :) == maxU);
line(fcmdata(index1, 1), fcmdata(index1, 2), 'linestyle',...
'none','marker', 'o','color','g');
line(fcmdata(index2,1),fcmdata(index2,2),'linestyle',...
'none','marker', 'x','color','r');
hold on
plot(center(1,1),center(1,2),'ko','markersize',15,'LineWidth',2)
plot(center(2,1),center(2,2),'kx','markersize',15,'LineWidth',2)
```

Cluster centers are indicated in the figure below by the large characters.



Subtractive Clustering

Suppose we don't have a clear idea how many clusters there should be for a given set of data. *Subtractive clustering*, [Chi94], is a fast, one-pass algorithm for estimating the number of clusters and the cluster centers in a set of data. The cluster estimates obtained from the `subclust` function can be used to initialize iterative optimization-based clustering methods (`fcm`) and model identification methods (like `anfis`). The `subclust` function finds the clusters by using the subtractive clustering method.

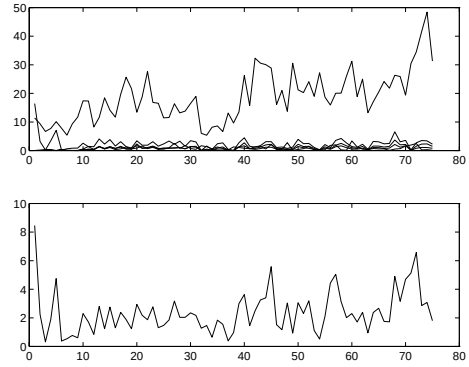
The `genfis2` function builds upon the `subclust` function to provide a fast, one-pass method to take input-output training data and generate a Sugeno-type fuzzy inference system that models the data behavior.

An Example: Suburban Commuting

In this example we apply the `genfis2` function to model the relationship between the number of automobile trips generated from an area and the area's demographics. Demographic and trip data are from 100 traffic analysis zones in New Castle County, Delaware. Five demographic factors are considered: population, number of dwelling units, vehicle ownership, median household income, and total employment. Hence the model has five input variables and one output variable.

Load the data by typing

```
tripdata
subplot(2,1,1), plot(datin)
subplot(2,1,2), plot(datout)
```



`tripdata` creates several variables in the workspace. Of the original 100 data points, we will use 75 data points as training data (`datin` and `datout`) and 25 data points as checking data, (as well as for test data to validate the model). The checking data input/output pairs are denoted by `chkdatin` and `chkdatout`. The `genfis2` function generates a model from data using clustering, and requires you to specify a cluster radius. The cluster radius indicates the range of influence of a cluster when you consider the data space as a unit hypercube. Specifying a small cluster radius will usually yield many small clusters in the data, (resulting in many rules). Specifying a large cluster radius will usually yield a few large clusters in the data, (resulting in fewer rules). The cluster radius is specified as the third argument of `genfis2`. Here we call the `genfis2` function using a cluster radius of 0.5.

```
fismat=genfis2(datin,datout,0.5);
```

`genfis2` is a fast, one-pass method that does not perform any iterative optimization. An FIS structure is returned; the model type for the FIS structure is a first order Sugeno model with three rules. We can use `evalfis` to verify the model.

```
fuzout=evalfis(datin,fismat);
trnRMSE=norm(fuzout-datout)/sqrt(length(fuzout))
```

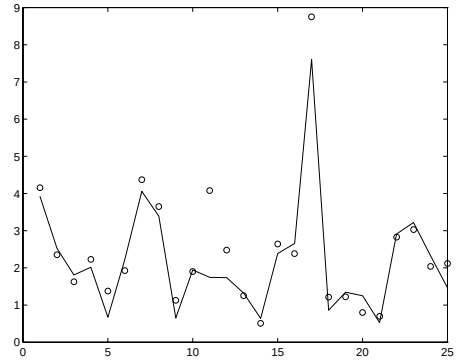
```
trnRMSE =
                    0.5276
```

The variable `trnRMSE` is the root mean square error of the system generated by the training data. To validate the generalizability of the model, we apply test data to the FIS. For this example, we use the checking data for both checking and testing the FIS parameters.

```
chkfuzout=evalfis(chkdatin,fismat);
chkRMSE=norm(chkfuzout-chkdatout)/sqrt(length(chkfuzout))
chkRMSE =
                    0.6170
```

Not surprisingly, the model doesn't do quite as good a job on the testing data. A plot of the testing data reveals the difference.

```
plot(chkdatout)
hold on
plot(chkfuzout,'o')
hold off
```



At this point, we can use the optimization capability of `anfis` to improve the model. First, we will try using a relatively short `anfis` training (50 epochs) without implementing the checking data option, but test the resulting FIS model against the test data. The command line version of this is as follows:

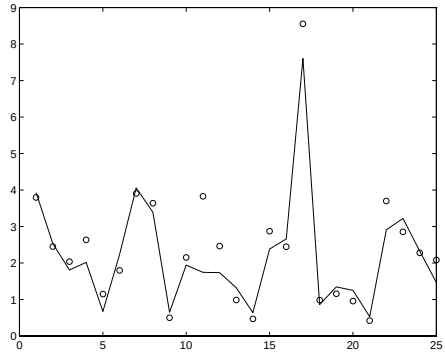
```
fismat2=anfis([datin datout],fismat,[50 0 0.1]);
```

After the training is done, we type:

```
fuzout2=evalfis(datin,fismat2);
trnRMSE2=norm(fuzout2-datout)/sqrt(length(fuzout2))
trnRMSE2 =
                    0.3407
chkfuzout2=evalfis(chkdatin,fismat2);
chkRMSE2=norm(chkfuzout2-chkdatout)/sqrt(length(chkfuzout2))
chkRMSE2 =
                    0.5827
```

The model has improved a lot with respect to the training data, but only a little with respect to the checking data. Here is a plot of the improved testing data.

```
plot(chkdatout)
hold on
plot(chkfuzout2,'o')
hold off
```



Here we see that `genfis2` can be used as a stand-alone, fast method for generating a fuzzy model from data, or as a pre-processor to `anfis` for determining the initial rules. An important advantage of using a clustering method to find rules is that the resultant rules are more tailored to the input data than they are in an FIS generated without clustering. This reduces the problem of combinatorial explosion of rules when the input data has a high dimension (the dreaded curse of dimensionality).

Overfitting

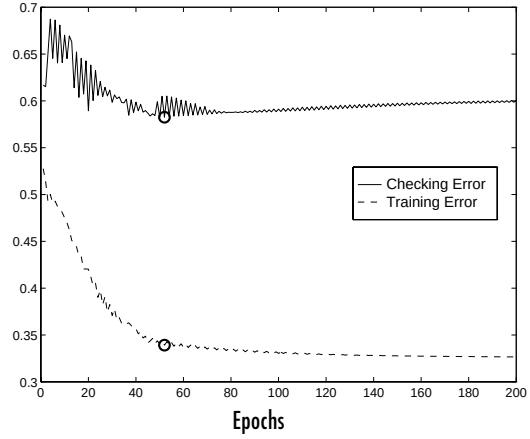
Now let's consider what happens if we carry out a longer (200 epoch) training of this system using `anfis`, including its checking data option.

```
[fismat3,trnErr,stepSize,fismat4,chkErr]= ...
                        anfis([datin datout],fismat2,[200 0
0.1],[], ...
                        [chkdatin chkdatout]);
```

The long list of output arguments returns a history of the step-sizes, the RMSE versus the training data, and the RMSE versus the checking data associated with each training epoch.

```
ANFIS training completed at epoch 200.
Minimal training RMSE = 0.326566
Minimal checking RMSE = 0.582545
```

This looks good. The error with the training data is the lowest we've seen, and the error with the checking data is also lower than before, though not by much. This suggests that maybe we had gotten about as close as possible with this system already. Maybe we have even gone so far as to overfit the system to the training data. Overfitting occurs when we fit the fuzzy system to the training data so well that it no longer does a very good job of fitting the checking data. The result is a loss of generality. A look at the error history against both the training data and the checking data reveals much.



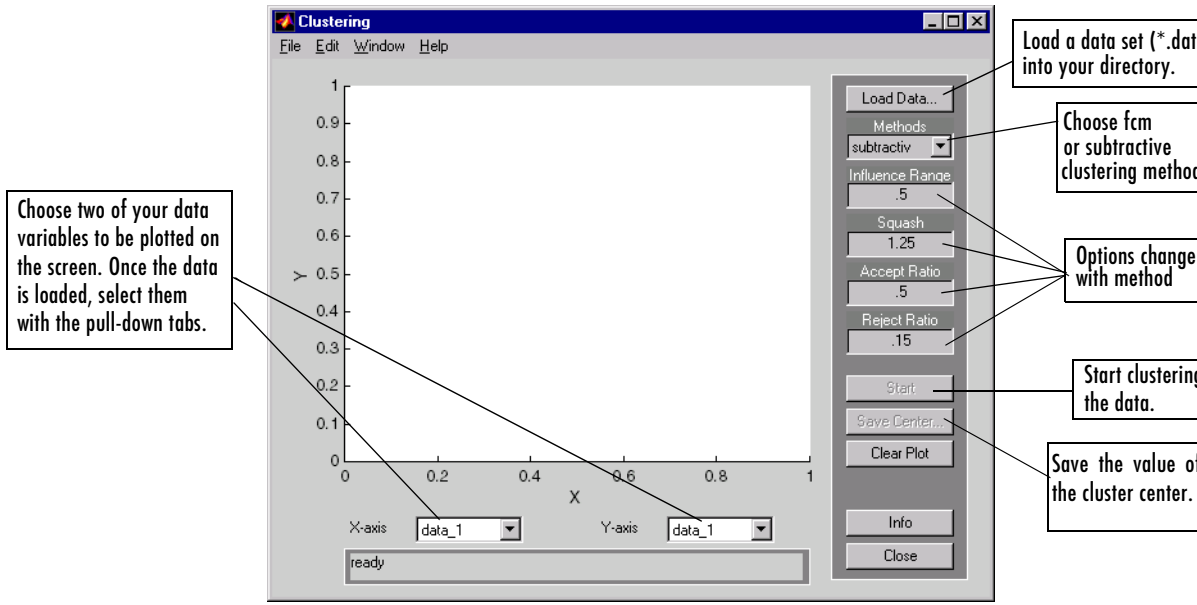
Here we can see that the training error settles at about the 50^{th} epoch point. In fact, the smallest value of the checking data error occurs at epoch 52, after which it increases slightly, even as `anfis` continues to minimize the error against the training data all the way to epoch 200. Depending on the specified

error tolerance, this plot also indicates the model's ability to generalize the test data.

A Clustering GUI Tool

There is also the Clustering GUI, which implements `fcm` and `subclust`, along with all of their options. Its use is fairly self-evident.

The clustering GUI looks like this, and is invoked using the command line function, `findcluster`.



You can invoke `findcluster` with a data set directly, in order to open the GUI with a data set. The data set must have the extension `.dat`. For example, to load the data set, `clusterdemo.dat`, type `findcluster('clusterdemo.dat')`.

You use the pull-down tab under **Method** to change between **fcm** (fuzzy c-means) and **subtractiv** (subtractive clustering). More information on the options can be found in the entries for `fcm` on page 3-22, and `subclust` on page 3-72, respectively.

The Clustering GUI works on multidimensional data sets, but only displays two of those dimensions. Use the pull-down tabs under **X-axis** and **Y-axis** to select which data dimension you want to view.

Stand-Alone C-Code Fuzzy Inference Engine

In the fuzzy/fuzzy directory of the toolbox, you can find two C files, fismain.c and fis.c, which are provided as the source codes for a stand-alone fuzzy inference engine. The stand-alone C-code fuzzy inference engine can read an FIS file and an input data file to perform fuzzy inference directly, or it can be embedded in other external applications.

To compile the stand-alone fuzzy inference engine on a UNIX system, type

```
% cc –O –o fismain fismain.c –lm
```

(Note that % is only symbolic of a UNIX prompt, and that you do not have to type fis.c explicitly, since it is included in fismain.c.) Upon successful compilation, type the executable command to see how it works:

```
% fismain
```

This prompts the following message:

```
% Usage: fismain data_file fis_file
```

This means that fismain needs two files to do its work: a data file containing rows of input vectors, and an FIS file that specifies the fuzzy inference system under consideration.

For example, consider an FIS structure file named, mam21.fis. We can prepare the input data file using MATLAB:

```
[x, y] = meshgrid(–5:5, –5:5);
input_data = [x(:) y(:)];
save fis_in input_data –ascii
```

This saves all the input data as a 121-by-2 matrix in the ASCII file fis_in, where each row of the matrix represents an input vector.

Now we can call the stand-alone code:

```
% fismain fis_in mam21.fis
```

This generates 121 outputs on your screen. You can direct the outputs to another file:

```
% fismain fis_in mam21.fis > fis_out
```

Now the file `fis_out` contains a 121-by-1 matrix. In general, each row of the output matrix represents an output vector. The syntax of `fismain` is similar to its MEX-file counterpart `evalfis.m`, except that all matrices are replaced with files.

To compare the results from the MATLAB MEX-file and the stand-alone executable, type the following within MATLAB:

