MATLAB®

Fuzzy Logic Toolbox

J.-S. Roger Jang
Ned Gulley

Computation

Visualization

Programming

User's Guide

Version 1

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 (for MATLAB 5)

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 as used by 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.”

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 problem that has to do with tipping. If you like you can refer to the one page summary of the fuzzy inference process shown at the beginning of Chapter 3, “Reference.”

All toolbox users should use Chapter 3, “Reference,” for information on specific tools. 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 names `fuzzy` within the main toolbox directory or folder.

Typographical Conventions

To Indicate	This Guide Uses	Example
Example code	`Monospace type`	To assign the value 5 to `A`, enter `A = 5`

MATLAB output	`Monospace type`	MATLAB responds with
		`A =`
		`5`

Function names	`Monospace type`	The cos function finds the cosine of each array element.
New terms	*Italics*	An *array* is an ordered collection of information.
Keys	**Boldface** with an initial capital letter	Press the **Return** key.
Menu names, items, and GUI controls	**Boldface** with an initial capital letter	Chose the **File** menu.
Mathematical expressions	Variables in *italics*. Functions, operators, and constants in standard type.	This vector represents the polynomial $p = x^2+2x+3$.

1

Introduction

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

There are also some pearls of folk wisdom that echo these thoughts:

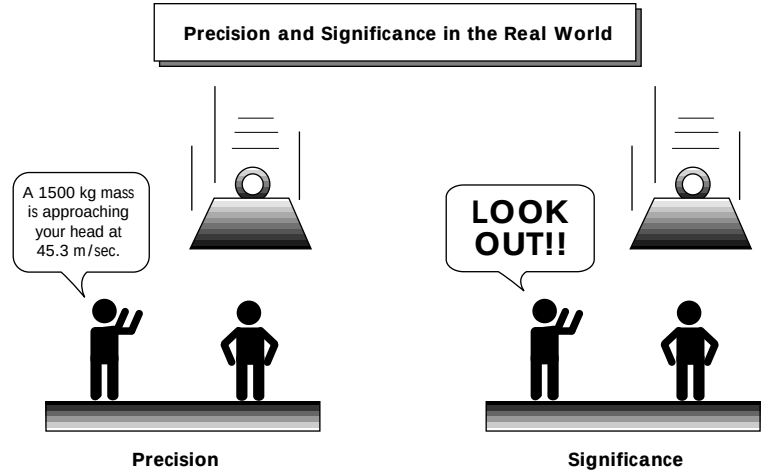
Don't lose sight of the forest for the trees.

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,

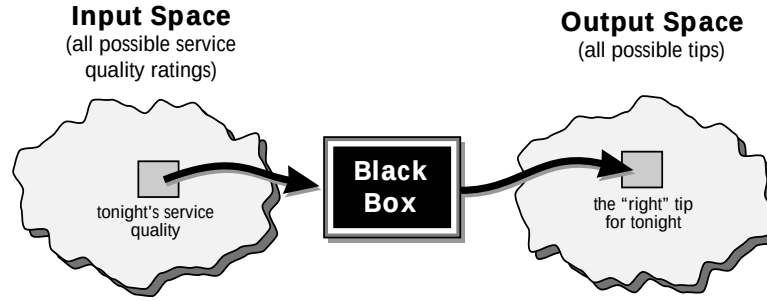
although the modern and methodical science of fuzzy logic is still young, the concepts of fuzzy logic reach right down to our bones.



What Is Fuzzy Logic?

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 monkeys with typewriters 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 stand on the shoulders 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.

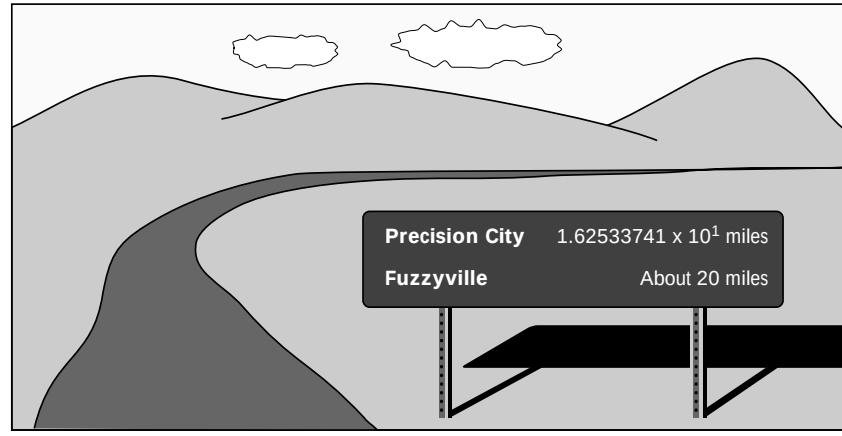
- Fuzzy logic is based on natural language.

  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. But since fuzzy logic is built

atop the structures of everyday language, it not only makes it easy for us to use it (since fuzzy logic more closely "speaks our language") but it also takes advantage of the long history of natural language. In other words, language is a fuzzy logic tool the human race has spent a hundred generations developing.

Clear language is about getting at the big picture. Fuzzy logic keeps you from bogging down in unnecessary detail. It's all a matter of perspective. Life is complicated enough already.



When Not to Use Fuzzy Logic

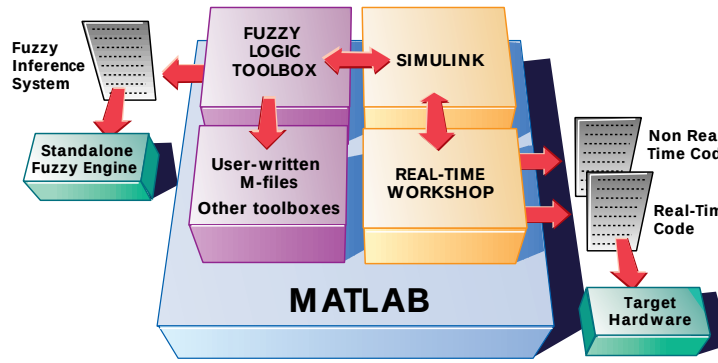
Fuzzy logic is not a cure-all. When should you not use fuzzy logic? The safest statement is the first one made in this introduction: fuzzy logic is a convenient way to map an input space to an output space. If you find it's not convenient, try something else. If a simpler solution already exists, use 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 being tweaked by fuzzy logic. But 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. Nonlinearity is everywhere, and if you don't go and find it, it will eventually come and find you.

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 by hand, 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®, the simulation tool that runs alongside MATLAB, you can easily test your fuzzy system in a block diagram simulation environment. If you have Real-Time Workshop® capabilities available, you can generate realtime or non-realtime code from the Simulink 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 (the stand-alone code, unlike that generated by the Real-Time Workshop, does not run in real time). 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.

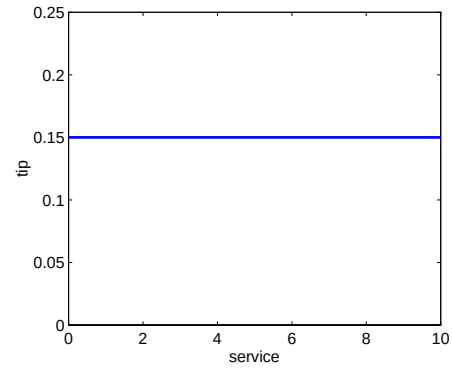
The Basic Tipping Problem. Given a number between 0 and 10 that represents the quality of service at a restaurant (where 10 is excellent), what should the tip be?

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

So let's start with the simplest possible relationship. We can say that the tip always equals 15% of the total bill. So