```
fismat = readfis('mam21');
matlab_out = evalfis(input_data, fismat);
load fis_out
max(max(matlab_out – fis_out))
ans =
                        4.9583e–13
```

This tiny difference comes from the limited length printout in the file `fis_out`. There are several things you should know about this stand-alone executable:

- It is compatible with both ANSI and K & R standards for C code, as long as `__STDC__` is defined in ANSI compilers.
- Customized functions are not allowed in the stand-alone executable, so you are limited to the 11 membership functions that come with the toolbox, as well as other factory settings for AND, OR, IMP, and AGG functions.
- `fismain.c` contains only the `main()` function and it is heavily documented for easy adaptation to other applications.
- To add a new membership function or new reasoning mechanism into the stand-alone code, you need to change the file `fis.c`, which contains all the necessary functions to perform the fuzzy inference process.
- For the Macintosh, the compiled command `fismain` tries to find `fismain.in` and `fismain.fis` as input data and FIS description files, respectively. The output is stored in `fismain.out`. These filenames are defined within Macintosh-specific `#define` symbols in `fismain.c` and can be changed if necessary.

Glossary

This section is designed to briefly explain some of the specialized terms that are derived from fuzzy logic.

aggregation - the combination of the consequents of each rule in a Mamdani fuzzy inference system in preparation for defuzzification.

Adaptive Neuro-Fuzzy Inference System (ANFIS) - a technique for automatically tuning Sugeno-type inference systems based on training data.

antecedent - the initial (or "if") part of a fuzzy rule.

consequent - the final (or "then") part of a fuzzy rule.

defuzzification - the process of transforming a fuzzy output of a fuzzy inference system into a crisp output.

degree of membership - the output of a membership function, this value is always limited to between 0 and 1. Also known as a membership value or membership grade.

degree of fulfillment - see **firing strength**.

firing strength - the degree to which the antecedent part of a fuzzy rule is satisfied. The firing strength may be the result of an AND or an OR operation, and it shapes the output function for the rule. Also known as degree of fulfillment.

fuzzification - the process of generating membership values for a fuzzy variable using membership functions.

fuzzy c-means clustering - a data clustering technique wherein each data point belongs to a cluster to a degree specified by a membership grade.

fuzzy inference system (FIS) - the overall name for a system that uses fuzzy reasoning to map an input space to an output space.

fuzzy operators - AND, OR, and NOT operators. These are also known as logical connectives.

fuzzy set - a set which can contain elements with only a partial degree of membership.

fuzzy singleton - a fuzzy set with a membership function that is unity at a one particular point and zero everywhere else.

implication - the process of shaping the fuzzy set in the consequent based on the results of the antecedent in a Mamdani-type FIS.

Mamdani-type inference - a type of fuzzy inference in which the fuzzy sets from the consequent of each rule are combined through the aggregation operator and the resulting fuzzy set is defuzzified to yield the output of the system.

membership function (MF) - a function that specifies the degree to which a given input belongs to a set or is related to a concept.

singleton output function - an output function that is given by a spike at a single number rather than a continuous curve. In the Fuzzy Logic Toolbox it is only supported as part of a zero-order Sugeno model.

subtractive clustering - a technique for automatically generating fuzzy inference systems by detecting clusters in input-output training data.

Sugeno-type inference - a type of fuzzy inference in which the consequent of each rule is a linear combination of the inputs. The output is a weighted linear combination of the consequents.

T-conorm - (also known as S-norm) a two-input function that describes a superset of fuzzy union (OR) operators, including maximum, algebraic sum, and any of several parameterized T-conorms.

T-norm - a two-input function that describes a superset of fuzzy intersection (AND) operators, including minimum, algebraic product, and any of several parameterized T-norms.

References

[Bez81] Bezdek, J.C., *Pattern Recognition with Fuzzy Objective Function Algorithms,* Plenum Press, New York, 1981.

[Chi94] Chiu, S., "Fuzzy Model Identification Based on Cluster Estimation," *Journal of Intelligent & Fuzzy Systems*, Vol. 2, No. 3, Sept. 1994.

[Dub80] Dubois, D. and H. Prade, *Fuzzy Sets and Systems: Theory and Applications*, Academic Press, New York, 1980.

[Jan91] Jang, J.-S. R., "Fuzzy Modeling Using Generalized Neural Networks and Kalman Filter Algorithm," *Proc. of the Ninth National Conf. on Artificial Intelligence (AAAI-91)*, pp. 762-767, July 1991.

[Jan93] Jang, J.-S. R., "ANFIS: Adaptive-Network-based Fuzzy Inference Systems," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 23, No. 3, pp. 665-685, May 1993.

[Jan94] Jang, J.-S. R. and N. Gulley, "Gain scheduling based fuzzy controller design," *Proc. of the International Joint Conference of the North American Fuzzy Information Processing Society Biannual Conference, the Industrial Fuzzy Control and Intelligent Systems Conference, and the NASA Joint Technology Workshop on Neural Networks and Fuzzy Logic*, San Antonio, Texas, Dec. 1994.

[Jan95] Jang, J.-S. R. and C.-T. Sun, "Neuro-fuzzy modeling and control," *Proceedings of the IEEE*, March 1995.

[Jan97] Jang, J.-S. R. and C.-T. Sun, *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*, Prentice Hall, 1997.

[Kau85] Kaufmann, A. and M.M. Gupta, *Introduction to Fuzzy Arithmetic*, V.N. Reinhold, 1985.

[Lee90] Lee, C.-C., "Fuzzy logic in control systems: fuzzy logic controller-parts 1 and 2," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 20, No. 2, pp 404-435, 1990.

[Mam75] Mamdani, E.H. and S. Assilian, "An experiment in linguistic synthesis with a fuzzy logic controller," *International Journal of Man-Machine Studies*, Vol. 7, No. 1, pp. 1-13, 1975.

[Mam76] Mamdani, E.H., "Advances in the linguistic synthesis of fuzzy controllers," *International Journal of Man-Machine Studies*, Vol. 8, pp. 669-678, 1976.

[Mam77] Mamdani, E.H., "Applications of fuzzy logic to approximate reasoning using linguistic synthesis," *IEEE Transactions on Computers*, Vol. 26, No. 12, pp. 1182-1191, 1977.

[Sch63] Schweizer, B. and A. Sklar, "Associative functions and abstract semi-groups," *Publ. Math Debrecen*, 10:69-81, 1963.

[Sug77] Sugeno, M., "Fuzzy measures and fuzzy integrals: a survey," (M.M. Gupta, G. N. Saridis, and B.R. Gaines, editors) *Fuzzy Automata and Decision Processes*, pp. 89-102, North-Holland, New York, 1977.

[Sug85] Sugeno, M., *Industrial applications of fuzzy control*, Elsevier Science Pub. Co., 1985.

[Wan94] Wang, L.-X., *Adaptive fuzzy systems and control: design and stability analysis*, Prentice Hall, 1994.

[WidS85] Widrow, B. and D. Stearns, *Adaptive Signal Processing*, Prentice Hall, 1985.

[Yag80] Yager, R., "On a general class of fuzzy connectives," *Fuzzy Sets and Systems*, 4:235-242, 1980.

[Yag94] Yager, R. and D. Filev, "Generation of Fuzzy Rules by Mountain Clustering," *Journal of Intelligent & Fuzzy Systems*, Vol. 2, No. 3, pp. 209-219, 1994.

[Zad65] Zadeh, L.A., "Fuzzy sets," *Information and Control*, Vol. 8, pp. 338-353, 1965.

[Zad73] Zadeh, L.A., "Outline of a new approach to the analysis of complex systems and decision processes," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 3, No. 1, pp. 28-44, Jan. 1973.

[Zad75] Zadeh, L.A., "The concept of a linguistic variable and its application to approximate reasoning, Parts 1, 2, and 3," *Information Sciences*, 1975, 8:199-249, 8:301-357, 9:43-80

[Zad88] Zadeh, L.A., "Fuzzy Logic," *Computer*, Vol. 1, No. 4, pp. 83-93, 1988.

[Zad89] Zadeh, L.A., "Knowledge representation in fuzzy logic," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, pp. 89-100, 1989.

3

Reference

This section of the chapter contains brief descriptions of all the functions in the Fuzzy Logic Toolbox. The following tables contain the functions listed by topic.

GUI Tools

Function	Purpose
anfisedit	ANFIS Editor GUI.
fuzzy	Basic FIS Editor.
mfedit	Membership Function Editor.
ruleedit	Rule Editor and parser.
ruleview	Rule Viewer and fuzzy inference diagram.
surfview	Output Surface Viewer.

Membership Functions

Function	Purpose
dsigmf	Difference of two sigmoid membership functions.
gauss2mf	Two-sided Gaussian curve membership function.
gaussmf	Gaussian curve membership function.
gbellmf	Generalized bell curve membership function.
pimf	Pi-shaped curve membership function.

Function	Purpose
psigmf	Product of two sigmoidal membership functions.
smf	S-shaped curve membership function.
sigmf	Sigmoid curve membership function.
trapmf	Trapezoidal membership function.
trimf	Triangular membership function.
zmf	Z-shaped curve membership function.

FIS Data Structure Management

Function	Purpose
addmf	Add membership function to FIS.
addrule	Add rule to FIS.
addvar	Add variable to FIS.
defuzz	Defuzzify membership function.
evalfis	Perform fuzzy inference calculation.
evalmf	Generic membership function evaluation.
gensurf	Generate FIS output surface.
getfis	Get fuzzy system properties.
mf2mf	Translate parameters between functions.
newfis	Create new FIS.
parsrule	Parse fuzzy rules.
plotfis	Display FIS input-output structure.
plotmf	Plot all of the membership functions associated with a given variable.

Function	Purpose
readfis	Load FIS from disk.
rmmf	Remove membership function from FIS.
rmvar	Remove variable from FIS.
setfis	Set fuzzy system properties.
showfis	Display annotated FIS.
showrule	Display FIS rules.
writefis	Save FIS to disk.

Advanced Techniques

Function	Purpose
anfis	Training routine for a Sugeno-type FIS (MEX only).
fcm	Find clusters with FCM clustering.
genfis1	Generate FIS matrix using grid method.
genfis2	Generate FIS matrix using subtractive clustering.
subclust	Find cluster centers with subtractive clustering.

Simulink Blocks

Function	Purpose
fuzblock	Fuzzy logic controller blocks and demo blocks.
sffis	Fuzzy inference S-function.

Demos

Function	Purpose
`defuzzdm`	Defuzzification methods.
`fcmdemo`	FCM clustering demo (2-D).
`fuzdemos`	GUI for Fuzzy Logic Toolbox demos.
`gasdemo`	ANFIS demo for fuel efficiency using subclustering.
`juggler`	Ball-juggler with Rule Viewer.
`invkine`	Inverse kinematics of a robot arm.
`irisfcm`	FCM clustering demo (4-D).
`noisedm`	Adaptive noise cancellation.
`slbb`	Ball and beam control (Simulink).
`slcp`	Inverted pendulum control (Simulink).
`sltank`	Water level control (Simulink).
`sltankrule`	Water level control with Rule Viewer (Simulink).
`sltbu`	Truck backer-upper (Simulink only).

addmf

Purpose

Add a membership function to an FIS.

Synopsis

```
a = addmf(a,'varType',varIndex,'mfName','mfType',mfParams)
```

Description

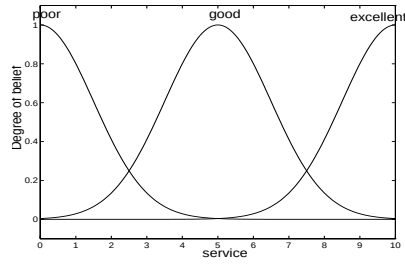
A membership function can only be added to a variable in an existing MATLAB workspace FIS. Indices are assigned to membership functions in the order in which they are added, so the first membership function added to a variable will always be known as membership function number one for that variable. You cannot add a membership function to input variable number two of a system if only one input has been defined.

The function requires six input arguments in this order:

1. A MATLAB variable name of a FIS structure in the workspace
2. A string representing the type of variable you want to add the membership function to (`'input'` or `'output'`)
3. The index of the variable you want to add the membership function to
4. A string representing the name of the new membership function
5. A string representing the type of the new membership function
6. The vector of parameters that specify the membership function

Example

```
a=newfis('tipper');
a=addvar(a,'input','service',[0 10]);
a=addmf(a,'input',1,'poor','gaussmf',[1.5 0]);
a=addmf(a,'input',1,'good','gaussmf',[1.5 5]);
a=addmf(a,'input',1,'excellent','gaussmf',[1.5 10]);
plotmf(a,'input',1)
```



See Also

addrule, addvar, plotmf, rmmf, rmvar

addrule

Purpose

Add a rule to an FIS.

Synopsis

```
a = addrule(a,ruleList)
```

Description

`addrule` has two arguments. The first argument is the MATLAB workspace variable FIS name. The second argument for `addrule` is a matrix of one or more rows, each of which represents a given rule. The format that the rule list matrix must take is very specific. If there are *m* inputs to a system and *n* outputs, there must be exactly *m* + *n* + 2 columns to the rule list.

The first *m* columns refer to the inputs of the system. Each column contains a number that refers to the index of the membership function for that variable.

The next *n* columns refer to the outputs of the system. Each column contains a number that refers to the index of the membership function for that variable.

The *m* + *n* + 1 column contains the weight that is to be applied to the rule. The weight must be a number between zero and one, and is generally left as one.

The *m* + *n* + 2 column contains a 1 if the fuzzy operator for the rule's antecedent is AND. It contains a 2 if the fuzzy operator is OR.

Example

```
ruleList=[
   1 1 1 1 1
   1 2 2 1 1];
a = addrule(a,ruleList);
```

If the above system `a` has two inputs and one output, the first rule can be interpreted as: "If input 1 is MF 1 and input 2 is MF 1, then output 1 is MF 1."

See Also

`addmf`, `addvar`, `rmmf`, `rmvar`, `parsrule`, `showrule`

Purpose

Add a variable to an FIS.

Synopsis

```
a = addvar(a,'varType','varName',varBounds)
```

Description

addvar has four arguments in this order:

1. The name of a FIS structure in the MATLAB workspace
2. A string representing the type of the variable you want to add ('input' or 'output')
3. A string representing the name of the variable you want to add
4. The vector describing the limiting range values for the variable you want to add

Indices are applied to variables in the order in which they are added, so the first input variable added to a system will always be known as input variable number one for that system. Input and output variables are numbered independently.

Example

```
a=newfis('tipper');
a=addvar(a,'input','service',[0 10]);
getfis(a,'input',1)
```

MATLAB replies

```
Name = service
NumMFs = 0
MFLabels =
Range = [0 10]
```

See Also

addmf, addrule, rmmf, rmvar

Purpose

Training routine for Sugeno-type FIS (MEX only).

Synopsis

```
[fismat,error1,stepsize] = anfis(trnData)
[fismat,error1,stepsize] = anfis(trnData,fismat)
[fismat1,error1,stepsize] = ...
   anfis(trnData,fismat,trnOpt,dispOpt)
[fismat1,error1,stepsize,fismat2,error2] = ...
   anfis(trnData,trnOpt,dispOpt,chkData)
[fismat1,error1,stepsize,fismat2,error2] = ...
   anfis(trnData,trnOpt,dispOpt,chkData,optMethod)
```

Description

This is the major training routine for Sugeno-type fuzzy inference systems. `anfis` uses a hybrid learning algorithm to identify parameters of Sugeno-type fuzzy inference systems. It applies a combination of the least-squares method and the backpropagation gradient descent method for training FIS membership function parameters to emulate a given training data set. `anfis` can also be invoked using an optional argument for model validation. The type of model validation that takes place with this option is a checking for model overfitting, and the argument is a data set called the checking data set.

The arguments in the above description for `anfis` are as follows:

- `trnData`: the name of a training data set. This is a matrix with all but the last column containing input data, while the last column contains a single vector of output data.
- `fismat`: the name of an FIS, (fuzzy inference system) used to provide `anfis` with an initial set of membership functions for training. Without this option, `anfis` will use `genfis1` to implement a default initial FIS for training. This default FIS will have two membership functions of the Gaussian type, when invoked with only one argument. If `fismat` is provided as a single number (or a vector), it is taken as the number of membership functions (or the vector whose entries are the respective numbers of membership functions associated with each respective input when these numbers differ for each input). In this case, both arguments of `anfis` are passed to `genfis1` to generate a valid FIS structure before starting the training process.

- `trnOpt`: vector of training options. When any training option is entered as `NaN` the default options will be in force. These options are as follows:

  `trnOpt(1)`: training epoch number (default: 10)

  `trnOpt(2)`: training error goal (default: 0)

  `trnOpt(3)`: initial step size (default: 0.01)

  `trnOpt(4)`: step size decrease rate (default: 0.9)

  `trnOpt(5)`: step size increase rate (default: 1.1)

- `dispOpt`: vector of display options that specify what message to display in the MATLAB command window during training. The default value for any display option is `1`, which means the corresponding information is displayed. A `0` means the corresponding information is not displayed on the screen. When any display option is entered as `NaN`, the default options will be in force. These options are as follows:

  `dispOpt(1)`: ANFIS information, such as numbers of input and output membership functions, and so on (default: 1)

  `dispOpt(2)`: error (default: 1)

  `dispOpt(3)`: step size at each parameter update (default: 1)

  `dispOpt(4)`: final results (default: 1)

- `chkData`: the name of an optional checking data set for overfitting model validation. This data set is a matrix in the same format as the training data set.

- `optMethod`: optional optimization method used in membership function parameter training: either `1` for the hybrid method or `0` for the backpropagation method. The default method is the hybrid method, which is a combination of least squares estimation with backpropagation. The default method is invoked whenever the entry for this argument is anything but `0`.

The training process stops whenever the designated epoch number is reached or the training error goal is achieved.

Note on anfis arguments: When `anfis` is invoked with two or more arguments, any optional arguments will take on their default values if they are entered as `NaN`s or empty matrices. Default values can be changed directly by modifying the file `anfis.m`. Either `NaN`s or empty matrices must be used as place-holders for variables if you don't want to specify them, but do want to specify succeeding arguments, for example, when you implement the checking data option of `anfis`.

The range variables in the above description for `anfis` are as follows:

- `fismat1` is the FIS structure whose parameters are set according to a minimum training error criterion.
- `error1 or error2` is an array of root mean squared errors representing the training data error signal and the checking data error signal, respectively.
- `stepsize` is an array of step sizes. The step size is decreased (by multiplying it with the component of the training option corresponding to the step size decrease rate) if the error measure undergoes two consecutive combinations of an increase followed by a decrease. The step size is increased (by multiplying it with the increase rate) if the error measure undergoes four consecutive decreases.
- `fismat2` is the FIS structure whose parameters are set according to a minimum checking error criterion.

Example

```
x = (0:0.1:10)';
y = sin(2*x)./exp(x/5);
trnData = [x y];
numMFs = 5;
mfType = 'gbellmf';
epoch_n = 20;
in_fismat = genfis1(trnData,numMFs,mfType);
out_fismat = anfis(trnData,in_fismat,20);
plot(x,y,x,evalfis(x,out_fismat));
legend('Training Data','ANFIS Output');
```

See Also

genfis1, anfis

References

Jang, J.-S. R., “Fuzzy Modeling Using Generalized Neural Networks and Kalman Filter Algorithm,” *Proc. of the Ninth National Conf. on Artificial Intelligence (AAAI-91)*, pp. 762-767, July 1991.

Jang, J.-S. R., “ANFIS: Adaptive-Network-based Fuzzy Inference Systems,” *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 23, No. 3, pp. 665-685, May 1993.

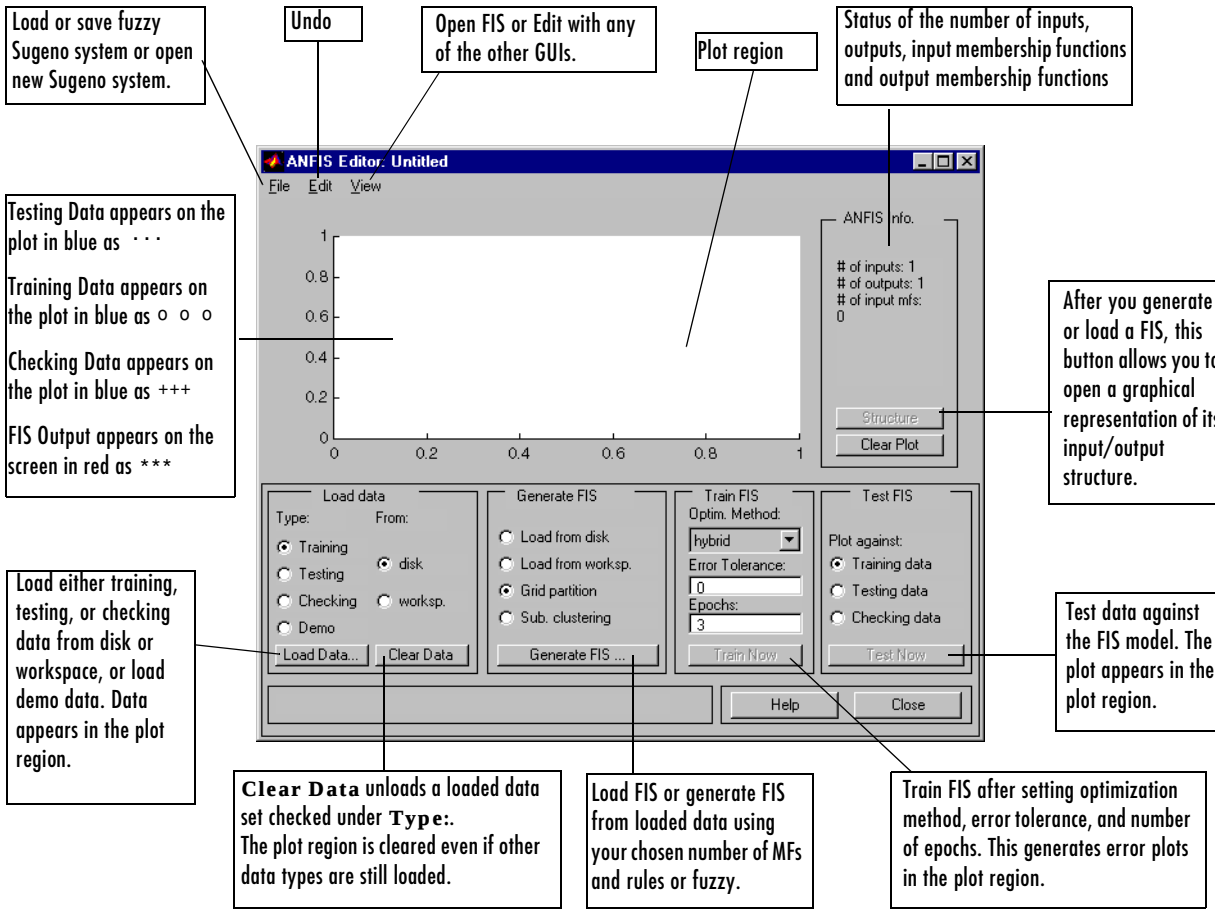
Purpose

To open the ANFIS Editor GUI.

Synopsis

```
anfisedit('a')
anfisedit(a)
anfisedit
```

Description



Using `anfisedit`, you bring up the ANFIS Editor GUI from which you can load a data set and train `anfis`. The ANFIS Editor GUI invoked using

`anfisedit('a')`, brings up the ANFIS Editor GUI from which you can implement `anfis` using a FIS structure stored as a file on your disk called, `a.fis`.

`anfisedit(a)` operates the same way for a FIS structure `a`, stored as a variable in the MATLAB workspace.

Refer to "anfis and the ANFIS Editor GUI" on page 2-92 for more information about how to use `anfisedit`.

Menu Items

On the ANFIS Editor GUI, there is a menu bar that allows you to open related GUI tools, open and save systems, and so on. The **File** menu is the same as the one found on the FIS Editor. Refer to `fuzzy` on page 3-29 for more information.

- Use the following **Edit** menu item:

  Undo to undo the most recent change.
- Use the following **View** menu items:

  Edit FIS properties... to invoke the FIS Editor.

  Edit rules... to invoke the Rule Editor.

  Edit membership functions... to invoke the Membership Function Editor.

  View rules... to invoke the Rule Viewer.

  View surface... to invoke the Surface Viewer.

See Also

`fuzzy`, `mfedit`, `ruleedit`, `ruleview`, `surfview`

convertfis

Purpose

Convert a Fuzzy Logic Toolbox version 1.0 FIS matrix to a version 2.0 FIS structure.

Synopsis

```
fis_new=convertfis(fis_old)
```

Description

`convertfis` takes a version 1.0 FIS matrix and converts it to a version 2.0 structure.

Purpose

Defuzzify membership function.

Synopsis

```
out = defuzz(x,mf,type)
```

Description

`defuzz(x,mf,type)` returns a defuzzified value `out`, of a membership function `mf` positioned at associated variable value `x`, using one of several defuzzification strategies, according to the argument, `type`. The variable `type` can be one of the following.

- `centroid`: centroid of area method
- `bisector`: bisector of area method
- `mom`: mean of maximum method
- `som`: smallest of maximum method
- `lom`: largest of maximum method

If `type` is not one of the above, it is assumed to be a user-defined function. `x` and `mf` are passed to this function to generate the defuzzified output.

Examples

```
x = −10:0.1:10;
mf = trapmf(x,[−10 −8 −4 7]);
xx = defuzz(x,mf,'centroid');
```

dsigmf

Purpose

Built-in membership function composed of the difference between two sigmoidal membership functions.

Synopsis

```
y = dsigmf(x,[a1 c1 a2 c2])
```

Description

The sigmoidal membership function used here depends on the two parameters a and c and is given by

$$f(x;a,c) = \frac{1}{1+e^{-a(x-c)}}$$

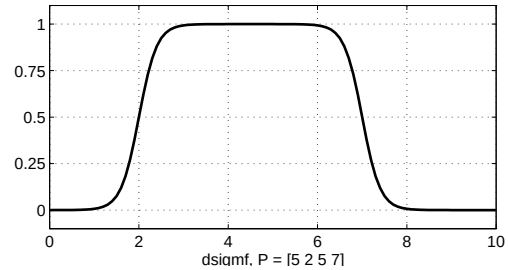
The membership function `dsigmf` depends on four parameters, `a1`, `c1`, `a2`, and `c2`, and is the difference between two of these sigmoidal functions:

$f_1(x; a1, c_1) - f_2(x; a2, c_2)$

The parameters are listed in the order: [$a1\ c_1\ a_2\ c_2$].

Example

```
x=0:0.1:10;
y=dsigmf(x,[5 2 5 7]);
plot(x,y)
xlabel('dsigmf, P=[5 2 5 7]')
```



See Also

`gaussmf`, `gauss2mf`, `gbellmf`, `evalmf`, `mf2mf`, `pimf`, `psigmf`, `sigmf`, `smf`, `trapmf`, `trimf`, `zmf`

Purpose

Perform fuzzy inference calculations.

Synopsis

```
output= evalfis(input,fismat)
output= evalfis(input,fismat, numPts)
[output, IRR, ORR, ARR]= evalfis(input,fismat)
[output, IRR, ORR, ARR]= evalfis(input,fismat, numPts)
```

Description

`evalfis` has the following arguments:

- `input`: a number or a matrix specifying input values. If `input` is an M-by-N matrix, where N is number of input variables, then `evalfis` takes each row of `input` as an input vector and returns the M-by-L matrix to the variable, `output`, where each row is an output vector and L is the number of output variables.
- `fismat`: an FIS structure to be evaluated.
- numPts: an optional argument that represents the number of sample points on which to evaluate the membership functions over the input or output range. If this argument is not used, the default value of 101 point is used.

The range labels for `evalfis` are as follows:

- `output`: the output matrix of size M-by-L, where M represents the number of input values specified above, and L is the number of output variables for the FIS.

The optional range variables for `evalfis` are only calculated when the `input` argument is a row vector, (only one set of inputs is applied). These optional range variables are:

- `IRR`: the result of evaluating the input values through the membership functions. This is a matrix of size *numRules*-by-*N*, where *numRules* is the number of rules, and *N* is the number of input variables.
- `ORR`: the result of evaluating the output values through the membership functions. This is a matrix of size `numPts`-by-*numRules*L*, where *numRules* is the number of rules, and *L* is the number of outputs. The first *numRules* columns of this matrix correspond to the first output, the next *numRules* columns of this matrix correspond to the second output, and so forth.
- `ARR`: the `numPts`-by-*L* matrix of the aggregate values sampled at `numPts` along the output range for each output.

When invoked with only one range variable, this function computes the output vector, `output`, of the fuzzy inference system specified by the structure, `fismat`, for the input value specified by the number or matrix, `input`.

Example

```
fismat = readfis('tipper');
out = evalfis([2 1; 4 9],fismat)
```

This generates the response

```
out =
   7.0169
   19.6810
```

See Also

`ruleview`, `gensurf`

Purpose

Generic membership function evaluation.

Synopsis

```
y = evalmf(x,mfParams,mfType)
```

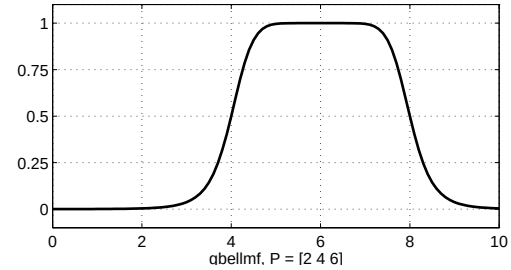
Description

`evalmf` evaluates any membership function, where `x` is the variable range for the membership function evaluation, `mfType` is a membership function from the toolbox, and `mfParams` are appropriate parameters for that function.

If you want to create your own custom membership function, `evalmf` will still work, because it evaluates any membership function whose name it doesn't recognize.

Examples