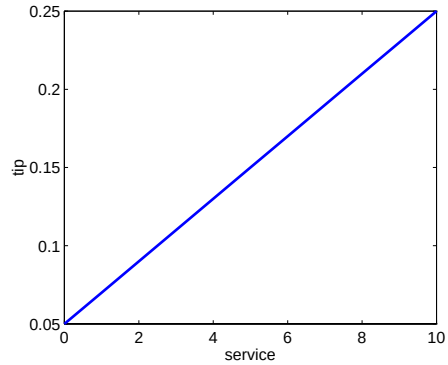
```
tip = 0.15
```



But 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 zero to

ten, then 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
```

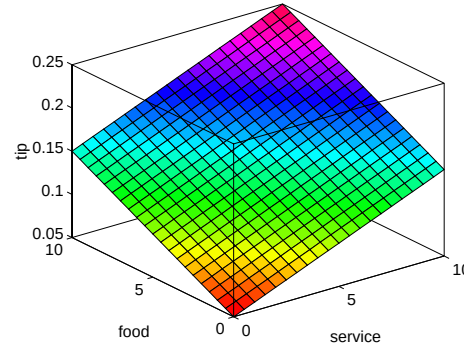


So far so good. The formula does what we want it to do, and it's pretty straightforward. But we may want the tip to reflect the quality of the food as well. This extension of the problem is defined as follows:

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

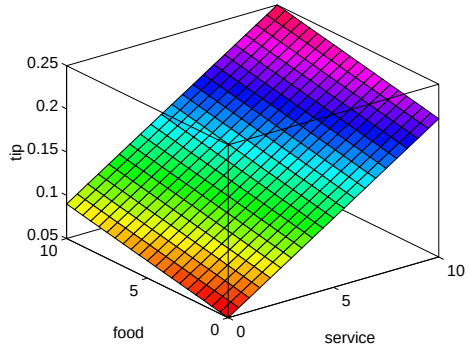
How will our formula be affected now that we've added another variable? Here's one attempt:

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



Well, that's one way to do it, and the picture is pretty, but when I look at it closely, it doesn't seem quite right. I want the service to be a more important factor than the food quality. Let's say that I want the service to account for 80% of the overall tipping "grade" and I'll let the food make up the other 20%. So let me try:

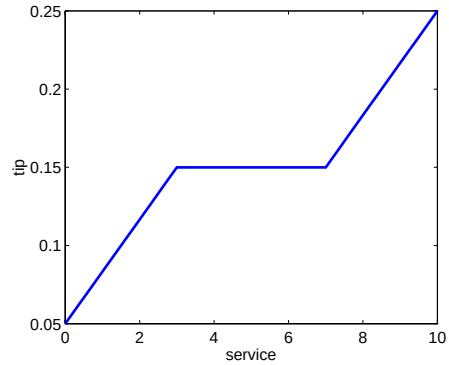
```
servRatio=0.8;
tip=servRatio*(0.20/10*service+0.05) + ...
    (1-servRatio)*(0.20/10*food+0.05);
```



But still the response is somehow too linear all the way around. I want more of a flat response in the middle; in other words, I want to give a 15% tip in general, and I will depart from this plateau only if the service is exceptionally good or bad. This, in turn, means my pleasant linear relations go out the window. But we can still salvage things by using a piecewise linear construction. Let's return to the one-dimensional problem of just considering the service. I can string together a simple conditional statement using breakpoints like this:

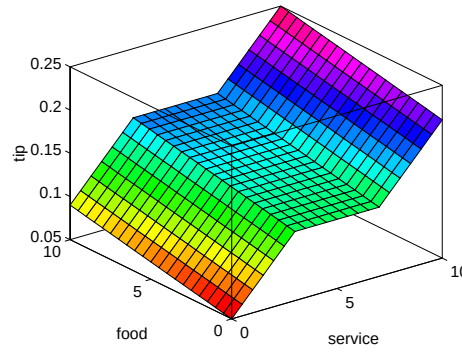
```
if service<3,
    tip=(0.10/3)*service+0.05;
elseif service<7,
    tip=0.15;
elseif service<=10,
    tip=(0.10/3)*(service-7)+0.15;
end
```

And the plot looks like this.



If we extend this back out 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 considering its humble start. How did we end up here? It was a little tricky to code this correctly, and it's definitely not easy to modify in the future. It works, but it's not easy to troubleshoot. It has hard-coded numbers going through the whole thing. 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 this:

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*

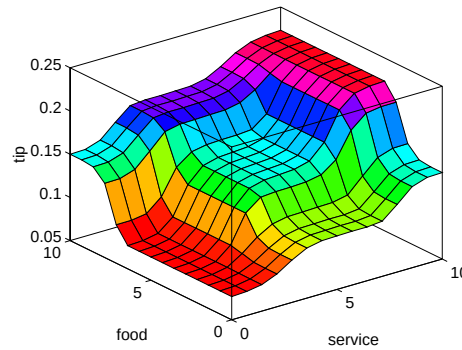
3. *if service is excellent or food is delicious then tip is generous*

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? or
- How do I define mathematically what an "average" tip is?

These are all questions we'll provide detailed answers to in the next few chapters. But 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 will be thoroughly explained in the next two sections. In the "Building Systems with the Fuzzy Logic Toolbox" section after that the entire tipping problem will be 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 massaged the shape of the overall output without needing to hack into what had already been done. In other words, the subsequent modification was pretty easy.

Moreover, by using fuzzy logic rules, the maintenance of the algorithm decouples along fairly clean lines. My 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. I don't have to change that part, no matter where in the world I travel. I can recalibrate the method quickly by simply shifting the fuzzy set that defines average without rewriting my rule.

You can do this sort of thing with lists of piecewise linear functions, but the medium is working against you. You're more likely to get tangled up in wires than you are to recalibrate the problem quickly. You can also buttress the piecewise linear solution by including many helpful comments. However, even if we lightly pass over the fact that the vast majority of code is woefully uncommented, it's still true that as code gets revised and updated, the comments can quickly slip into uselessness, or worse, they can actually provide misinformation.

Let me illustrate what I mean. 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.

And 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.

Tutorial

This section is designed to guide you through the fuzzy logic process step by step. The first several sections are meant to provide 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

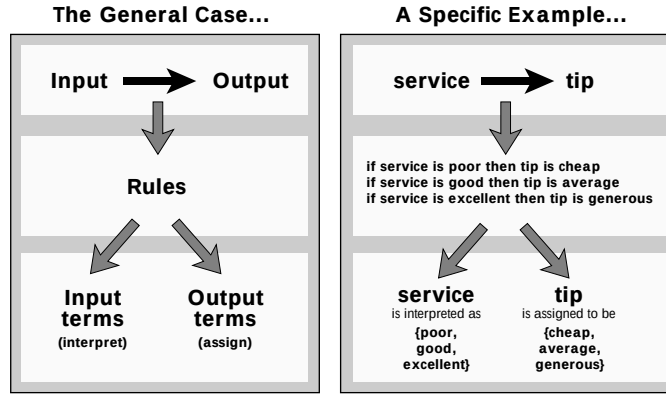
- **Foundations of fuzzy logic**, which is an introduction to the general concepts. If you're already familiar with fuzzy logic, you may want to skip this section.
- **Fuzzy inference systems**, which explains the specific methods of fuzzy inference used in the Fuzzy Logic Toolbox. Since the field of 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.

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.



The whole idea behind fuzzy inference is to interpret the values in the input vector and, based on some set of rules, assign values to the output vector. And that's really all there is to it.

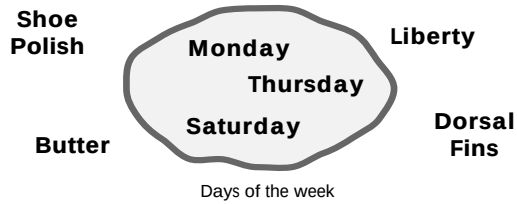
Foundations of Fuzzy Logic

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.

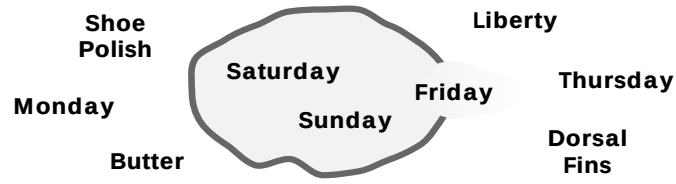


We call this set a classical set because it's been around for such a long time. It was Aristotle who first formulated the Law of the Excluded Middle, which says X must either be in set A or in set not-A. Another version runs like this:

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 that make up the weekend. The diagram below is one attempt at classifying the weekend days.



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)

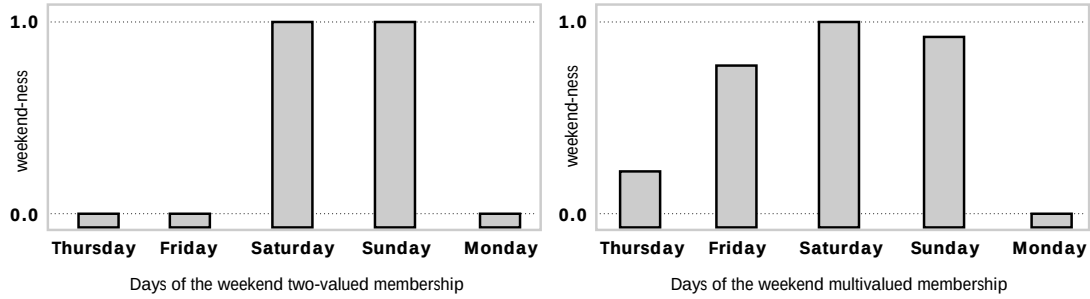
Q: Is Friday a weekend day?

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

Q: Is Sunday a weekend day?

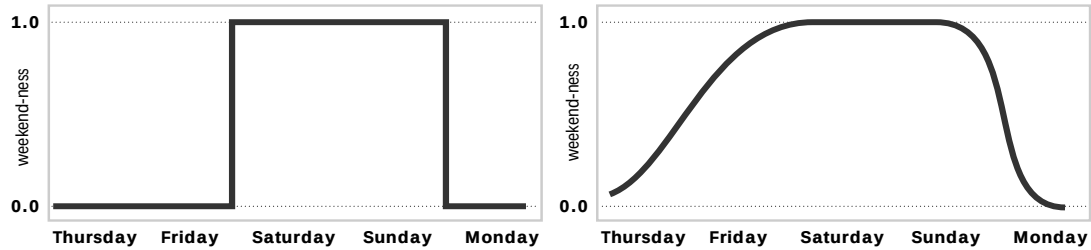
A: 0.95 (yes, but not quite as much as Saturday).

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.



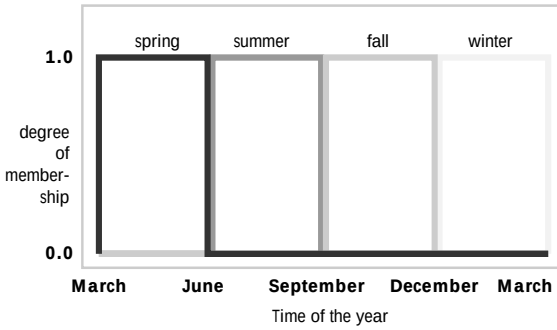
Days of the weekend two-valued membership

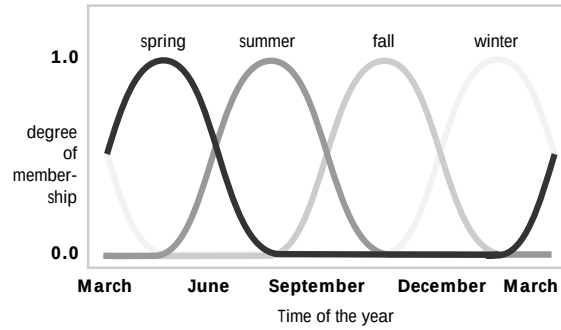
Days of the weekend multivalued membership

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 parts of Thursday to a small degree 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).

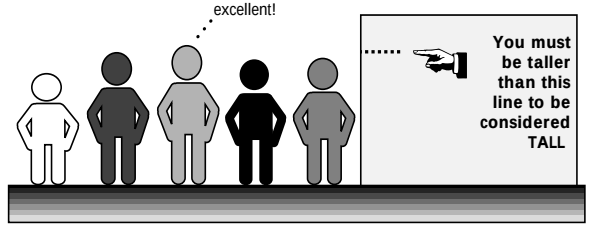




Membership Functions

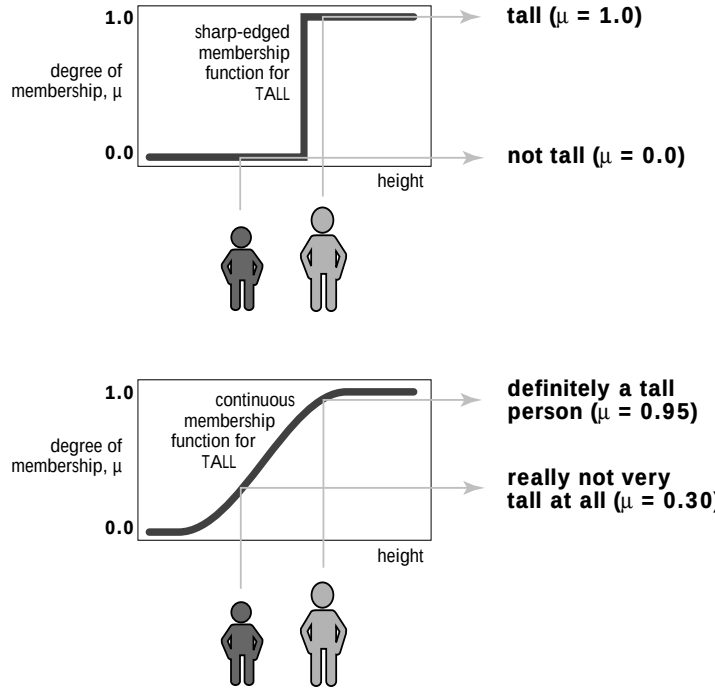