```
x=0:0.1:10;
mfparams = [2 4 6];
mftype = 'gbellmf';
y=evalmf(x,mfparams,mftype);
plot(x,y)
xlabel('gbellmf, P=[2 4 6]')
```



See Also

`dsigmf`, `gaussmf`, `gauss2mf`, `gbellmf`, `mf2mf`, `pimf`, `psigmf`, `sigmf`, `smf`, `trapmf`, `trimf`, `zmf`

fcm

Purpose

Fuzzy c-means clustering.

Synopsis

```
[center,U,obj_fcn] = fcm(data,cluster_n)
```

Description

`[center, U, obj_fcn] = fcm(data, cluster_n)` applies the fuzzy c-means clustering method to a given data set.

The input arguments of this function are:

- `data`: data set to be clustered; each row is a sample data point
- `cluster_n`: number of clusters (greater than one)

The output arguments of this function are:

- `center`: matrix of final cluster centers where each row provides the center coordinates
- `U`: final fuzzy partition matrix (or membership function matrix)
- `obj_fcn`: values of the objective function during iterations

`fcm(data,cluster_n,options)` uses an additional argument variable, `options`, to control clustering parameters, introduce a stopping criteria, and/or set the iteration information display:

`options(1)`: exponent for the partition matrix `U` (default: 2.0)

`options(2)`: maximum number of iterations (default: 100)

`options(3)`: minimum amount of improvement (default: 1e-5)

`options(4)`: info display during iteration (default: 1)

If any entry of `options` is `NaN`, the default value for that option is used instead. The clustering process stops when the maximum number of iterations is reached, or when the objective function improvement between two consecutive iterations is less than the minimum amount of improvement specified.

Example

```
data = rand(100, 2);
[center,U,obj_fcn] = fcm(data, 2);
plot(data(:,1), data(:,2),'o');
maxU = max(U);
index1 = find(U(1,:) == maxU);
index2 = find(U(2, :) == maxU);
line(data(index1,1), data(index1, 2), 'linestyle', 'none',
'marker', '*',
'color', 'g');
line(data(index2,1), data(index2, 2), 'linestyle', 'none',
'marker', '*',
'color', 'r');
```

findcluster

Purpose

Interactive clustering GUI for fuzzy c-means and subclustering.

Synopsis

```
findcluster
findcluster('file.dat')
```

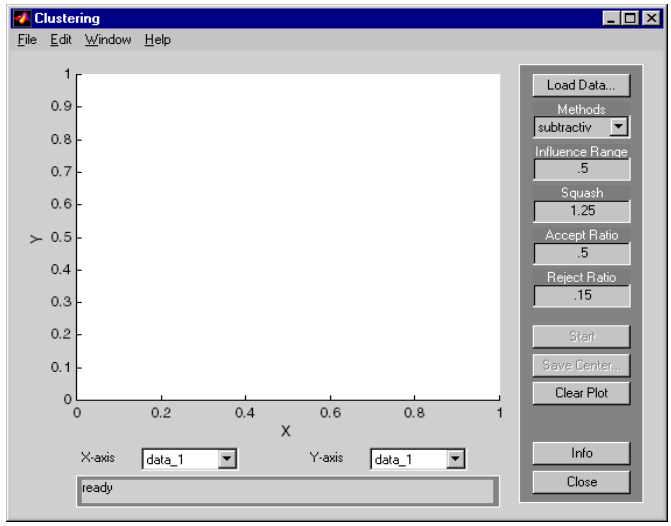
Description

`findcluster` brings up a GUI to implement fuzzy c-means (**fcm**) and/or fuzzy subtractive clustering (**subtractiv**) using the pull-down tab under **Method** on the GUI. Data is entered using the **Load Data** button. The options for each of these methods are set to default values. These can be changed. A description of the options for fuzzy c-means is found in `fcm` on page 3-22. A description of the options for fuzzy subclustering is found in `subclust` on page 3-72.

This tool works on multidimensional data sets, but only displays two of those dimensions. Use the pull-down tabs under **X-axis** and **Y-axis** to select which data dimension you want to view. For example, if you have data that is five-dimensional, this tool labels the data as **data_1**, **data_2**, **data_3**, **data_4**, **data_5**, in the order in which the data appears in the data set. **Start** will perform the clustering, and **Save Center** will save the cluster center.

When operating on a data set, `file.dat`, `findcluster` (`file.dat`) loads the data set automatically, plotting up to the first two dimensions of the data only.

You can still choose which two dimensions of the data you want to cluster after the GUI comes up.



Example

```
findcluster('clusterdemo.dat')
```

See Also

`fcm`, `subclust`

fuzarith

Purpose

To perform fuzzy arithmetic.

Synopsis

```
C = fuzarith(X, A, B, operator)
```

Description

Using interval arithmetic, `C = fuzarith(X, A, B, operator)` returns a fuzzy set `C` as the result of applying the function represented by the string, `operator`, that performs a binary operation on the sampled convex fuzzy sets `A` and `B`. The elements of `A` and `B` are derived from convex functions of the sampled universe, `X`.

- `A`, `B`, and `X` are vectors of the same dimension.
- `operator` is one of the following strings: `'sum'`, `'sub'`, `'prod'`, and `'div'`.
- The returned fuzzy set `C` is a column vector with the same length as `X`.

Remark

Fuzzy addition might generate the message: *divide by zero*, but this will not affect the correctness of this function.

Example

```
point_n = 101;% this determines MF's resolution
   min_x = -20; max_x = 20;% universe is [min_x, max_x]
   x = linspace(min_x, max_x, point_n)';
   A = trapmf(x, [-10 -2 1 3]);% trapezoidal fuzzy set A
   B = gaussmf(x, [2 5]);% Gaussian fuzzy set B
   C1 = fuzarith(x, A, B, 'sum');
   subplot(2,1,1);
   plot(x, A, 'b--', x, B, 'm:', x, C1, 'c');
   title('fuzzy addition A+B');
   C2 = fuzarith(x, A, B, 'sub');
   subplot(2,1,2);
   plot(x, A, 'b--', x, B, 'm:', x, C2, 'c');
   title('fuzzy subtraction A-B');
   C3 = fuzarith(x, A, B, 'prod');
```

Purpose

Simulink fuzzy logic controller block.

Synopsis

```
fuzblock
```

Description

This command brings up a Simulink system that, in addition to some Simulink demo blocks, contains two Simulink blocks you can use:

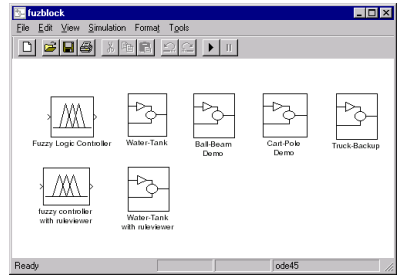
- The Fuzzy Logic Controller
- The Fuzzy Logic Controller With Rule Viewer, (see also `ruleview` on page 3-61). This block forces the Rule Viewer to pop open during a Simulink simulation.

The dialog box associated with either of these blocks is found by double-clicking on the Fuzzy Logic Controller block. This box contains the name of the FIS structure in the workspace that corresponds to the desired fuzzy system you want in your Simulink model.

To open this dialog box for the Fuzzy Logic Controller With Rule Viewer block, you have to

1. Double-click on this block, and a Simulink diagram with a Fuzzy Logic Controller block opens.
2. Double-click on the second Fuzzy Logic Controller block that pops open.

If the fuzzy inference system has multiple inputs, these inputs should be multiplexed together before feeding them into either the Fuzzy Logic Controller or the Fuzzy Logic Controller With Rule Viewer block. Similarly, if the system has multiple outputs, these signals will be passed out of the block on one multiplexed line.



See Also

`sffis`, `ruleview`

fuzdemos

Purpose

List of all Fuzzy Logic Toolbox demos.

Synopsis

```
fuzdemos
```

Description

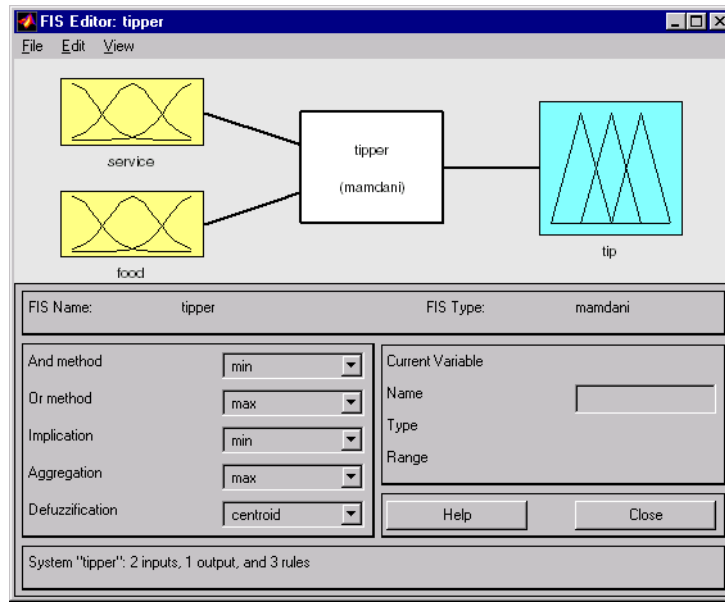
This function brings up a GUI that allows you to choose between any of several Fuzzy Logic Toolbox demos listed under “Demos” on page 3-5.

Purpose

To invoke the basic FIS editor.

Synopsis

```
fuzzy
fuzzy(fismat)
```



This GUI tool allows you to edit the highest level features of the fuzzy inference system, such as the number of input and output variables, the defuzzification method used, and so on. Refer to “The FIS Editor” on page 2-49 and ff., for more information about how to use the GUIs associated with `fuzzy`.

The FIS Editor is the high-level display for any fuzzy logic inference system. It allows you to call the various other editors to operate on the FIS. This interface allows convenient access to all other editors with an emphasis on maximum flexibility for interaction with the fuzzy system.

The Diagram

The diagram displayed at the top of the window shows the inputs, outputs, and a central fuzzy rule processor. Click on one of the variable boxes to make the selected box the current variable. You should see the box highlighted in red. Double-click on one of the variables to bring up the Membership Function Editor. Double-click on the fuzzy rule processor to bring up the Rule Editor. If

a variable exists but is not mentioned in the rule base, it is connected to the rule processor block with a dashed rather than a solid line.

Menu Items

The FIS Editor displays a menu bar that allows you to open related GUI tools, open and save systems, and so on.

- Under **File** select:

  New Mamdani FIS... to open a new Mamdani-style system with no variables and no rules called `Untitled`.

  New Sugeno FIS... to open a new Sugeno-style system with no variables and no rules called `Untitled`.

  Open from disk... to load a system from a specified `.fis` file on disk.

  Save to disk to save the current system to a `.fis` file on disk.

  Save to disk as... to save the current system to disk with the option to rename or relocate the file.

  Open from workspace... to load a system from a specified FIS structure variable in the workspace.

  Save to workspace... to save the system to the currently named FIS structure variable in the workspace.

  Save to workspace as... to save the system to a specified FIS structure variable in the workspace.

  Close window to close the GUI.

- Under **Edit** select:

  Add input to add another input to the current system.

  Add output to add another output to the current system.

  Remove variable to delete a selected variable.

  Undo to undo the most recent change.

- Under **View** select:

  Edit MFs... to invoke the Membership Function Editor.

  Edit rules... to invoke the Rule Editor.

  Edit anfis... to invoke the ANFIS Editor for single output Sugeno systems only.

  View rules... to invoke the Rule Viewer.

  View surface... to invoke the Surface Viewer.

Inference Method Pop-up Menus

Five pop-up menus are provided to change the functionality of the five basic steps in the fuzzy implication process:

- **And method**: Choose `min`, `prod`, or `Custom`, for a custom operation.
- **Or method**: Choose `max`, `probor` (probabilistic or), or `Custom`, for a custom operation.
- **Implication method**: Choose `min`, `prod`, or `Custom`, for a custom operation. This selection is not available for Sugeno-style fuzzy inference.
- **Aggregation method**: Choose `max`, `sum`, `probor`, or `Custom`, for a custom operation. This selection is not available for Sugeno-style fuzzy inference.
- **Defuzzification method**: For Mamdani-style inference, choose `centroid`, `bisector`, `mom` (middle of maximum), `som` (smallest of maximum), `lom` (largest of maximum), or `Custom`, for a custom operation. For Sugeno-style inference, choose between `wtaver` (weighted average) or `wtsum` (weighted sum).

See Also

`mfedit`, `ruleedit`, `ruleview`, `surfview`, `anfisedit`

gauss2mf

Purpose

Gaussian combination membership function.

Synopsis

```
y = gauss2mf(x,[sig1 c1 sig2 c2])
```

Description

The Gaussian function depends on two parameters *sig* and *c* as given by

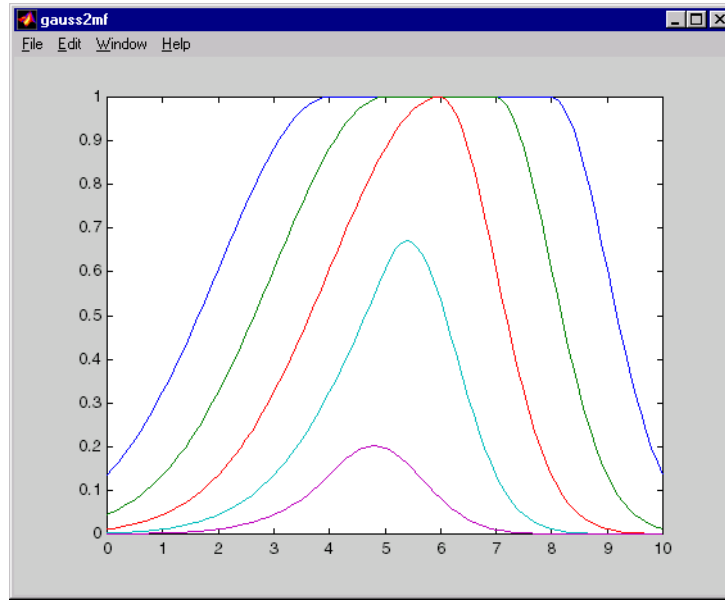
$$f(x;\sigma, c) = e^{\frac{-(x-c)^2}{2\sigma^2}}$$

The function `gauss2mf` is a combination of two of these. The first function, specified by *sig1* and *c1*, determines the shape of the leftmost curve. The second function specified by *sig2* and *c2* determines the shape of the right-most curve. Whenever *c1* < *c2*, the `gauss2mf` function reaches a maximum value of 1. Otherwise, the maximum value is less than one. The parameters are listed in the order:

[*sig1*, *c1*, *sig2*, *c2*].

Examples