A *membership function* (MF) is a curve that defines how each point in the input space is mapped to a membership value (or degree of membership) between 0 and 1. The input space is sometimes referred to as the *universe of discourse*, a fancy name for a simple concept.

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

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

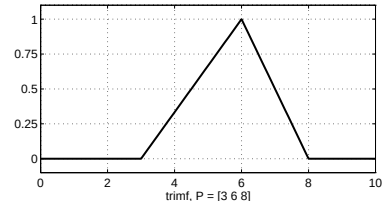
A fuzzy set is an extension of a classical set. If X is the universe of discourse and its elements are denoted by *x*, then a fuzzy set A in X is defined as a set of ordered pairs:

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

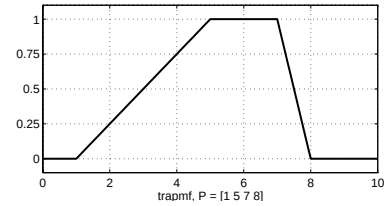
$\mu_A(x)$ is called the membership function (or MF) of *x* in A. The membership function maps each element of X to a membership value between 0 and 1.

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 Code`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.



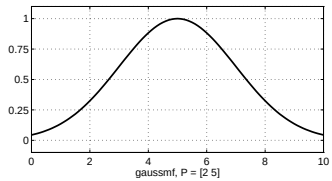
trimf



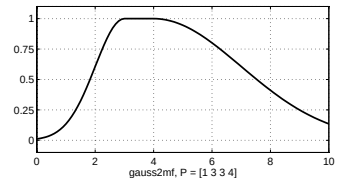
trapmf

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`.

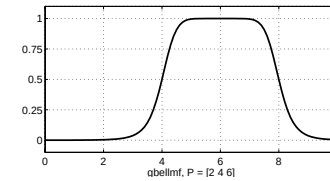
The *generalized bell* membership function is specified by three parameters and has the function name `gbellmf`. The bell membership function has one more parameter than the Gaussian membership function, so it can approach a non-fuzzy set if the free parameter is tuned. Because of their smoothness and 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

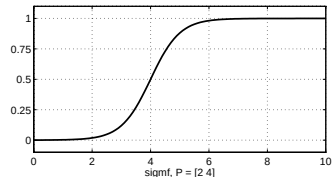


gauss2mf

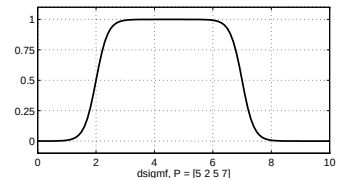


gbellmf

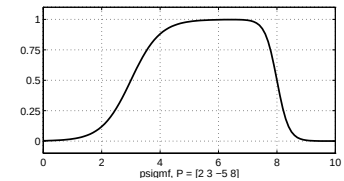
Although the Gaussian membership functions and bell membership functions achieve smoothness, they are unable to specify asymmetric membership functions, which are important in certain applications. Next we define the *sigmoidal* membership function, which is either open left or right. Asymmetric and 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

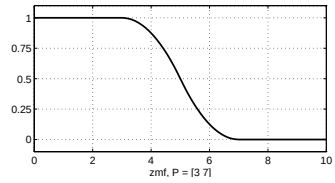


dsigmf

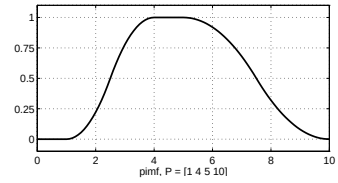


psigmf

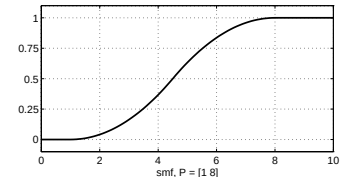
Polynomial based curves account for several of the membership functions in the toolbox. Three related membership functions are the *Z*, *S*, and *Pi curves*, all named because of their shape. The function `zmf` is the asymmetrical polynomial curve open to the left, `smf` is the mirror-image function 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 to the 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

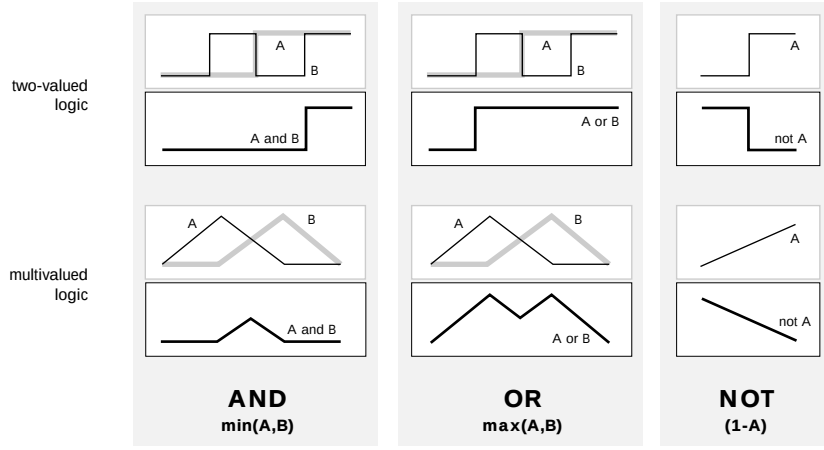
A	1 - A
0	1
1	0

NOT

Moreover, since there is a function behind the truth table rather than just the truth table itself, we can now go on to 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. But 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 the AND and OR operators can be easily customized if desired.

The intersection of two fuzzy sets A and B is specified in general by a function T which aggregates two membership grades as follows

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

where ⊗ is a binary operator for the function T. 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 two-place function T(.,.) 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 function S:

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

where ⊕ is a binary operator for the function S. These fuzzy union operators, which are often referred to as T-conorm (or S-norm) operators, satisfy the following basic requirements.

A *T-conorm* (or *S-norm*) operator is a two-place function S(.,.) 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)

The justification of these basic requirements is similar to that of the requirements for the T-norm operators.

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. But in order to say anything useful we need to make complete sentences. Conditional statements, if-then rules, are the things that make fuzzy logic useful.

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 the antecedent is an interpretation that returns a single number between 0 and 1, whereas the consequent is an assignment that assigns the entire fuzzy set B to the output variable *y*. So 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

So the input to an if-then rule is the current value for the input variable (*service*) and the output is an entire fuzzy set (*average*).

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. But 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 true or false)

in fuzzy logic: 0.5 $p \rightarrow$ 0.5 q (partial antecedents imply partially)

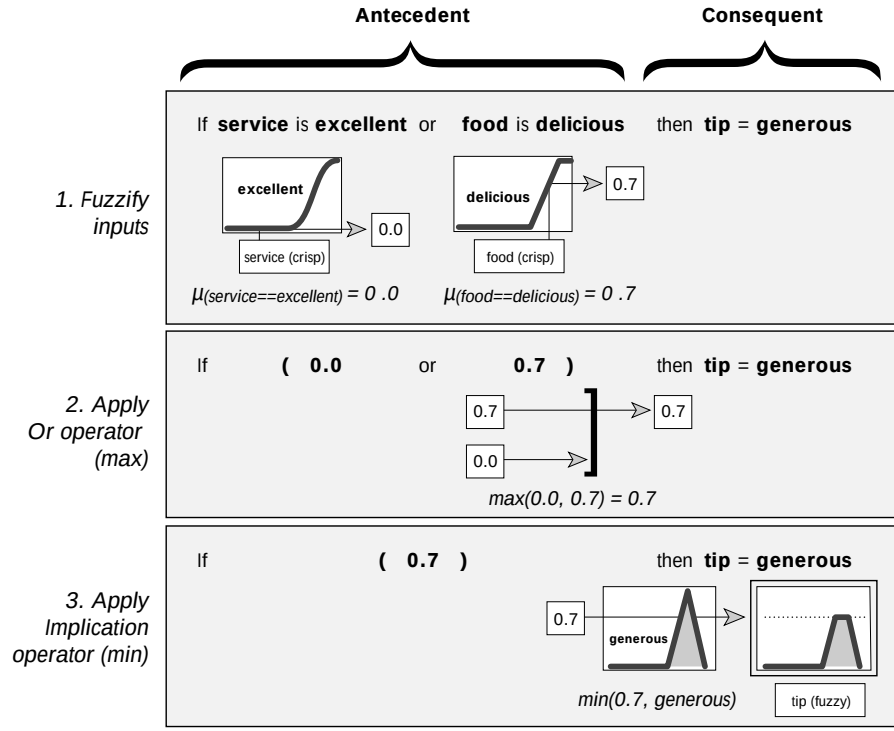
The antecedent of a rule can have multiple parts:

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 discussed 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. But 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 will be using truncation for the examples in this section.



Summary of If-Then Rules

Interpreting if-then rules is a three part process.

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

   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. If the antecedent is only partially true, 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, but in general we want the output for an entire collection of rules to be a single number. How are all these fuzzy sets distilled into a single crisp result for the output variable? First the output fuzzy sets for each rule are *aggregated* into a single output fuzzy set. Then the resulting set is *defuzzified*, or resolved to a single number. The next section shows how whole process works from beginning to end.

Fuzzy Inference Systems

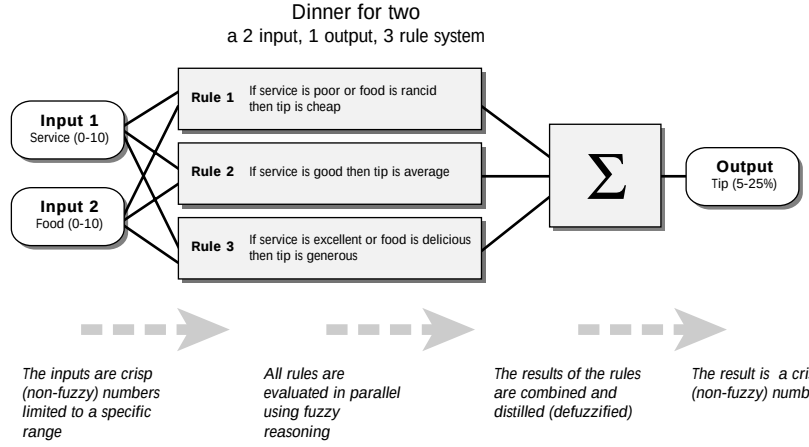
Fuzzy inference is the actual process of mapping from a given input to an output using fuzzy logic. The process involves all the pieces that we have discussed in the previous sections: membership functions, fuzzy logic operators, and if-then rules.

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 multi-disciplinary nature, the fuzzy inference system is known by a number of names, such as fuzzy-rule-based system, fuzzy expert system, fuzzy model, fuzzy associative memory, fuzzy logic controller, and simply (and ambiguously) fuzzy system. 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.

Dinner for Two, Reprise

In this section, we'll see how everything fits together using the same two-input one-output three-rule tipping problem that we saw in the introduction. Only this time we won't skip over any details. 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