```
x = (0:0.1:10)';
        y1 = gauss2mf(x, [2 4 1 8]);
        y2 = gauss2mf(x, [2 5 1 7]);
        y3 = gauss2mf(x, [2 6 1 6]);
        y4 = gauss2mf(x, [2 7 1 5]);
        y5 = gauss2mf(x, [2 8 1 4]);
        plot(x, [y1 y2 y3 y4 y5]);
        set(gcf, 'name', 'gauss2mf', 'numbertitle', 'off');
```



See Also

dsigmf, gauss2mf, gbellmf, evalmf, mf2mf, pimf, psigmf, sigmf, smf, trapmf, trimf, zmf

gaussmf

Purpose

Gaussian curve built-in membership function.

Synopsis

```
y = gaussmf(x,[sig c])
```

Description

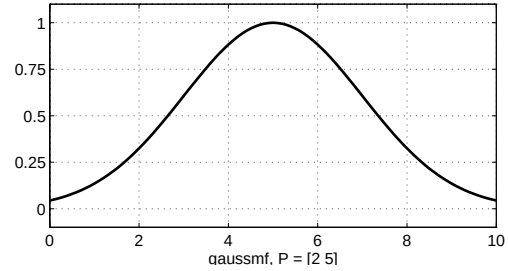
The symmetric Gaussian function depends on two parameters σ and c as given by

$$f(x;\sigma, c) = e^{\frac{-(x-c)^2}{2\sigma^2}}$$

The parameters for `gaussmf` represent the parameters σ and c listed in order in the vector `[sig c]`.

Example

```
x=0:0.1:10;
y=gaussmf(x,[2 5]);
plot(x,y)
xlabel('gaussmf, P=[2 5]')
```



See Also

`dsigmf`, `gaussmf`, `gbellmf`, `evalmf`, `mf2mf`, `pimf`, `psigmf`, `sigmf`, `smf`, `trapmf`, `trimf`, `zmf`

Purpose

Generalized bell-shaped built-in membership function.

Synopsis

```
y = gbellmf(x,params)
```

Description

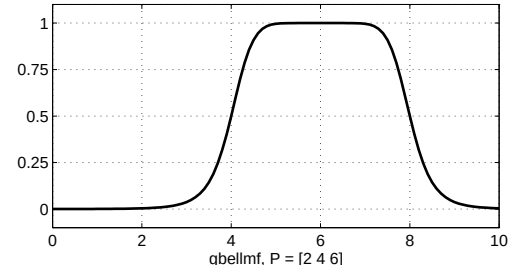
The generalized bell function depends on three parameters *a*, *b*, and *c* as given by

$$f(x;a,b,c) = \frac{1}{1+\left|\frac{x-c}{a}\right|^{2b}}$$

where the parameter *b* is usually positive. The parameter *c* locates the center of the curve. Enter the parameter vector `params`, the second argument for `gbellmf`, as the vector whose entries are *a*, *b*, and *c*, respectively.

Example

```
x=0:0.1:10;
y=gbellmf(x,[2 4 6]);
plot(x,y)
xlabel('gbellmf, P=[2 4 6]')
```



See Also

`dsigmf`, `gaussmf`, `gauss2mf`, `evalmf`, `mf2mf`, `pimf`, `psigmf`, `sigmf`, `smf`, `trapmf`, `trimf`, `zmf`

genfis1

Purpose

Generate an FIS structure from data without data clustering.

Synopsis

```
fismat = genfis1(data)
fismat = genfis1(data,numMFs,inmftype, outmftype)
```

Description

`genfis1` generates a Sugeno-type FIS structure used as initial conditions (initialization of the membership function parameters) for `anfis` training. `genfis1(data, numMFs, inmftype, outmftype)` generates a FIS structure from a training data set, `data`, using a grid partition on the data (no clustering).

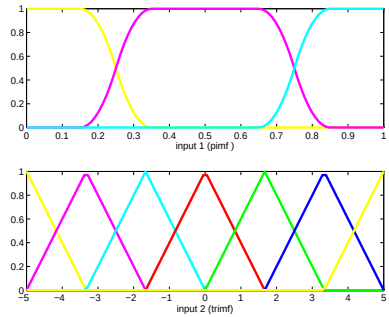
The arguments for `genfis1` are as follows:

- `data` is the training data matrix, which must be entered with all but the last columns representing input data, and the last column representing the single output.
- `numMFs` is a vector whose coordinates specify the number of membership functions associated with each input. If you want the same number of membership functions to be associated with each input, then it suffices to make `numMFs` a single number.
- `inmftype` is a string array in which each row specifies the membership function type associated with each input. Again, this can be a one-dimensional single string if the type of membership functions associated with each input is the same.
- `outmftype` is a string that specifies the membership function type associated with the output. There can only be one output, since this is a Sugeno-type system. The output membership function type must be either *linear* or *constant*.

The number of membership functions associated with the output is the same as the number of rules generated by `genfis1`. The default number of membership functions, `numMFs`, is 2; the default input or output membership function type is `'gbellmf'`. These are used whenever `genfis1` is invoked without the last three arguments.

Example

```
data = [rand(10,1) 10*rand(10,1)-5 rand(10,1)];
numMFs = [3 7];
mfType = str2mat('pimf','trimf');
fismat = genfis1(data,numMFs,mfType);
[x,mf] = plotmf(fismat,'input',1);
subplot(2,1,1), plot(x,mf);
xlabel('input 1 (pimf)');
[x,mf] = plotmf(fismat,'input',2);
subplot(2,1,2), plot(x,mf);
xlabel('input 2 (trimf)');
```



See Also

anfis

Purpose

Generate an FIS structure from data using subtractive clustering.

Synopsis

```
fismat = genfis2(Xin,Xout,radii)
fismat = genfis2(Xin,Xout,radii,xBounds)
fismat = genfis2(Xin,Xout,radii,xBounds,options)
```

Description

Given separate sets of input and output data, `genfis2` generates an FIS using fuzzy subtractive clustering. When there is only one output, `genfis2` may be used to generate an initial FIS for `anfis` training by first implementing subtractive clustering on the data. `genfis2` accomplishes this by extracting a set of rules that models the data behavior. The rule extraction method first uses the `subclust` function to determine the number of rules and antecedent membership functions and then uses linear least squares estimation to determine each rule's consequent equations. This function returns an FIS structure that contains a set of fuzzy rules to cover the feature space.

The arguments for `genfis2` are as follows:

- `Xin` is a matrix in which each row contains the input values of a data point.
- `Xout` is a matrix in which each row contains the output values of a data point.
- `radii` is a vector that specifies a cluster center's range of influence in each of the data dimensions, assuming the data falls within a unit hyperbox. For example, if the data dimension is 3 (e.g., `Xin` has two columns and `Xout` has one column), `radii` = [0.5 0.4 0.3] specifies that the ranges of influence in the first, second, and third data dimensions (i.e., the first column of `Xin`, the second column of `Xin`, and the column of `Xout`) are 0.5, 0.4, and 0.3 times the width of the data space, respectively. If `radii` is a scalar, then the scalar value is applied to all data dimensions, i.e., each cluster center will have a spherical neighborhood of influence with the given radius.
- `xBounds` is a 2-by-*N* optional matrix that specifies how to map the data in `Xin` and `Xout` into a unit hyperbox, where *N* is the data (row) dimension. The first row of `xBounds` contains the minimum axis range values and the second row contains the maximum axis range values for scaling the data in each dimension. For example, `xBounds` = [-10 0 -1; 10 50 1] specifies that data values in the first data dimension are to be scaled from the range [-10 +10] into values in the range [0 1]; data values in the second data dimension are to be scaled from the range [0 50]; and data values in the third data

dimension are to be scaled from the range [-1 +1]. If `xBounds` is an empty matrix or not provided, then `xBounds` defaults to the minimum and maximum data values found in each data dimension.

- `options` is an optional vector for specifying algorithm parameters to override the default values. These parameters are explained in the help text for `subclust` on page 3-72. Default values are in place when this argument is not specified.

Examples

```
fismat = genfis2(Xin,Xout,0.5)
```

This is the minimum number of arguments needed to use this function. Here a range of influence of 0.5 is specified for all data dimensions.

```
fismat = genfis2(Xin,Xout,[0.5 0.25 0.3])
```

This assumes the combined data dimension is 3. Suppose `Xin` has two columns and `Xout` has one column, then 0.5 and 0.25 are the ranges of influence for each of the `Xin` data dimensions, and 0.3 is the range of influence for the `Xout` data dimension.

```
fismat = genfis2(Xin,Xout,0.5,[-10 -5 0; 10 5 20])
```

This specifies how to normalize the data in `Xin` and `Xout` into values in the range [0 1] for processing. Suppose `Xin` has two columns and `Xout` has one column, then the data in the first column of `Xin` are scaled from [-10 +10], the data in the second column of `Xin` are scaled from [-5 +5], and the data in `Xout` are scaled from [0 20].

See Also

`subclust`

gensurf

Purpose

Generate an FIS output surface.

Synopsis

```
gensurf(fis)
gensurf(fis,inputs,output)
gensurf(fis,inputs,output,grids,refinput)
```

Description

`gensurf(fis)` generates a plot of the output surface of a given fuzzy inference system (`fis`) using the first two inputs and the first output.

`gensurf(fis,inputs,output)` generates a plot using the inputs (one or two) and output (only one is allowed) given, respectively, by the vector, `inputs`, and the scalar, `output`.

`gensurf(fis,inputs,output,grids)` allows you to specify the number of grids in the X (first, horizontal) and Y (second, vertical) directions. If `grids` is a two element vector, the grids in the X and Y directions can be set independently.

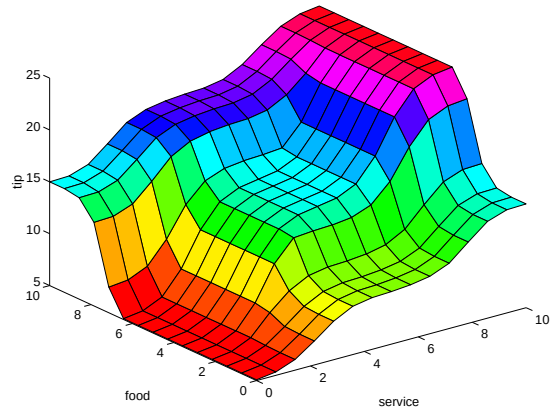
`gensurf(fis,inputs,output,grids,refinput)` can be used if there are more than two outputs. The length of the vector `refinput` is the same as the number of inputs.

- Enter `NaN`s for the entries of `refinput` corresponding to the inputs whose surface is being displayed.
- Enter real double scalars to fix the values of other inputs.

`[x,y,z]=gensurf(...)` returns the variables that define the output surface and suppresses automatic plotting.

Example 1

```
a = readfis('tipper');
gensurf(a)
```



Example 2

```
a = gensurf(Temp,[1 2],1,[20 20],[nan nan 0.2]);
```

generates the surface of a three-input FIS named `Temp` from its first two inputs to its first output, while fixing a reference value for the third input at .2.

See Also

`evalfis`, `surfview`

getfis

Purpose

Get fuzzy system properties.

Synopsis

```
getfis(a)
getfis(a,'fisprop')
getfis(a,'vartype',varindex,'varprop')
getfis(a,'vartype',varindex,'mf',mfindex)
getfis(a,'vartype',varindex,'mf',mfindex,'mfprop')
```

Description

This is the fundamental access function for the FIS structure. With this one function you can learn about every part of the fuzzy inference system.

The arguments for `getfis` are as follows:

- `a`: the name of a workspace variable FIS structure.
- `'vartype'`: a string indicating the type of variable you want (*`input`* or *`output`*).
- `varindex`: an integer indicating the index of the variable you want (1, for input 1, for example).
- `'mf'`: a required string that indicates you are searching for membership function information.
- `mfindex`: the index of the membership function for which you are seeking information.

Examples

One input argument (output is the empty set)

```
a = readfis('tipper');
getfis(a)
        Name = tipper
        Type = mamdani
        NumInputs = 2
        InLabels =
                service
                food
        NumOutputs = 1
        OutLabels =
                tip
        NumRules = 3
        AndMethod = min
        OrMethod = max
        ImpMethod = min
        AggMethod = max
        DefuzzMethod = centroid
```

Two input arguments

```
getfis(a,'type')
ans =
mamdani
```

Three input arguments (output is the empty set)

```
getfis(a,'input',1)
        Name = service
        NumMFs = 3
        MFLabels =
                poor
                good
                excellent
        Range = [0 10]
```

Four input arguments

```
getfis(a,'input',1,'name')
ans =
service
```

Five input arguments

```
getfis(a,'input',1,'mf',2)
        Name = good
        Type = gaussmf
        Params =
    1.5000    5.0000
```

Six input arguments

```
getfis(a,'input',1,'mf',2,'name')
ans =
good
```

See Also

`setfis`, `showfis`

Purpose

Transform Mamdani FIS into a Sugeno FIS.

Synopsis

```
sug_fis=mam2sug(mam_fis)
```

Description

`mam2sug(mam_fis)` transforms a (not necessarily single output) Mamdani FIS structure `mam_fis` into a Sugeno FIS structure `sug_fis`. The returned Sugeno system has constant output membership functions. These constants are determined by the centroids of the consequent membership functions of the original Mamdani system. The antecedent remains unchanged.

Examples

```
mam_fismat = readfis('mam22.fis');
sug_fismat = mam2sug(mam_fismat);
subplot(2,2,1); gensurf(mam_fismat, [1 2], 1);
title('Mamdani system (Output 1)');
subplot(2,2,2); gensurf(sug_fismat, [1 2], 1);
title('Sugeno system (Output 1)');
subplot(2,2,3); gensurf(mam_fismat, [1 2], 2);
title('Mamdani system (Output 2)');
subplot(2,2,4); gensurf(sug_fismat, [1 2], 2);
title('Sugeno system (Output 2)');
```

mf2mf

Purpose

Translates parameters between membership functions.

Synopsis