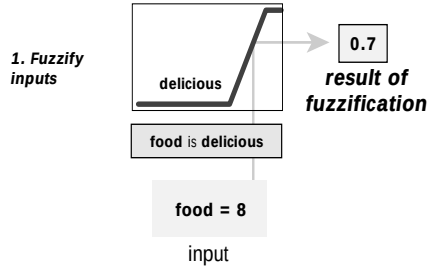
systems. Instead of sharp switching between modes based on breakpoints, we will glide smoothly from regions where the system's behavior is dominated now by this rule, now by that one.

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. 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 (always the interval between 0 and 1). So fuzzification really doesn't amount to anything more than table lookup or 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 against 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) fits the linguistic variable "delicious". In this case, we rated the food as an 8, which, given our graphical definition of delicious, corresponds to $\mu = 0.7$.



(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 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.

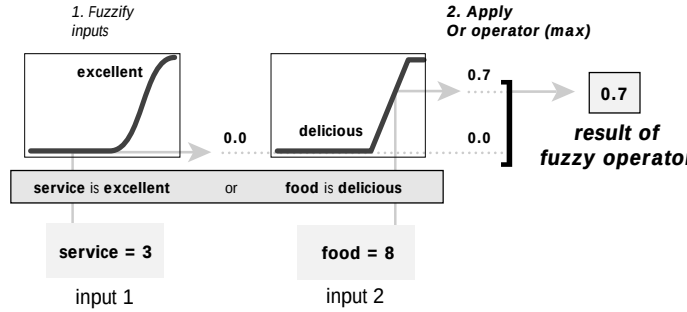
As described in the section on fuzzy logical operations, any number of well-defined methods can fill in for the AND operation or the OR operation. In the Fuzzy Logic Toolbox, two built-in AND methods are supported: *min* (minimum) and *prod* (product). Two built-in OR methods are also supported: *max* (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.

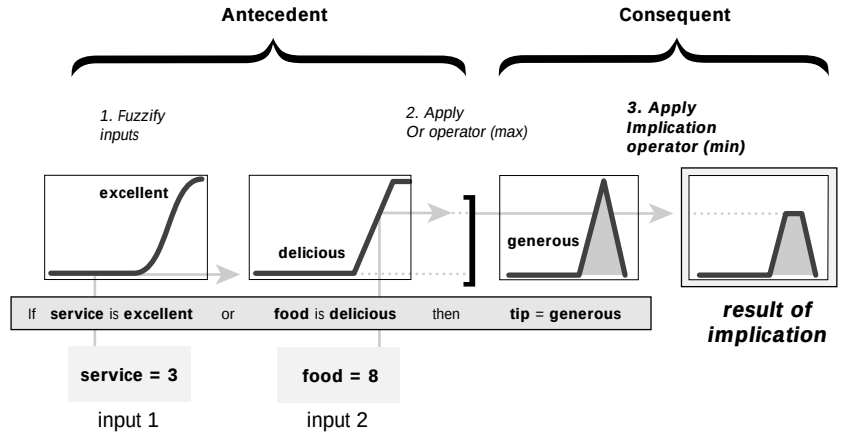


Step 3. Apply Implication Method

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. But 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.

The implication method is defined as the shaping of the consequent (a fuzzy set) based on 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 occurs 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.

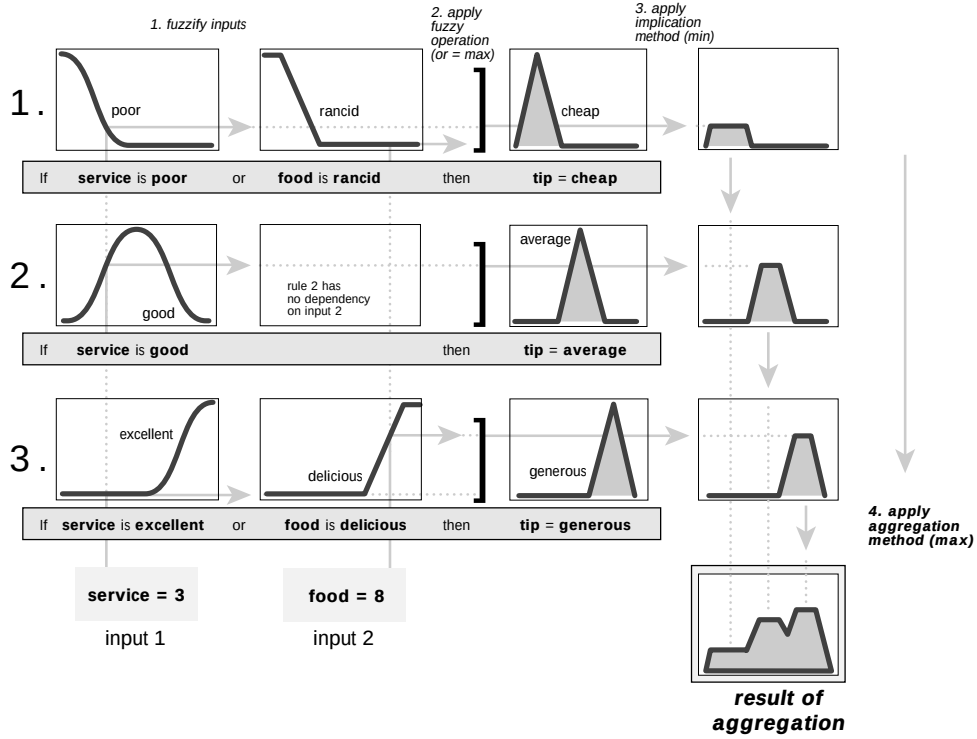


Step 4. Aggregate All Outputs

Aggregation is when we unify the outputs of each rule by joining the parallel threads. It's just a matter of taking all the fuzzy sets that represent the output of each rule and combining them into a single fuzzy set in preparation for the fifth and final step, defuzzification. Aggregation only occurs once for each output variable. 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 for the overall output.

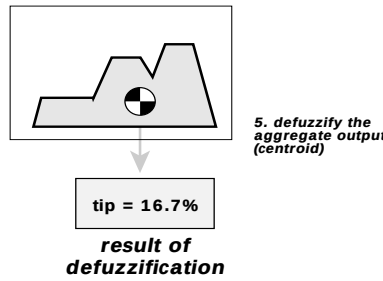


Step 5. Defuzzify

The input for the defuzzification process is a fuzzy set (the aggregate output fuzzy set) and the output is a single number—crispness recovered from fuzziness at last. As much as fuzziness helps the rule evaluation during the intermediate steps, the final output for each variable is generally a single crisp number. So, given a fuzzy set that encompasses a range of output values, we need to return one number, thereby moving from a fuzzy set to a crisp output.

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

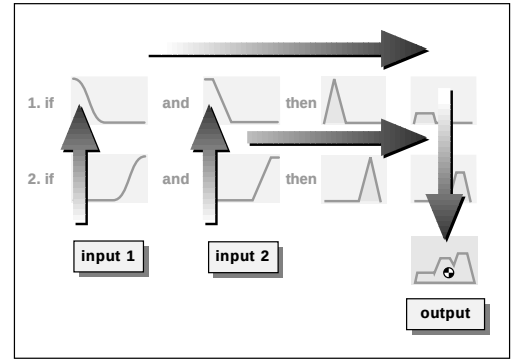
maximum value of the output set), largest of maximum, and smallest of maximum.



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.

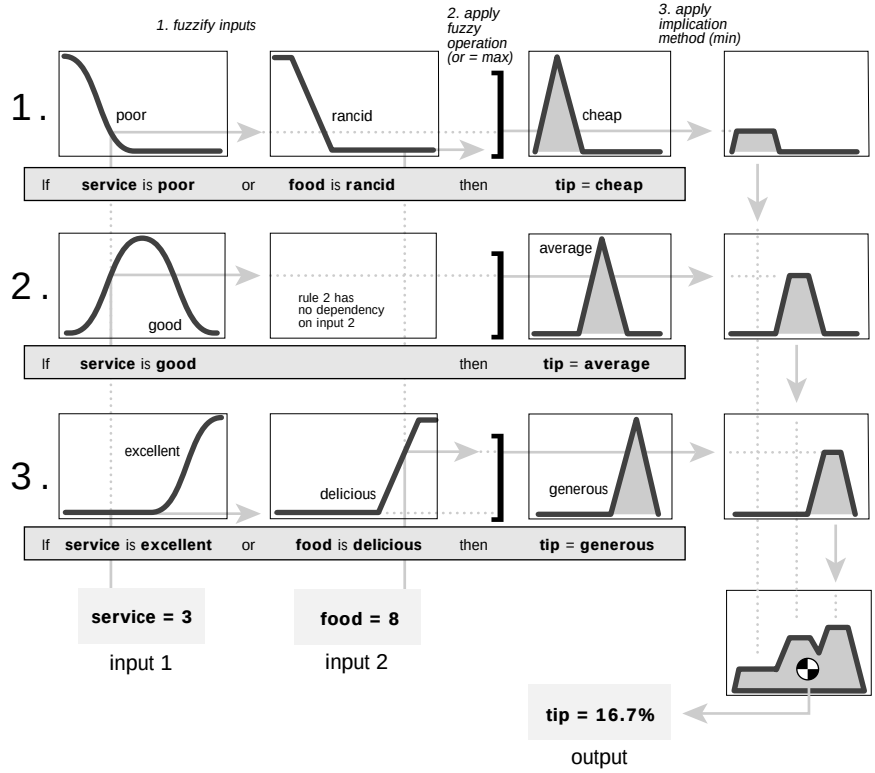
Interpreting the Fuzzy Inference Diagram



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, we can easily tell 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 Inference Viewer, described in the next section, is a MATLAB implementation of the fuzzy inference diagram.



Customization

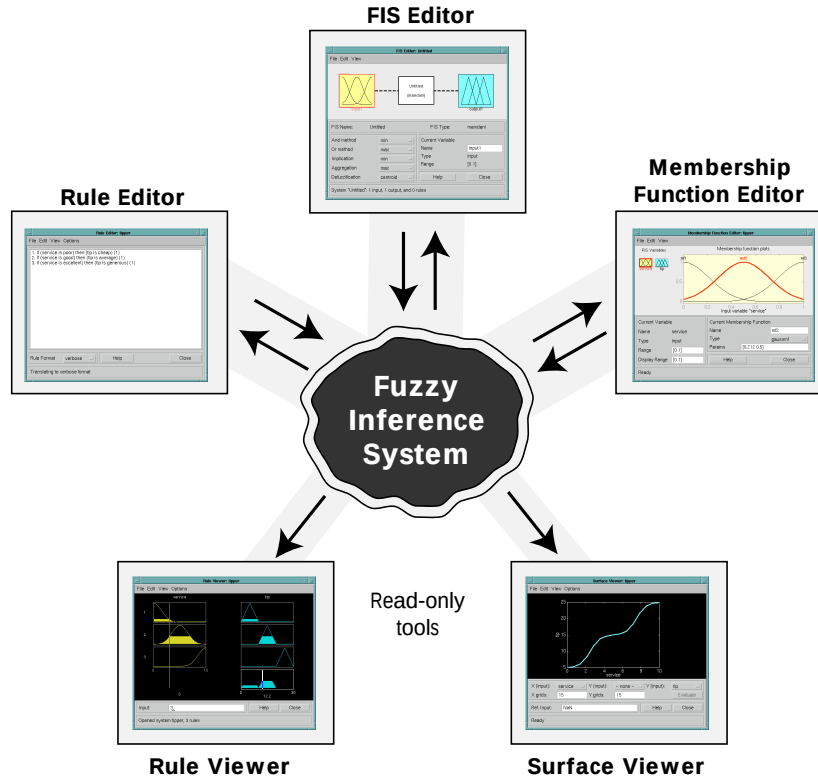
The Fuzzy Logic Toolbox is designed to give you as much freedom as possible, within the basic constraints of the process described here, to customize the fuzzy inference process for your application. For example, you can substitute your own MATLAB functions for any of the default functions used in the five steps detailed above: you make your own membership functions, AND methods, OR methods, implication methods, aggregation methods, and defuzzification methods. An open and easily modified system is one of the

primary goals of the Fuzzy Logic Toolbox.The next section will detail exactly how to make this system work using the tools provided.

Building Systems with the Fuzzy Logic Toolbox

Dinner for Two, from the Top

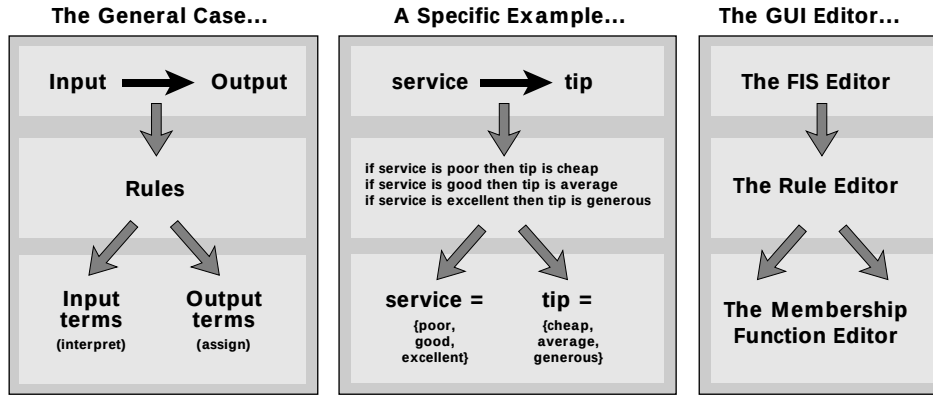
Now we're going to work through the exact same tipping example, only this time we'll be building it using the graphical user interface (GUI) tools provided by the Fuzzy Logic Toolbox. Although it's possible to use the Fuzzy Logic Toolbox by working strictly from the command line, in general it's much easier to build a system up 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 different GUIs are all effectively siblings in that you can have any or all of them open for any given system.



The FIS Editor handles the high level issues for the system: How many input and output variables? What are their names? 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 last two GUIs 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. It's a very powerful window full of information. The last of the five GUI siblings is the Surface Viewer. This tool can display how one of the outputs depends on any one or two of the inputs—that is, it generates and plots an

output surface map for the system. Some of the GUI tools have the potential to influence the others. For example, if you add a rule, you can expect to see the output surface change.

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



The five principal GUI editors all exchange information, if appropriate. Any one of them can read and write both to the workspace and to the disk. For any fuzzy inference system, any or all of these five editors 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.

Notice that the FIS Editor, the Membership Function Editor and the Rule Editor can all read and modify the FIS data, but the Rules Viewer and the Surface Viewer do not modify the FIS data in any way.

Getting Started

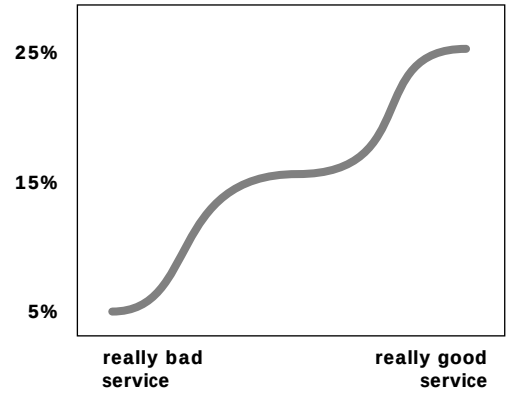
We'll start from scratch again with a basic description of the problem (noting that it is based on tipping as practiced in the U.S.)

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

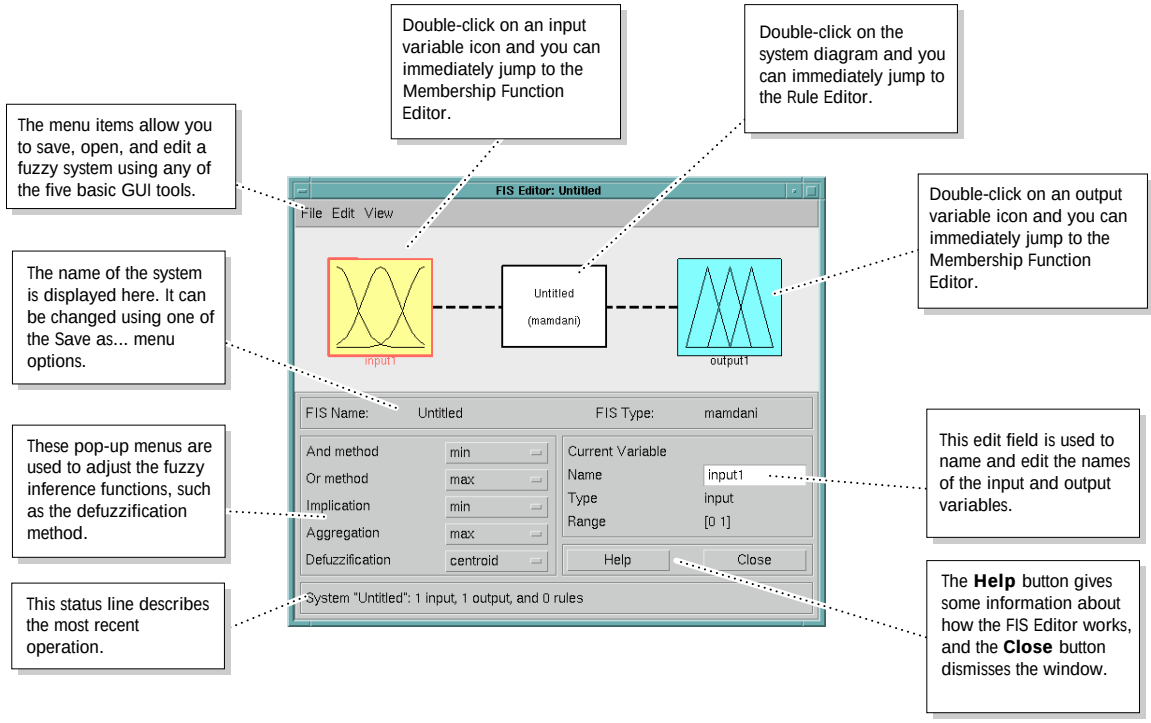
1. *if the service is poor then tip is cheap*
2. *if the service is good then tip is average*
3. *if the service is excellent then tip is generous*

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. So we know the rules, and we have an idea of what the output should look like. Now we can begin working with the GUI tools.

The FIS Editor



When creating a new fuzzy inference system from scratch, the place to start is the FIS Editor. To do that, type

```
fuzzy
```

This will call up a window that acts as the high-level (or "big picture") view of a FIS. At the top of the figure, there's a diagram that shows inputs on the left and outputs on the right. The system that is displayed is a default "start-up" system, since we didn't specify any particular system.

The purpose of this section of the manual is to build a new system from scratch. But if you want to save time and follow along quickly, you can load the already built system by typing

```
fuzzy tipper1
```

This will load the fuzzy inference system associated with the file `tipper1.fis` (the `.fis` is implied) and launch the FIS Editor. More on loading and saving later.

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 and each output variable. The sample membership functions shown in the boxes are just icons and do not represent the shapes of the actual membership functions.

Below the diagram is the name of the system and the type of inference used. The default, Mamdani-style inference, is what we've been describing so far and what we'll continue to use for this example. There is another slightly different type of inference, called Sugeno-style inference, that is explained in the next section. Below the name, 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 names of the input and output variables. Below that are the **Help** and **Close** buttons that call up on-line help and dismiss the window, respectively, and finally, at the bottom is a status line that relays information about the system from time to time.

The first thing to notice from the diagram at the top of the figure is that the default system already has one input and one output. That suits us well, since our one input is service and our one output is tip. We'd like to change the names to reflect that, though.

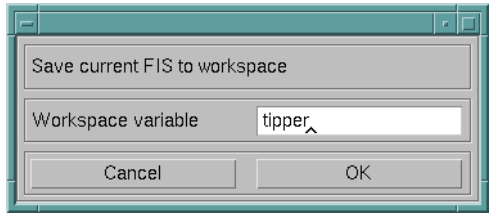
1. Click once on the left-hand (yellow) box marked *input1* (the box will be highlighted in red).

2. In the white edit field on the right, change *input1* to *service* and press **Return**.

3. Click once on the right-hand (blue) box marked *output1*.

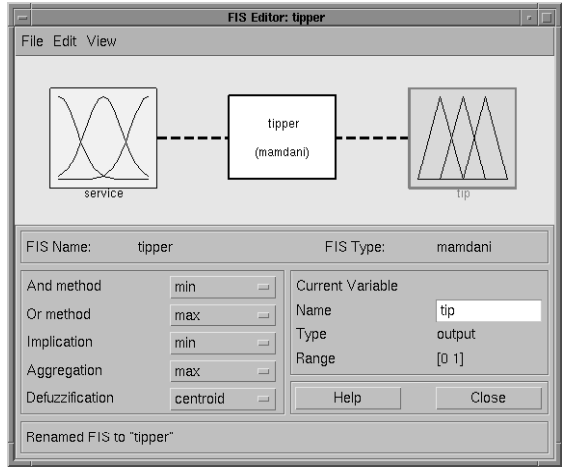
4. In the white edit field on the right, change *output1* to *tip*.

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



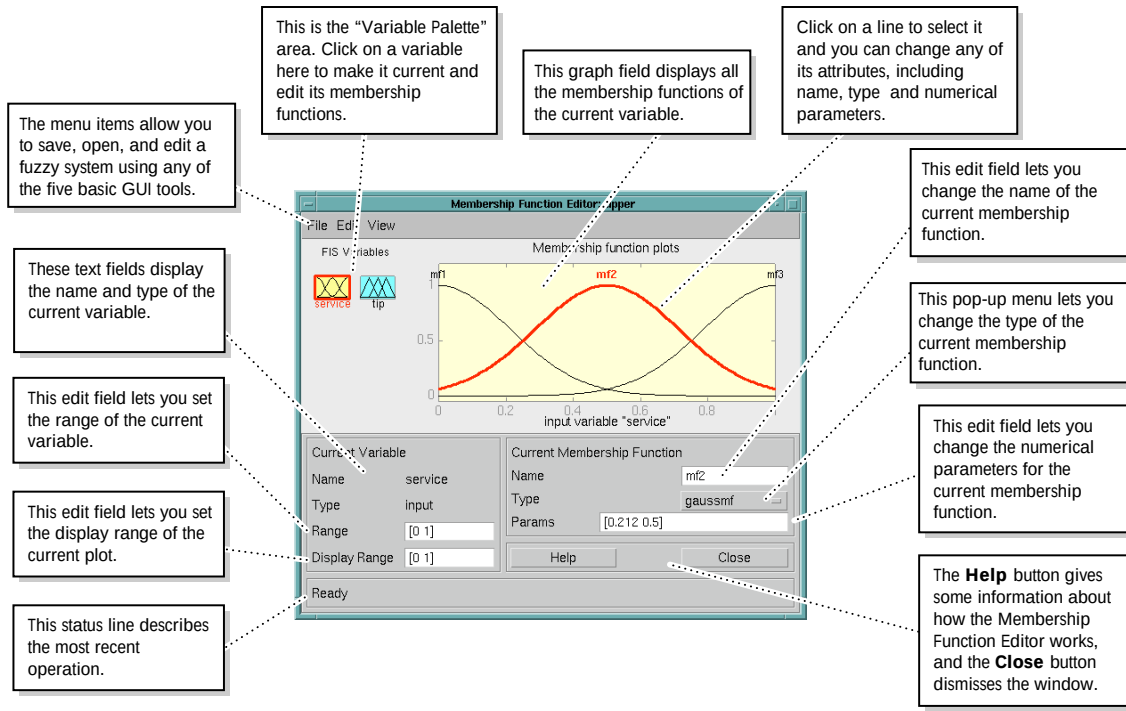
6 Enter the variable name *tipper* and click on **OK.**

You should 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 should look something like this.



We'll leave the inference options in the lower left in their default positions for now. So we've entered all the information we need to in this particular GUI. The next thing to do is define the membership functions associated with each of the variables. To do this, we need to open up the Membership Function Editor by pulling down the **View** menu item and selecting **Edit Membership Functions...**.

The Membership Function Editor



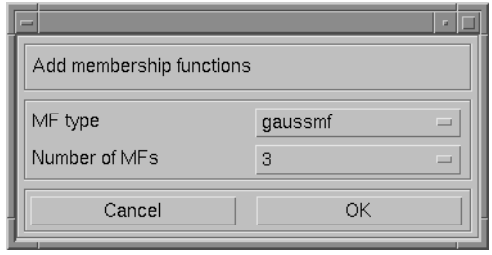
The Membership Function Editor shares some features with the FIS Editor. In fact, all of the five basic GUI tools have similar menu options, status lines, and **Help** and **Close** buttons. The Membership Function Editor is the tool that lets you display and edit all of the membership functions for the entire fuzzy inference system, including both input and output variables.

There are no membership functions to start off with. On the left side of the graph area is a "Variable Palette" that lets you set the current variable. The membership functions from the current variable are displayed in the main graph. Below the Variable Palette is some information about the type and name of the current variable. There is one text field 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). In the lower right of the window are the controls that let you change the name, position, and shape of the currently selected membership function.

1 Make sure the input variable is selected in the Variable Palette. Set the **Range** to vector `[0 10]`.

2 Select **Add MFs...** from the **Edit** menu and add three Gaussian curves to the input variable service.



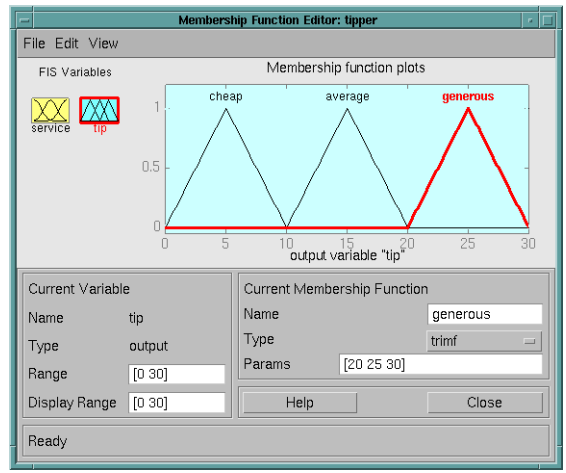
3 Click once directly on the leftmost curve. Change the name of the curve to *poor*. Change the parameters listing to [1.5 0].

4 Name the middle curve *good* and the rightmost curve *excellent* and change the first parameters to 1.5.

Next we need to create the membership functions for the output variable, tip. We already know the names for these membership functions: cheap, average, and generous. To display the output variable membership functions, use the Variable Palette on the left. The input range was a rating scale of 0 to 10, but the output scale is going to be a tip between 5 and 25 percent.

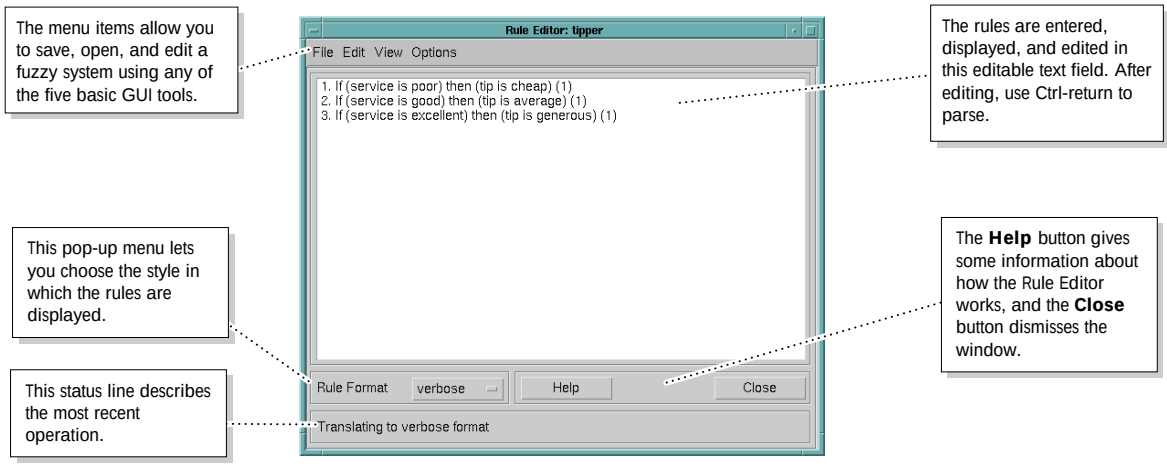
We'll use triangular membership function types for the output. First, set the Range (not the Display Range) to [0 30]. The *cheap* membership function will have the parameters [0 5 10], the *average* membership function will be [10 15 20] and the *generous* membership function will be [20 25 30]. So each of these

is a fuzzy set centered on the typical number. Your system should look something like this.



Now that the variables have been named, and the membership functions have appropriate shapes and names, we're ready to write down the rules. To call up the rule editor, go to the **View** menu and select **Edit rules...**.

The Rule Editor



The Rule Editor contains a large editable text field for displaying and editing rules. It also has some by now familiar landmarks similar to those in the FIS Editor and the Membership Function Editor, including the menu bar and the status line. A format pop-up menu is the only window specific control—this is used to set the format for the display.

In the main (white) text area, type the following rules and then press **Ctrl-Return**.

if service is crummy then tip is cheap

if service is good then tip is average

if service is excellent then tip is generous

It gets returned as

 # if service is crummy then tip is cheap

1. If (service is good) then (tip is average) (1)

2. If (service is excellent) then (tip is generous) (1)

There should be a message in the status window at the bottom of the figure that reads “There is no MF called crummy for the input variable service.” The # symbol is inserted at the beginning of the first line to indicate there was a problem parsing that rule. Every time you press **Ctrl-Return**, the Rule Editor tries to parse every rule. Any rules that confuse the parser are marked with the

symbol. Change the word “crummy” to “poor” and press **Ctrl-Return** so the editor can interpret the rule properly.

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