```
outParams = mf2mf(inParams,inType,outType)
```

Description

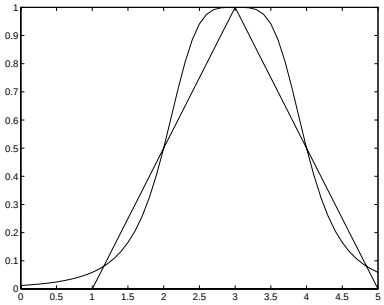
This function translates any built-in membership function type into another, in terms of its parameter set. In principle, `mf2mf` mimics the symmetry points for both the new and old membership functions. Occasionally this translation results in lost information, so that if the output parameters are translated back into the original membership function type, the transformed membership function will not look the same as it did originally.

The input arguments for `mf2mf` are as follows:

- `inParams`: the parameters of the membership function you are transforming
- `inType`: a string name for the type of membership function you are transforming
- `outType`: a string name for the new membership function you are transforming to

Examples

```
x=0:0.1:5;
mfp1 = [1 2 3];
mfp2 = mf2mf(mfp1,'gbellmf','trimf');
plot(x,gbellmf(x,mfp1),x,trimf(x,mfp2))
```



See Also

`dsigmf`, `gaussmf`, `gauss2mf`, `gbellmf`, `evalmf`, `pimf`, `psigmf`, `sigmf`, `smf`, `trapmf`, `trimf`, `zmf`

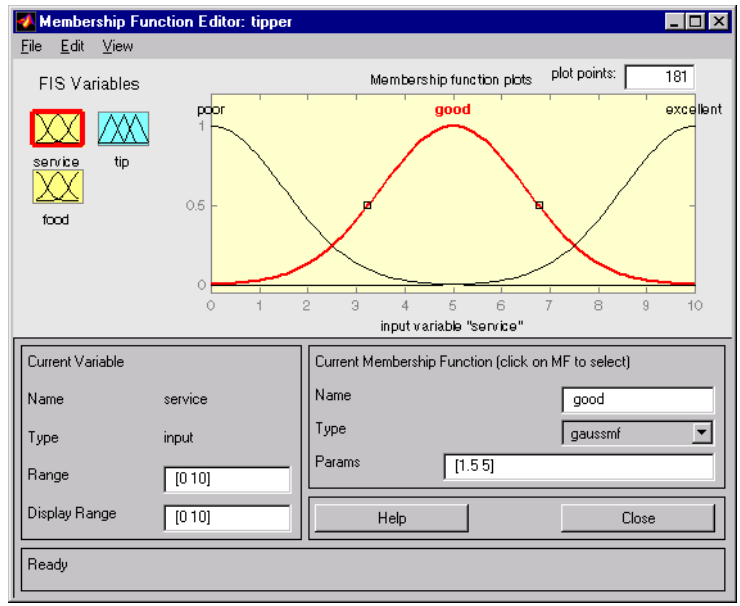
Purpose

Membership function editor.

Synopsis

```
mfedit('a')
mfedit(a)
mfedit
```

Description



`mfedit('a')` generates a membership function editor that allows you to inspect and modify all the membership functions for your FIS stored in the file, `a.fis`.

`mfedit(a)` operates on a MATLAB workspace variable, for a FIS structure, `a`.

`mfedit` alone brings up the membership function editor with no FIS loaded.

For each membership function you can change the name, the type, and the parameters. Eleven built-in membership functions are provided for you to choose from, although of course you can always create your own specialized versions. Refer to “The Membership Function Editor” on page 2-52 for more information about how to use `mfedit`.

Select the icon for the variable on the upper left side of the diagram (under the heading “FIS Variables”) to display its associated membership functions in the plot region. Select membership functions by clicking once on them or their labels.

Menu Items

On the Membership Function Editor, there is a menu bar that allows you to open related GUI tools, open and save systems, and so on. The **File** menu for the Membership Function Editor is the same as the one found on the FIS Editor. Refer to `fuzzy` on page 3-29 for more information.

- Under **Edit**, select:

  Add MF... to add membership functions to the current variable.

  Add custom MF... to add a customized membership function to the current variable.

  Remove current MF to delete the current membership function.

  Remove all MFs to delete all membership functions of the current variable.

  Undo to undo the most recent change.

- Under **View**, select:

  Edit FIS properties... to invoke the FIS Editor.

  Edit rules... to invoke the Rule Editor.

  View rules... to invoke the Rule Viewer.

  View surface... to invoke the Surface Viewer.

Membership Function Pop-up Menu

There are 11 built-in membership functions to choose from, and you also have the option of installing a customized membership function.

See Also

`fuzzy`, `ruleedit`, `ruleview`, `surfview`

Purpose

Create new FIS.

Synopsis

```
a=newfis(fisName,fisType,andMethod,orMethod,impMethod, ...
                          aggMethod,defuzzMethod)
```

Description

This function creates new FIS structures. `newfis` has up to seven input arguments, and the output argument is an FIS structure. The seven input arguments are as follows:

- `fisName` is the string name of the FIS structure, `fisName.fis` you create.
- `fisType` is the type of FIS.
- `andMethod`, `orMethod`, `impMethod`, `aggMethod`, and `defuzzMethod`, respectively provide the methods for AND, OR, implication, aggregation, and defuzzification.

Examples

The following example shows what the defaults are for each of the methods:

```
a=newfis('newsys');
getfis(a)
```

returns

```
          Name = newsys
          Type = mamdani
          NumInputs = 0
          InLabels =
          NumOutputs = 0
          OutLabels =
          NumRules   0
          AndMethod   min
          OrMethod   max
          ImpMethod   min
          AggMethod   max
          DefuzzMethod   centroid
ans =
     [newsys]
```

See Also

`readfis`, `writefis`

parsrule

Purpose

Parse fuzzy rules.

Synopsis

```
fis2 = parsrule(fis,txtRuleList)
fis2 = parsrule(fis,txtRuleList,ruleFormat)
fis2 = parsrule(fis,txtRuleList,ruleFormat,lang)
```

Description

This function parses the text that defines the rules (`txtRuleList`) for a MATLAB workspace FIS variable, `fis`, and returns a FIS structure with the appropriate rule list in place. If the original input FIS structure, `fis`, has any rules initially, they are replaced in the new structure, `fis2`. Three different rule formats (indicated by `ruleFormat`) are supported: `'verbose'`, "symbolic," and "indexed." The default format is "verbose". When the optional language argument, `lang`, is used, the rules are parsed in verbose mode, assuming the key words are in the language, `lang`. This language must be either 'english', 'francais', or 'deutsch'. The key language words in English are: if, then, is, AND, OR, and NOT.

Examples

```
a = readfis('tipper');
ruleTxt = 'if service is poor then tip is generous';
a2 = parsrule(a,ruleTxt,'verbose');
showrule(a2)
ans =
   1. If (service is poor) then (tip is generous) (1)
```

See Also

`addrule`, `ruleedit`, `showrule`

Purpose

Π-shaped built-in membership function.

Synopsis

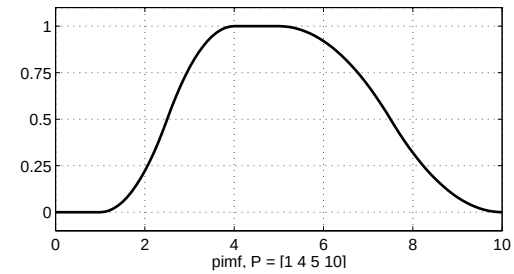
```
y = pimf(x,[a b c d])
```

Description

This spline-based curve is so named because of its Π-shape. This membership function is evaluated at the points determined by the vector x. The parameters *a* and *d* locate the "feet" of the curve, while *b* and *c* locate its "shoulders."

Examples

```
x=0:0.1:10;
y=pimf(x,[1 4 5 10]);
plot(x,y)
xlabel('pimf, P=[1 4 5 10]')
```



See Also

dsigmf, gaussmf, gauss2mf, gbellmf, evalmf, mf2mf, psigmf, sigmf, smf, trapmf, trimf, zmf

plotfis

Purpose

Plot an FIS.

Synopsis

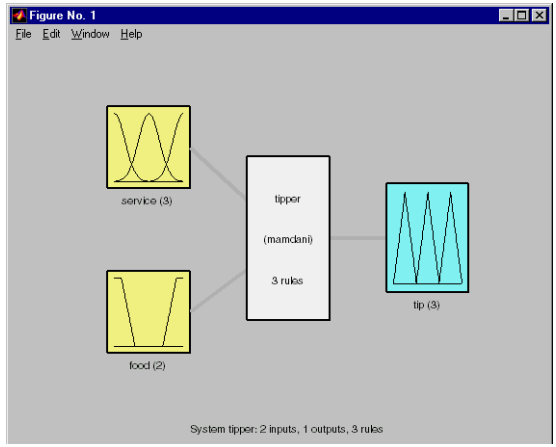
```
plotfis(fismat)
```

Description

This function displays a high level diagram of an FIS, `fismat`. Inputs and their membership functions appear to the left of the FIS structural characteristics, while outputs and their membership functions appear on the right.

Examples

```
a = readfis('tipper');
plotfis(a)
```



See Also

`evalmf`, `plotmf`

Purpose

Plot all of the membership functions for a given variable.

Synopsis

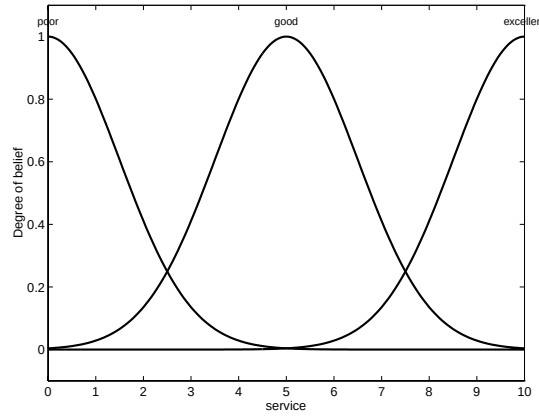
```
plotmf(fismat,varType,varIndex)
```

Description

This function plots all of the membership functions in the FIS called `fismat` associated with a given variable whose type and index are respectively given by `varType` (`'input'` or `'output'`), and `varIndex`. This function can also be used with the MATLAB function, `subplot`.

Examples

```
a = readfis('tipper');
plotmf(a,'input',1)
```



See Also

`evalmf`, `plotfis`

psigmf

Purpose

Built-in membership function composed of the product of two sigmoidally-shaped membership functions.

Synopsis

```
y = psigmf(x,[a1 c1 a2 c2])
```

Description

The sigmoid curve plotted for the vector `x` depends on two parameters *a* and *c* as given by

$$f(x;a,c) = \frac{1}{1+e^{-a(x-c)}}$$

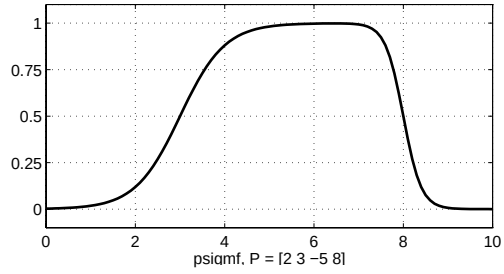
`psigmf` is simply the product of two such curves plotted for the values of the vector `x`

$f_1(x; a1, c_1) * f_2(x; a2, c_2)$

The parameters are listed in the order: $[a1\ c_1\ a_2\ c_2]$.

Examples

```
x=0:0.1:10;
y=psigmf(x,[2 3 -5 8]);
plot(x,y)
xlabel('psigmf, P=[2 3 -5 8]')
```



See Also

`dsigmf`, `gaussmf`, `gauss2mf`, `gbellmf`, `evalmf`, `mf2mf`, `pimf`, `sigmf`, `smf`, `trapmf`, `trimf`, `zmf`

Purpose

Load an FIS from the disk.

Synopsis

```
fismat = readfis('filename')
```

Description

Read a fuzzy inference system from a `.fis` file (named *`filename`*) on the disk and bring the resulting file into the workspace.

`fismat = readfis` (no input arguments) brings up a `uigetfile` dialog box to assist with the name and directory location of the file.

Examples

```
fismat = readfis('tipper');
getfis(fismat)
```

returns

```
    Name = tipper
    Type = mamdani
    NumInputs = 2
    InLabels =
        service
        food
    NumOutputs = 1
    OutLabels =
        tip
    NumRules = 3
    AndMethod = min
    OrMethod = max
    ImpMethod = min
    AggMethod = max
    DefuzzMethod = centroid
ans =
tipper
```

See Also

`writefis`

rmmf

Purpose

To remove membership function from an FIS.

Synopsis

```
fis = rmmf(fis,'varType',varIndex,'mf',mfIndex)
```

Description

`fis = rmmf(fis,varType,varIndex,'mf',mfIndex)` removes the membership function, `mfIndex`, of variable type, `varType`, of index `varIndex`, from the fuzzy inference system associated with the workspace FIS structure, `fis`:

- The string `varType` must be `'input'` or `'output'`.
- `varIndex` is an integer for the index of the variable. This index represents the order in which the variables are listed.
- The argument `'mf'` is a string representing the membership function.
- `mfIndex` is an integer for the index of the membership function. This index represents the order in which the membership functions are listed.

Examples

```
a = newfis('mysys');
a = addvar(a,'input','temperature',[0 100]);
a = addmf(a,'input',1,'cold','trimf',[0 30 60]);
getfis(a,'input',1)
```

returns

```
          Name = temperature
          NumMFs = 1
          MFLabels =
                  cold
          Range = [0 100]
```

```
b = rmmf(a,'input',1,'mf',1);
getfis(b,'input',1)
```

returns

```
          Name = temperature
          NumMFs = 0
          MFLabels =
          Range = [0 100]
```

See Also

`addmf`, `addrule`, `addvar`, `plotmf`, `rmvar`

Purpose

To remove variables from an FIS.

Synopsis

```
[fis2,errorStr] = rmvar(fis,'varType',varIndex)
fis2 = rmvar(fis,'varType',varIndex)
```

Description

`fis2 = rmvar(fis,'varType',varIndex)` removes the variable `'varType'`, of index `varIndex`, from the fuzzy inference system associated with the workspace FIS structure, `fis`:

- The string `varType` must be `'input'` or `'output'`.
- `varIndex` is an integer for the index of the variable. This index represents the order in which the variables are listed.

`[fis2,errorStr] = rmvar(fis,'varType',varIndex)` returns any error messages to the string, `errorStr`.

This command automatically alters the rule list to keep its size consistent with the current number of variables. You must delete from the FIS any rule that contains a variable you want to remove, before removing it. You cannot remove a fuzzy variable currently in use in the rule list.

Examples