The numbers in the parentheses represent weights that can be applied to each rule if desired. If you do not specify them, they are assumed to be one. The **Rule Format** pop-up menu in the lower left indicates that you're looking at the verbose form of the rules. Try changing it to symbolic. You should see

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

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

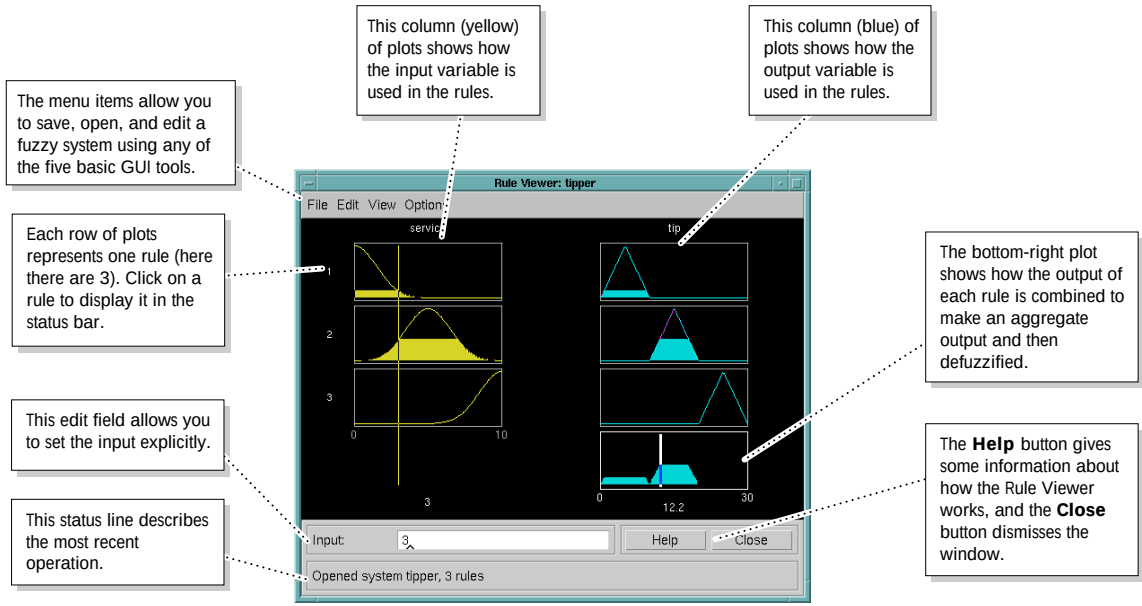
3, 3 (1) : 1

This is the version that the machine deals with. The first column in the matrix 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. So 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 immaterial.

So the symbolic format doesn't bother with the terms if, then, and so on. But the indexed format doesn't even bother with the names of your variables. Obviously the functionality of your system doesn't depend on how beautifully you named your variables and membership functions (if it did, it would be called fuzzy poetry instead of fuzzy logic). The whole point of naming variables descriptively is, as always, making the system easier to interpret.

Now the system has been completely defined: we've got the variables, membership functions, and rules necessary to calculate tips. 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'll see a single figure window with seven small plots nested in it. In addition there are the now familiar items like the status line and the menu bar. In the lower right there is a text field where you can enter a specific input value, if desired.

The two 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. So the first column of plots (the three yellow plots) shows the membership functions referenced by the antecedent, or if-part, of each rule. The second column of plots (the three blue plots) shows the membership

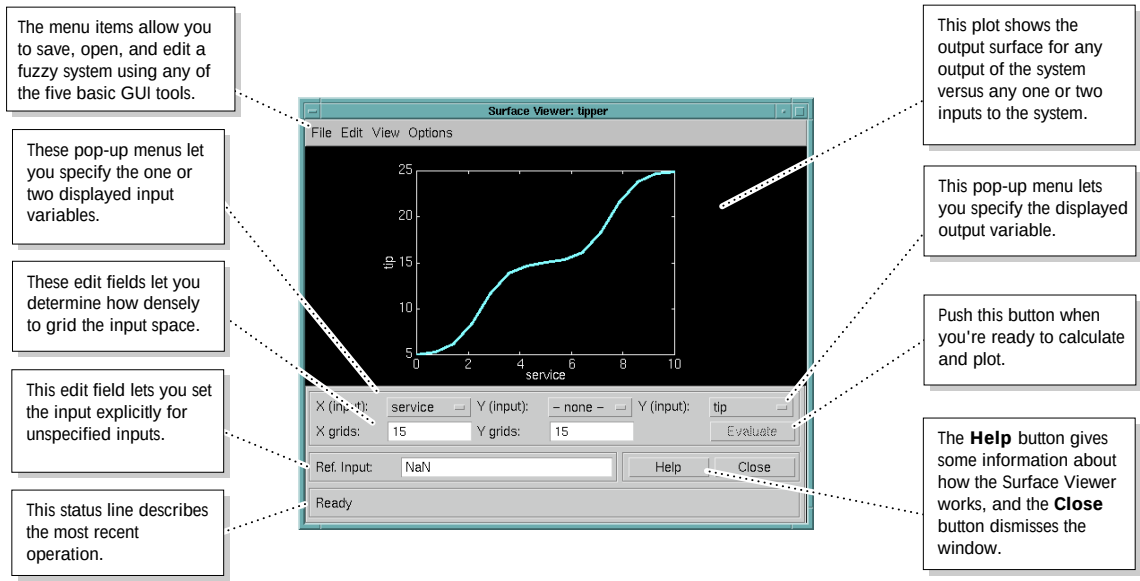
functions referenced by the consequent, or then-part of each rule. If you click once on a rule number, the corresponding rule will be displayed at the bottom of the figure.

There is a yellow index line across the input variable plots that you can move left and right by clicking and dragging with the mouse. This changes the input value. When you release the line, a new calculation is performed, and you can see the whole fuzzy inference process take place before your eyes. 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. 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 antecedent—this is the implication process in action. Finally the aggregation occurs down the second 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 presents a busy scene, and interpreting it can take some getting used to, but once you become familiar with it, you can take in the whole fuzzy inference process in one sweeping view. The Rule Viewer is very good, for example, at showing how the shape of certain membership functions is influencing the overall result. Since it plots every part of every rule, it can become unwieldy for particularly large systems, but in general 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 not surprisingly, 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. But 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.

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 Reference Input field is used in situations when there are more inputs required by the system than are currently being varied. 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 our monitors simply cannot display a five-dimensional shape. In such a case the Reference Input would be a four element vector with NaNs holding the place of the varying inputs while numerical values would indicate those values that remain fixed.

This concludes the quick walkthrough of each of the main GUI tools. Notice that for the tipping problem, the output of the fuzzy system nicely matches our original idea for what the shape of the fuzzy mapping from service to tip. If, after all this work, this were the only value we got from the fuzzy system, we might be tempted to say "Why bother? I could have just drawn a quick lookup table and been done an hour ago!" But one of the beauties of fuzzy logic is the ease with which a system can be quickly modified, extended, and massaged.

Two-inputs One-output, or What About the Food?

Now we might say: "This is all well and good, but I think the food quality should be reflected in the tip as well." We've looked at this problem in earlier sections, but for the sake of clarity, we will restate the extended tipping problem.

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

The thing we need to do right away is add another input variable, and to do that, we need to get back to the FIS Editor.

1. Return to the FIS Editor and from the **Edit** menu, select **Add input variable**.

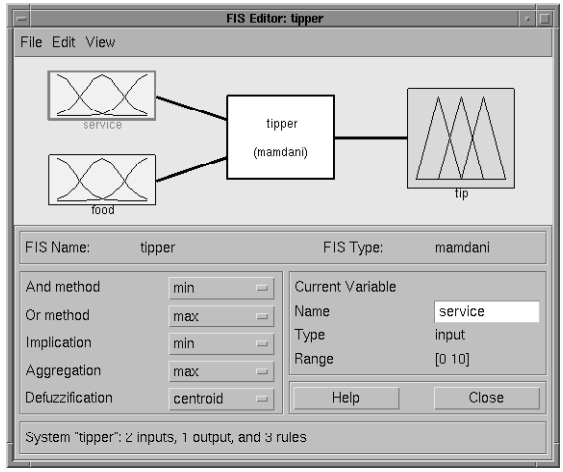
2. Name the new variable *food*.

3. Return to the Membership Function Editor and add two trapezoidal membership functions.

4. Change the Range to [0 10].

5. Name the leftmost membership function *rancid* and give it the parameters [-2 0 1 3]. Name the rightmost membership function *delicious* and give it the parameters [7 9 10 12].

If you return to the FIS Editor at this point, you should see something like this.

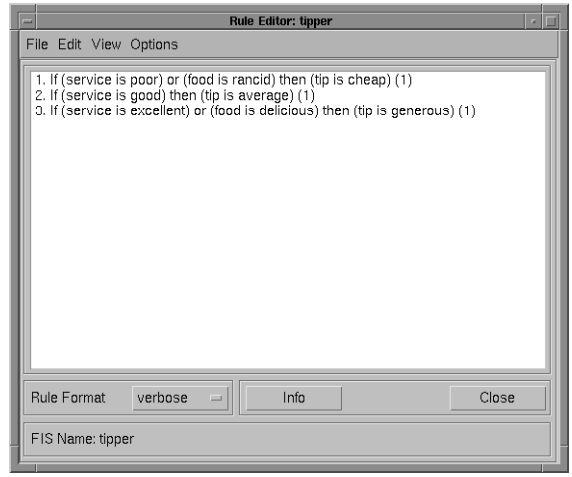


Now we need to update the rules appropriately. Add two new rules to the bottom of the list:

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

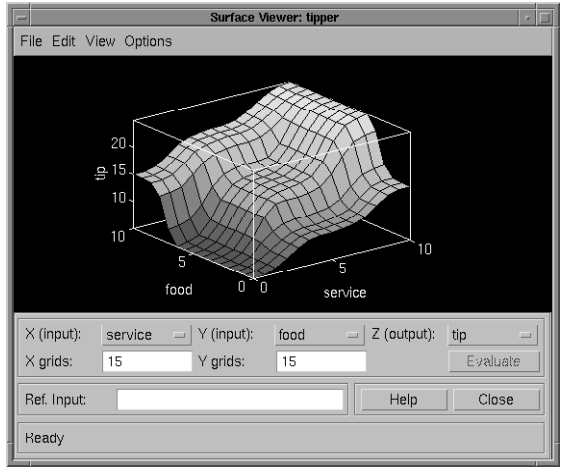
In fact, because of the parallel nature in which the rules get evaluated, it makes no difference whether these two rules are added to the bottom of the rule list, or the three existing rules are modified like so

1. *if service is poor or food is rancid then tip is low*
2. *if service is good then tip is average*
3. *if service is excellent or food is delicious then tip is generous*



Finally return to the Surface Viewer—here is where we see the real value of fuzzy logic. We originally built a system that was effectively doing a simple one-dimensional table lookup. But by adding only two new rules, we've generated a complex surface that nevertheless conforms to our desires for the

tipping algorithm. Fuzzy logic systems are easily modified and sculpted to suit the needs of the problem.



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 FIS matrix for the system. FIS files and FIS matrices can represent the same system, but they're extremely different from one another.

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. The implication

method does an element by element matrix operation, also like the `min` function, as in

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

The only limitation on customized membership functions is that they cannot use more than four parameters.

Working from the Command Line

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

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

If you look at the text file `tipper.fis` by entering