```
a = newfis('mysys');
a = addvar(a,'input','temperature',[0 100]);
getfis(a)
```

```
returns

      Name = mysys
         Type      = mamdani
         NumInputs = 1
         InLabels  =
               temperature
         NumOutputs = 0
         OutLabels =
         NumRules = 0
         AndMethod = min
         OrMethod = max
         ImpMethod = min
         AggMethod = max
         DefuzzMethod = centroid
   ans =
   mysys

   b = rmvar(a,'input',1);
   getfis(b)
returns

           Name = mysys
           Type = mamdani
           NumInputs = 0
           InLabels =
           NumOutputs = 0
           OutLabels =
           NumRules = 0
         AndMethod = min
         OrMethod = max
         ImpMethod = min
         AggMethod = max
         DefuzzMethod = centroid
   ans =
   mysys
```

See Also

addmf, addrule, addvar, rmmf

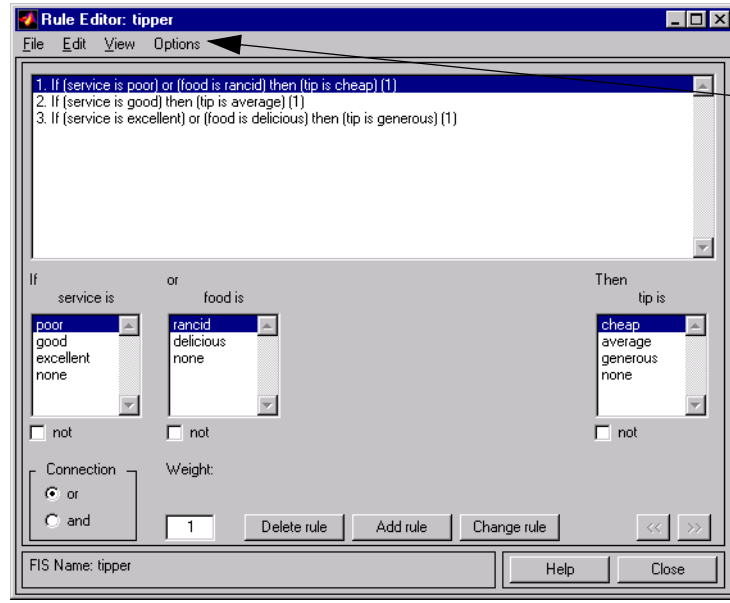
Purpose

Rule editor and parser.

Synopsis

```
ruleedit('a')
ruleedit(a)
```

Description



Language and Format are options on this GUI. Languages are English, Deutsch, and Francais. Formats are verbose, symbolic, and indexed.

The Rule Editor, when invoked using `ruleedit('a')`, is used to modify the rules of an FIS structure stored in a file, `a.fis`. It can also be used to inspect the rules being used by a fuzzy inference system.

To use this editor to create rules, you must first define all of the input and output variables you want to use with the FIS editor. You can create the rules using the listbox and check box choices for input and output variables, connections, and weights. Refer to “The Rule Editor” on page 2-56 for more information about how to use `ruleedit`.

The syntax `ruleedit(a)` is used when you want to operate on a workspace variable for an FIS structure called `a`.

Menu Items

On the Rule Editor, there is a menu bar that allows you to open related GUI tools, open and save systems, and so on. The **File** menu for the Rule Editor is the same as the one found on the FIS Editor. Refer to `fuzzy` on page 3-29 for more information.

- Use the following **Edit** menu item:

  Undo to undo the most recent change.

- Use the following **View** menu items:

  Edit FIS properties... to invoke the FIS Editor.

  Edit membership functions... to invoke the Membership Function Editor.

  View rules... to invoke the Rule Viewer.

  View surface... to invoke the Surface Viewer.

- Use the **Options** menu items:

  Language to select the language: **English**, **Deutsch**, and **Francais**

  Format to select the format:

  - **verbose** uses the words "if," "then," "AND," "OR," and so on to create actual sentences.
  - **symbolic** substitutes some symbols for the words used in the verbose mode. For example, "if *A* AND *B* then *C*" becomes "*A* & *B* => *C*."
  - **indexed** mirrors how the rule is stored in the FIS structure.

See Also

`addrule`, `fuzzy`, `mfedit`, `parsrule`, `ruleview`, `showrule`, `surfview`

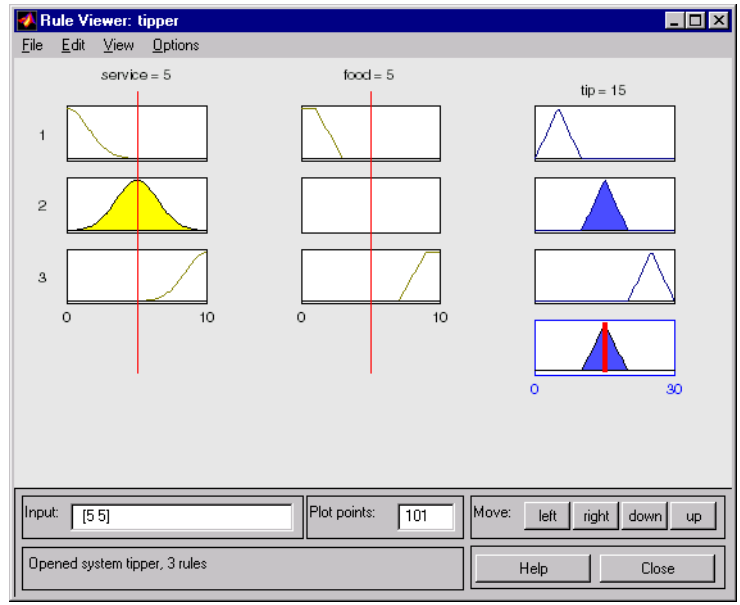
Purpose

Rule viewer and fuzzy inference diagram.

Synopsis

```
ruleview('a')
```

Description



The Rule Viewer invoked using `ruleview('a')` depicts the fuzzy inference diagram for an FIS stored in a file, `a.fis`. It is used to view the entire implication process from beginning to end. You can move around the line indices that correspond to the inputs and then watch the system readjust and compute the new output. Refer to “The Rule Viewer” on page 2-59 for more information about how to use `ruleview`.

Menu Items

On the Rule Viewer, there is a menu bar that allows you to open related GUI tools, open and save systems, and so on. The **File** menu for the Rule Viewer is

the same as the one found on the FIS Editor. Refer to `fuzzy` on page 3-29 for more information.

- Use the **View** menu items:

  Edit FIS properties... to invoke the FIS Editor

  Edit membership functions... to invoke the Membership Function Editor

  Edit rules... to invoke the Rule Editor

  View surface... to invoke the Surface Viewer

- Use the **Options** menu item:

  Rule display format to set the format in which the rule appears. If you click on the rule numbers on the left side of the fuzzy inference diagram, the rule associated with that number will appear in the Status Bar at the bottom of the Rule Viewer.

See Also

`fuzzy`, `mfedit`, `ruleedit`, `surfview`

Purpose

Set fuzzy system properties.

Synopsis

```
a = setfis(a,'fispropname','newfisprop')
a = setfis(a,'vartype',varindex,'varpropname','newvarprop')
a = setfis(a,'vartype',varindex,'mf',mfindex, ...
                          'mfpropname','newmfprop');
```

Description

The command `setfis` can be called with three, five, or seven input arguments, depending on whether you want to set a property of the entire FIS structure, for a particular variable belonging to that FIS structure, or for a particular membership function belonging to one of those variables. The arguments are:

- `a` – a variable name of an FIS from the workspace
- `'vartype'` – a string indicating the variable type: *input* or *output*
- `varindex` – the index of the input or output variable
- `'mf'` – a required string for the fourth argument of a 7-argument call for `setfis`, indicating this variable is a membership function
- `mfindex` – the index of the membership function belonging to the chosen variable
- `'fispropname'` – a string indicating the property of the FIS field you want to set: *name*, *type*, *andmethod*, *ormethod*, *impmethod*, *aggmethod*, *defuzzmethod*
- `'newfisprop'` – a string describing the name of the FIS property or method you want to set
- `'varpropname'` – a string indicating the name of the variable field you want to set: *name* or *range*
- `'newvarprop'` – a string describing the name of the variable you want to set (for *name*), or an array describing the range of that variable (for *range*)
- `'mfpropname'`– a string indicating the name of the membership function field you want to set: *name*, *type*, or *params*.
- `'newmfprop'` – a string describing the name or type of the membership function field want to set (for *name* or *type*), or an array describing the range of the parameters (for *params*)

Examples

Called with three arguments,

```
a = readfis('tipper');
a2 = setfis(a, 'name', 'eating');
getfis(a2, 'name');
```

Results in

```
out =
eating
```

If used with five arguments, `setfis` will update two variable properties.

```
a2 = setfis(a,'input',1,'name','help');
getfis(a2,'input',1,'name')
ans =
   help
```

If used with seven arguments, `setfis` will update any of several membership function properties.

```
a2 = setfis(a,'input',1,'mf',2,'name','wretched');
getfis(a2,'input',1,'mf',2,'name')
ans =
   wretched
```

See Also

`getfis`

Purpose

Fuzzy inference S-function for Simulink.

Synopsis

```
output = sffis(t,x,u,flag,fismat)
```

Description

This MEX-file is used by Simulink to undertake the calculation normally performed by `evalfis`. It has been optimized to work in the Simulink environment. This means, among other things, that `sffis` builds a data structure in memory during the initialization phase of a Simulink simulation, which it then continues to use until the simulation is complete.

The arguments `t`, `x`, and `flag` are standard Simulink S-function arguments (see Chapter 8, "S-Functions" in the *Using Simulink* documentation). The argument `u` is the input to the MATLAB workspace FIS structure, `fismat`. If, for example, there are two inputs to `fismat`, then `u` will be a two-element vector.

See Also

`evalfis`, `fuzblock`

showfis

Purpose

Display annotated FIS.

Synopsis

```
showfis(fismat)
```

Description

`showfis(fismat)` prints a version of the MATLAB workspace variable FIS, `fismat`, allowing you to see the significance and contents of each field of the structure.

Examples

```
a = readfis('tipper');
showfis(a)
```

returns

```
1.  Name             tipper
2.  Type             mamdani
3.  Inputs/Outputs   [2 1]
4.  NumInputMFs      [3 2]
5.  NumOutputMFs     3
6.  NumRules         3
7.  AndMethod        min
8.  OrMethod         max
9.  ImpMethod        min
10. AggMethod        max
11. DefuzzMethod     centroid
12. InLabels         service
13.                  food
14. OutLabels        tip
15. InRange          [0 10]
16.                  [0 10]
17. OutRange         [0 30]
18. InMFLabels       poor
19.                  good
20.                  excellent
21.                  rancid
22.                  delicious
23. OutMFLabels      cheap
24.                  average
25.                  generous
26. InMFTypes        gaussmf
27.                  gaussmf
```

```
28.                  gaussmf
29.                  trapmf
30.                  trapmf
31. OutMFTypes       trimf
32.                  trimf
33.                  trimf
34. InMFParams       [1.5 0 0 0]
35.                  [1.5 5 0 0]
36.                  [1.5 10 0 0]
37.                  [0 0 1 3]
38.                  [7 9 10 10]
39. OutMFParams      [0 5 10 0]
40.                  [10 15 20 0]
41.                  [20 25 30 0]
42. Rule Antecedent  [1 1]
43.                  [2 0]
44.                  [3 2]
42. Rule Consequent  1
43.                  2
44.                  3
42. Rule Weigth      1
43.                  1
44.                  1
42. Rule Connection  2
43.                  1
44.                  2
```

See Also

getfis

showrule

Purpose

Display FIS rules.

Synopsis

```
showrule(fis)
showrule(fis,indexList)
showrule(fis,indexList,format)
showrule(fis,indexList,format,Lang)
```

Description

This command is used to display the rules associated with a given system. It can be invoked with one to four arguments. The first argument, `fis`, is required. This is the MATLAB workspace variable name for a FIS structure. The second (optional) argument `indexList` is the vector of rules you want to display. The third argument (optional) is the string representing the format in which the rules are returned. `showrule` can return the rule in any of three different formats: '*verbose*' (the default mode, for which English is the default language), '*symbolic*', and '*indexed*', for membership function index referencing.

When used with four arguments, the forth argument must be verbose, and `showrule(fis,indexList,format,lang)` displays the rules in the language given by `lang`, which must be either '*english*', '*francais*', or '*deutsch*'.

Examples

```
a = readfis('tipper');
showrule(a,1)
ans =
1. If (service is poor) or (food is rancid) then (tip is cheap) (1)

showrule(a,2)
ans =
2. If (service is good) then (tip is average) (1)

showrule(a,[3 1],'symbolic')
ans =
3. (service==excellent) | (food==delicious) => (tip=generous) (1)
1. (service==poor) | (food==rancid) => (tip=cheap) (1)

showrule(a,1:3,'indexed')
ans =
1 1, 1 (1) : 2
2 0, 2 (1) : 1
3 2, 3 (1) : 2
```

See Also

`parsrule`, `ruleedit`, `addrule`

sigmf

Purpose

Sigmoidally-shaped built-in membership function.

Synopsis

```
y = sigmf(x,[a c])
```

Description

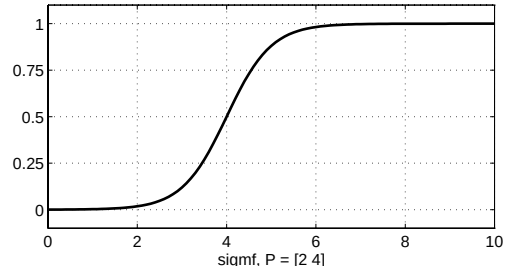
The sigmoidal function, `sigmf(x,[a c])`, as given below by $f(x, a, c)$, is a mapping on a vector `x`, and depends on two parameters a and c:

$$f(x, a, c) = \frac{1}{1 + e^{-a(x-c)}}$$

Depending on the sign of the parameter a, the sigmoidal membership function is inherently open to the right or to the left, and thus is appropriate for representing concepts such as "very large" or "very negative." More conventional-looking membership functions can be built by taking either the product or difference of two different sigmoidal membership functions. You can find out more about this in this chapter's entries for `dsigmf` and `psigmf`.

Examples