```
type tipper.fis
```

you'll see that this fuzzy system is stored as ASCII text in a fairly straightforward way. The function `readfis` takes all the information in this text file and puts it into a big matrix, in this case, `a`. The matrix `a` is known as a FIS (Fuzzy Inference System) matrix. This matrix is simply a bookkeeping mechanism that keeps object-like information in a two-dimensional matrix of floating point numbers. This matrix is always cast as a an array of numbers, even though much of it is ASCII text. In fact, it's almost never convenient to look at it as a raw variable. Because of this, specialized access functions exist to simplify the process of dealing with the FIS matrix. To learn more about it, type

```
getfis(a)
```

This returns some fairly generic information about the fuzzy inference system, such as its name, the number of input variables, output variables, and so on. You can use `getfis` to learn more about any fuzzy inference system. Try the following:

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

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

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

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

```
showfis(a)
```

This returns a long printout listing all the rows of `a` and what they store. This function is intended more for debugging than anything else, but it shows all the information recorded in the FIS matrix row by row. As a rule, you will never have to worry about what information goes on which line in the FIS matrix. You need only keep straight which variable is associated with which system.

Since `a` designates the fuzzy tipping system, we 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 desired GUI.

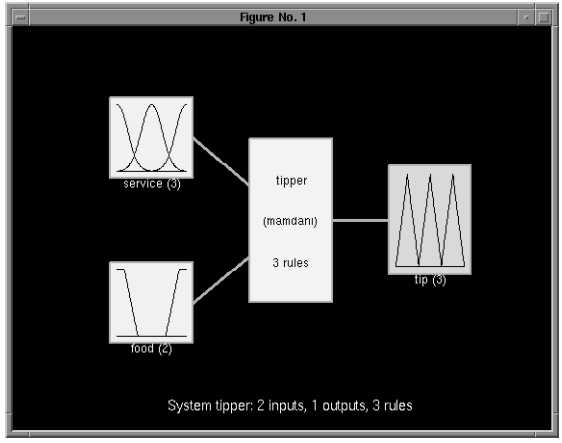
- `fuzzy(a)` FIS Editor
- `mfedit(a)` Membership Function Editor
- `ruleedit(a)` Rule Editor
- `ruleview(a)` Inference Viewer
- `surfview(a)` Surface Viewer

And once any of these GUIs has been opened, you can jump to any of the other GUIs using the pull-down menu rather than the command line.

System Display Functions

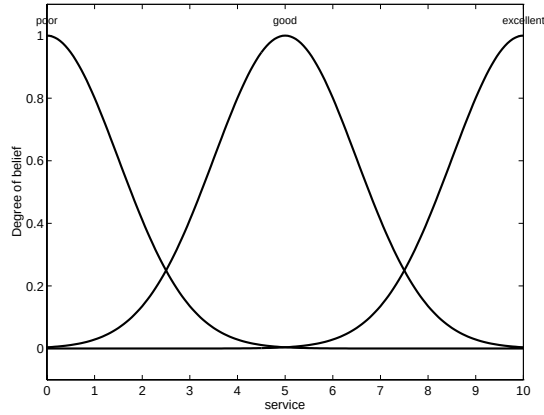
There are three functions designed to give you a high-level view of your fuzzy inference system from the command line: `plotfis`, `plotmf`, and `gensurf`. The first of these displays the whole system as a block diagram much as it would appear on the FIS Editor.