```
x=0:0.1:10;
y=sigmf(x,[2 4]);
plot(x,y)
xlabel('sigmf, P=[2 4]')
```



See Also

`dsigmf`, `gaussmf`, `gauss2mf`, `gbellmf`, `evalmf`, `mf2mf`, `pimf`, `psigmf`, `smf`, `trapmf`, `trimf`, `zmf`

Purpose

S–shaped built-in membership function.

Synopsis

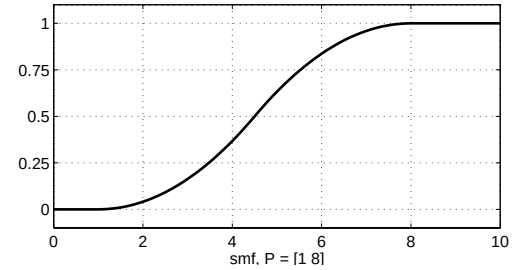
```
y = smf(x,[a b])
```

Description

This spline-based curve is a mapping on the vector `x`, and is named because of its S–shape. The parameters *a* and *b* locate the extremes of the sloped portion of the curve.

Examples

```
x=0:0.1:10;
y=smf(x,[1 8]);
plot(x,y)
xlabel('smf, P=[1 8]')
```



See Also

`dsigmf`, `gaussmf`, `gauss2mf`, `gbellmf`, `evalmf`, `mf2mf`, `pimf`, `psigmf`, `sigmf`, `trapmf`, `trimf`, `zmf`

subclust

Purpose

Find cluster centers with subtractive clustering.

Synopsis

```
[C,S] = subclust(X,radii,xBounds,options)
```

Description

This function estimates the cluster centers in a set of data by using the subtractive clustering method. The subtractive clustering method assumes each data point is a potential cluster center and calculates a measure of the likelihood that each data point would define the cluster center, based on the density of surrounding data points. The algorithm

- Selects the data point with the highest potential to be the first cluster center
- Removes all data points in the vicinity of the first cluster center (as determined by `radii`), in order to determine the next data cluster and its center location
- Iterates on this process until all of the data is within `radii` of a cluster center

The subtractive clustering method is an extension of the mountain clustering method proposed by R. Yager.

The matrix `X` contains the data to be clustered; each row of `X` is a data point. The variable `radii` is a vector of entries between 0 and 1 that specifies a cluster center's range of influence in each of the data dimensions, assuming the data falls within a unit hyperbox. Small `radii` values generally result in finding a few large clusters. Good values for `radii` are usually between 0.2 and 0.5.

For example, if the data dimension is two (`X` has two columns), `radii` = [0.5 0.25] specifies that the range of influence in the first data dimension is half the width of the data space and the range of influence in the second data dimension is one quarter the width of the data space. If `radii` is a scalar, then the scalar value is applied to all data dimensions, i.e., each cluster center will have a spherical neighborhood of influence with the given radius.

`xBounds` is a 2-by-N matrix that specifies how to map the data in `X` into a unit hyperbox, where N is the data dimension. This argument is optional if `X` is already normalized. The first row contains the minimum axis range values and the second row contains the maximum axis range values for scaling the data in each dimension. For example, `xBounds` = [-10 -5; 10 5] specifies that data values in the first data dimension are to be scaled from the range [-10 +10] into values in the range [0 1]; data values in the second data dimension are to be scaled from the range [-5 +5] into values in the range [0 1]. If `xBounds` is an empty

matrix or not provided, then `xBounds` defaults to the minimum and maximum data values found in each data dimension.

The `options` vector can be used for specifying clustering algorithm parameters to override the default values. These components of the vector `options` are specified as follows:

- `options(1) = quashFactor`: This is the factor used to multiply the radii values that determine the neighborhood of a cluster center, so as to quash the potential for outlying points to be considered as part of that cluster. (default: 1.25)
- `options(2) = acceptRatio`: This sets the potential, as a fraction of the potential of the first cluster center, above which another data point will be accepted as a cluster center. (default: 0.5)
- `options(3) = rejectRatio`: This sets the potential, as a fraction of the potential of the first cluster center, below which a data point will be rejected as a cluster center. (default: 0.15)
- `options(4) = verbose`: If this term is not zero, then progress information will be printed as the clustering process proceeds. (default: 0)

The function returns the cluster centers in the matrix `C`; each row of `C` contains the position of a cluster center. The returned `S` vector contains the sigma values that specify the range of influence of a cluster center in each of the data dimensions. All cluster centers share the same set of sigma values.

Examples

```
[C,S] = subclust(X,0.5)
```

This is the minimum number of arguments needed to use this function. A range of influence of 0.5 has been specified for all data dimensions.

```
[C,S] = subclust(X,[0.5 0.25 0.3],[],[2.0 0.8 0.7])
```

This assumes the data dimension is 3 (`X` has 3 columns) and uses a range of influence of 0.5, 0.25, and 0.3 for the first, second and third data dimension, respectively. The scaling factors for mapping the data into a unit hyperbox will be obtained from the minimum and maximum data values. The `squashFactor` is set to 2.0, indicating that we only want to find clusters that are far from each other. The `acceptRatio` is set to 0.8, indicating that we will only accept data points that have a very strong potential for being cluster centers. The

`rejectRatio` is set to 0.7, indicating that we want to reject all data points without a strong potential.

See Also

`genfis2`

References

Chiu, S., "Fuzzy Model Identification Based on Cluster Estimation," *Journal of Intelligent & Fuzzy Systems*, Vol. 2, No. 3, Sept. 1994.

Yager, R. and D. Filev, "Generation of Fuzzy Rules by Mountain Clustering," *Journal of Intelligent & Fuzzy Systems*, Vol. 2, No. 3, pp. 209-219, 1994.

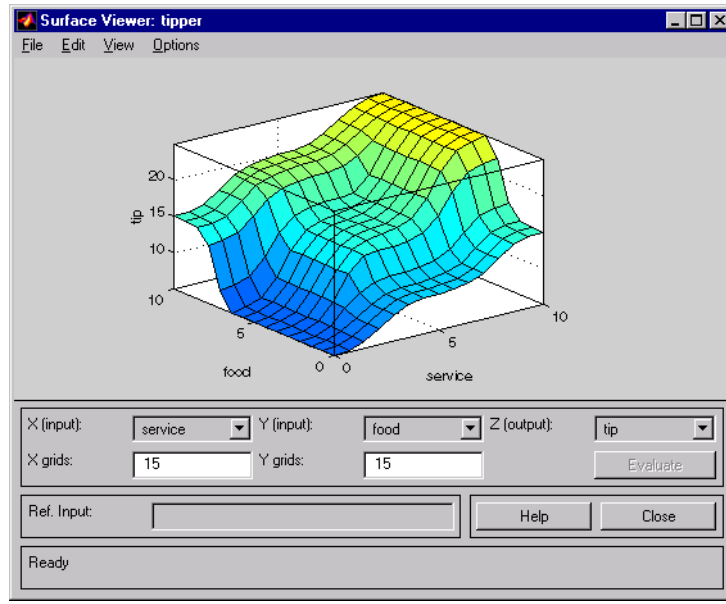
Purpose

Output surface viewer.

Synopsis

```
surfview('a')
```

Description



The Surface Viewer invoked using `surfview('a')` is a GUI tool that lets you examine the output surface of an FIS stored in a file, `a.fis`, for any one or two inputs. Since it does not alter the fuzzy system or its associated FIS structure in any way, it is a read-only editor. Using the pop-up menus, you select the two input variables you want assigned to the two input axes (X and Y), as well the output variable you want assigned to the output (or Z) axis. Select the **Evaluate** button to perform the calculation and plot the output surface.

By clicking on the plot axes and dragging the mouse, you can manipulate the surface so that you can view it from different angles.

If there are more than two inputs to your system, you must supply the constant values associated with any unspecified inputs in the reference input section.

Refer to “The Surface Viewer” on page 2-61 for more information about how to use `surfview`.

Menu Items

On the Surface Viewer, there is a menu bar that allows you to open related GUI tools, open and save systems, and so on. The **File** menu for the Surface Viewer is the same as the one found on the FIS Editor. Refer to `fuzzy` on page 3-29 for more information.

- Use the **View** menu items:

  Edit FIS properties... to invoke the FIS Editor.

  Edit membership functions... to invoke the Membership Function Editor.

  Edit rules... to invoke the Rule Editor.

  View rules... to invoke the Rule Viewer.

- Use the **Options** menu items:

  Plot to choose among eight different kinds of plot styles.

  Color Map to choose among several different color schemes.

  Always evaluate to automatically evaluate and plot a new surface every time you make a change that affects the plot (like changing the number of grid points). This is the default option. To deselect this option, select it once more.

See Also

`anfisedit`, `fuzzy`, `gensurf`, `mfedit`, `ruleedit`, `ruleview`

Purpose

Trapezoidal-shaped built-in membership function.

Synopsis

```
y = trapmf(x,[a b c d])
```

Description

The trapezoidal curve is a function of a vector, `x`, and depends on four scalar parameters *a*, *b*, *c*, and *d*, as given by

$$f(x;a,b,c,d) = \begin{Bmatrix} 0, & x \le a \\ \frac{x-a}{b-a}, & a \le x \le b \\ 1, & b \le x \le c \\ \frac{d-x}{d-c}, & c \le x \le d \\ 0, & d \le x \end{Bmatrix}$$

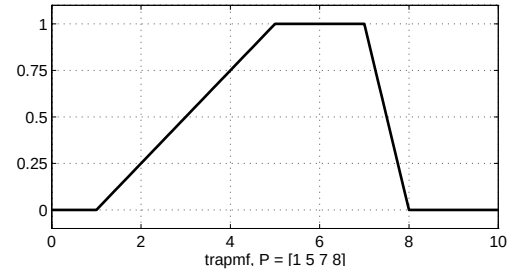
or, more compactly, by

$$f(x;a,b,c,d) = \max\left(\min\left(\frac{x-a}{b-a}, 1, \frac{d-x}{d-c}\right), 0\right)$$

The parameters *a* and *d* locate the "feet" of the trapezoid and the parameters *b* and *c* locate the "shoulders."

Examples

```
x=0:0.1:10;
y=trapmf(x,[1 5 7 8]);
plot(x,y)
xlabel('trapmf, P=[1 5 7 8]')
```



See Also

dsigmf, gaussmf, gauss2mf, gbellmf, evalmf, mf2mf, pimf, psigmf, sigmf, smf, trimf, zmf

Purpose

Triangular-shaped built-in membership function.

Synopsis

```
y = trimf(x,params)
y = trimf(x,[a b c])
```

Description

The triangular curve is a function of a vector, x, and depends on three scalar parameters *a*, *b*, and *c*, as given by

$$f(x;a,b,c) = \begin{cases} 0, & x \le a \\ \dfrac{x-a}{b-a}, & a \le x \le b \\ \dfrac{c-x}{c-b}, & b \le x \le c \\ 0, & c \le x \end{cases}$$

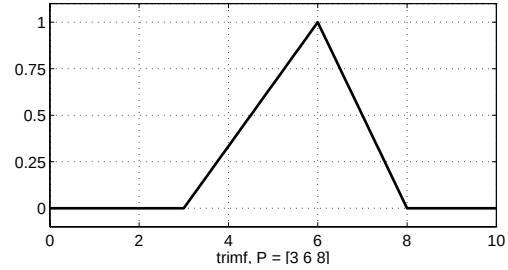
or, more compactly, by

$$f(x;a,b,c) = \max\left(\min\left(\frac{x-a}{b-a}, \frac{c-x}{c-b}\right), 0\right)$$

The parameters *a* and *c* locate the "feet" of the triangle and the parameter *c* locates the peak.

Examples

```
x=0:0.1:10;
y=trimf(x,[3 6 8]);
plot(x,y)
xlabel('trimf, P=[3 6 8]')
```



See Also

dsigmf, gaussmf, gauss2mf, gbellmf, evalmf, mf2mf, pimf, psigmf, sigmf, smf, trapmf

Purpose

Save an FIS to the disk.

Synopsis

```
writefis(fismat)
writefis(fismat,'filename')
writefis(fismat,'filename','dialog'
```

Description

`writefis` saves a MATLAB workspace FIS structure, `fismat`, as a `.fis` file on disk.

`writefis(fismat)` brings up a dialog box to assist with the naming and directory location of the file.

`writefis(fismat,'filename')` writes a `.fis` file corresponding to the FIS structure, `fismat`, to a disk file called `filename.fis`. No dialog box is used and the file is saved to the current directory.

`writefis(fismat,'filename','dialog')` brings up a dialog box with the default name `filename.fis` supplied.

The extension `.fis` is only added to *`filename`* if it is not already included in the name.

Examples

```
a = newfis('tipper');
a = addvar(a,'input','service',[0 10]);
a = addmf(a,'input',1,'poor','gaussmf',[1.5 0]);
a = addmf(a,'input',1,'good','gaussmf',[1.5 5]);
a = addmf(a,'input',1,'excellent','gaussmf',[1.5 10]);
writefis(a,'my_file')
```

See Also

`readfis`

zmf

Purpose

Z-shaped built-in membership function.

Synopsis

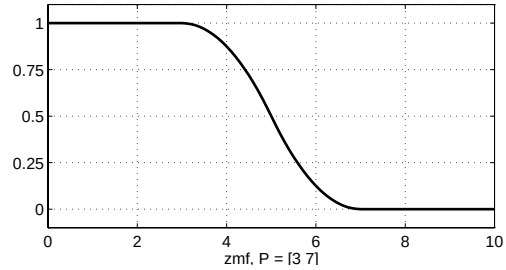
```
y = zmf(x,[a b])
```

Description

This spline-based function of x is so named because of its Z-shape. The parameters *a* and *b* locate the extremes of the sloped portion of the curve.

Examples

```
x=0:0.1:10;
y=zmf(x,[3 7]);
plot(x,y)
xlabel('zmf, P=[3 7]')
```



See Also

dsigmf, gaussmf, gauss2mf, gbellmf, evalmf, mf2mf, pimf, psigmf, sigmf, smf, trapmf, trimf

Index

G

H

I

L

M

N

O

P

R

S

T

W

Z