```
plotfis(a)
```

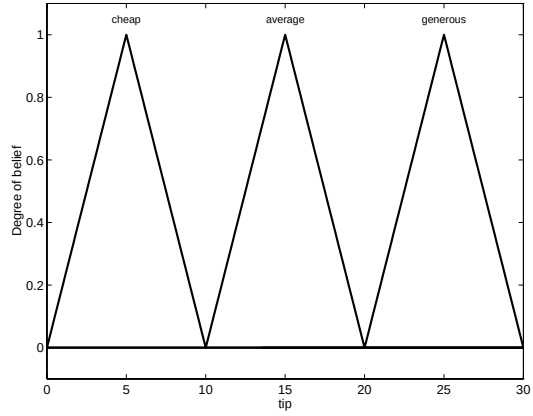


The function `plotmf` will plot all the membership functions associated with a given variable.

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

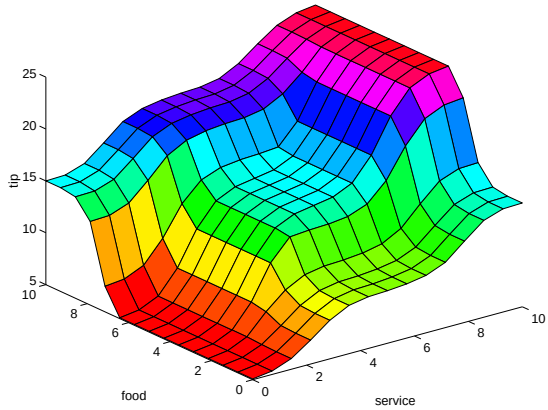



```
plotmf(a,'output',1)
```



Finally, the function `gensurf` will plot any one or two inputs versus any one output of a given system. The result is either a line or a three-dimensional surface.

```
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. 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 matrix according to the following logic: If there are *m* inputs to a system and *n* outputs, then the first *m* columns of the rule matrix 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). So the rule matrix associated with the rule shown above is

1 3 2 0.5 2

Here is how you would build the entire tipping system from the command line.

```
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]);

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`.

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

This function can also be used for multiple collections of inputs, so each row of the input matrix is a different input vector. By doing multiple evaluations at once, you get a tremendous boost in speed.

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

M-File or MEX-File?

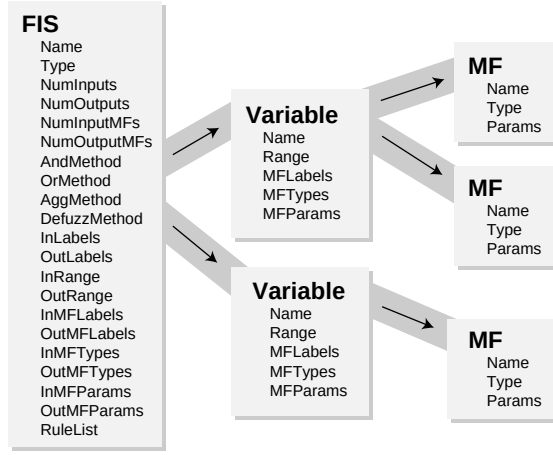
There are two different functions (`evalfis.m` and `evalfis.mex`) that can do the actual fuzzy inference for a given set of inputs, though only one of them is used at any given time. One is an M-file and the other is a MEX-file, and they return exactly the same result. The MEX-file is much much faster, but if you are curious about how the algorithms are implemented, you may want to inspect or even modify the M-file. As long as it is on the MATLAB path, `evalfis.mex` will be used preferentially to `evalfis.m`. Every time `evalfis.mex` is called it builds a data structure in memory, performs the FIS evaluation, and destroys the data structure. You cannot access this data structure directly.

The FIS Matrix

The FIS matrix is the MATLAB object that contains all the fuzzy inference system information. This matrix is stored inside each GUI tool. Access functions such as `getfis` and `setfis` make it easy to examine and modify its structure. The access functions are also important because they protect you from any changes to the data structure in future versions of the toolbox. The data structure may change, but the access functions will continue to work as before.

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

This object can itself be thought of as a hierarchy of other objects, as shown in the diagram below:



Since MATLAB deals only with matrices of double precision floating point numbers, the FIS matrix is exactly that. The information is arranged in it as shown below (the following list is actually just the output of the `showfis` command).

```
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. RuleList         [1 1 1 1 2]
43.                  [2 0 2 1 1]
44.                  [3 2 3 1 2]
```

Access functions for dealing with this matrix include `getfis`, `setfis`, `showfis`, `addvar`, `addmf`, `addrule`, `rmvar`, and `rmmf`. These are the only functions that interact directly with the elements of the FIS matrix. Other functions may use the information provided in the FIS matrix, but they will do so by means of one of these functions. See Chapter 3, Reference, for more information.

Since the matrix is necessarily rectangular, zeros are used to fill out each row to the required length. These matrices can therefore be saved as sparse matrices if memory savings are desired.

FIS Files on Disk

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

If you prefer, you can interact with the fuzzy inference system by editing its `.fis` text file rather than using any of the GUIs. This is occasionally the most convenient way to edit a fuzzy inference system. You should be aware, however, that changing one entry may obligate you to change another. For example if you add a rule to the rule list you must also increment the `NumRules` variable or the system will not load properly. Also notice that the rules are in "indexed" format. 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
```

Sugeno-style 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. 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 have described in previous sections differs somewhat from the methods described in the original paper, the basic idea is much the same.

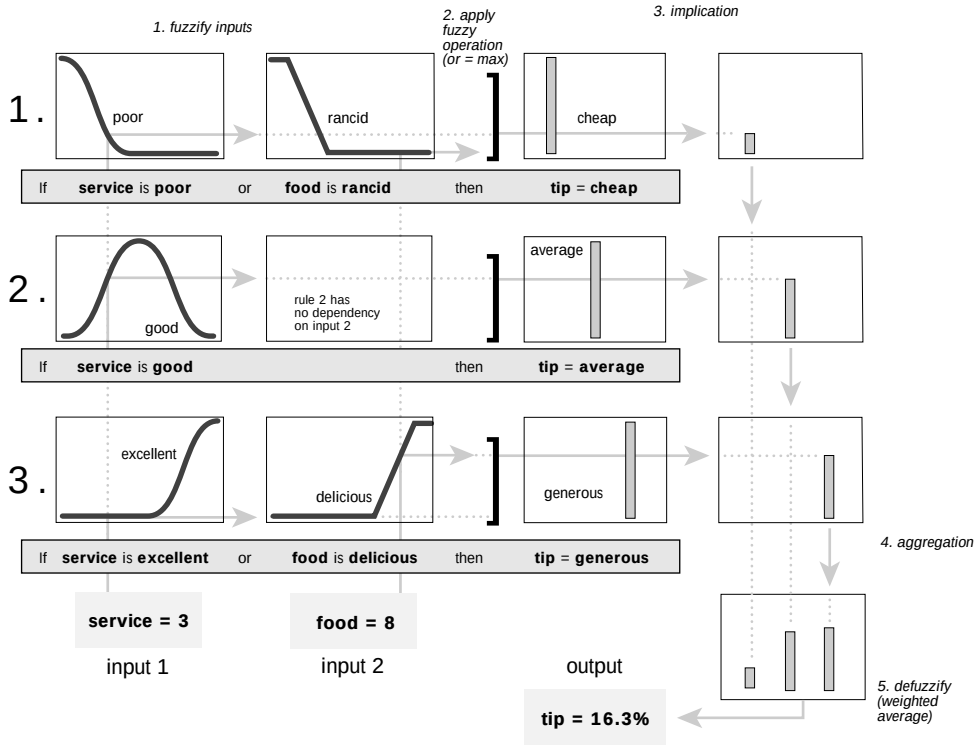
Mamdani-style inference, as we have defined it for the Fuzzy Logic Toolbox, expects the output membership functions to be fuzzy sets. After the aggregation process, there is a fuzzy set for each output variable that needs defuzzification. It's possible, and in many cases much more efficient, to use a 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 to find the centroid of a two-dimensional shape. Rather than integrating across a continuously varying two-dimensional shape to find the centroid, we can just find the weighted average of a few data points. Sugeno systems support this kind of behavior.

In this section we will 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.

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

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

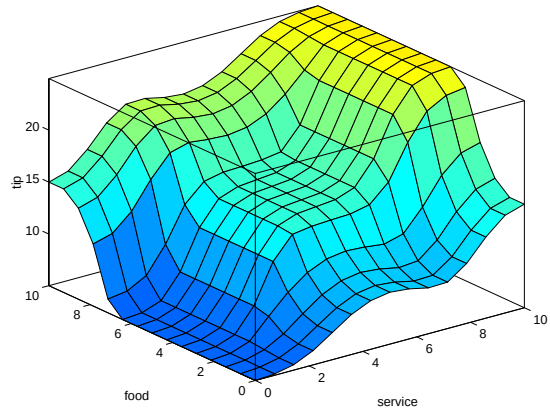
where A and B are fuzzy sets in the antecedent, while k is a crisply defined constant in the consequent. When the output of each rule is a constant like this, the similarity with Mamdani's method is striking. The only distinctions are the fact that all output membership functions are singleton spikes, 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-style representation of the now-familiar tipping model. If you load the

system and plot its output surface, you will see it is almost exactly 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 walk around 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 of 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 interpolative supervisor of multiple linear controllers that apply in 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 linear gains across the input space; it's a natural and

efficient gain scheduler. A Sugeno system is also 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]
```

So 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

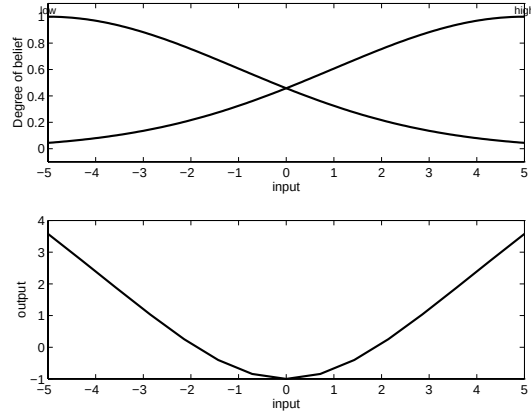
output = (1)**input* + (-1)

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-style system gives you the freedom to incorporate linear systems into your fuzzy systems. By extension, we could build a fuzzy system that switches between several optimal linear controllers as a very nonlinear system moves around in its operating space.

Conclusion

Any one Sugeno rule can be more expressive than a rule in a Mamdani system. Because it is a more compact and computationally efficient representation than a Mamdani system, the Sugeno system lends itself to adaptive techniques. These adaptive techniques in turn open up a whole new world by creating the entire fuzzy system for you.

Here are some final considerations about the two different methods.

Advantages of Sugeno's method:

- Computational efficiency
- Works well with linear techniques (e.g. PID control, etc.)
- Works well with optimization and adaptive techniques
- Guaranteed continuity of the output surface
- Better suited to mathematical analysis

Advantages of Mamdani's method:

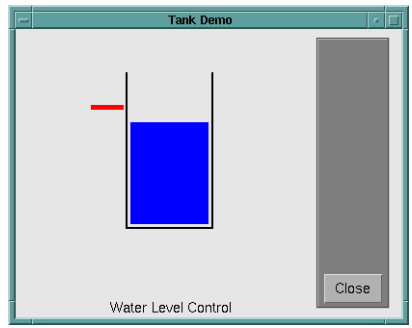
- More intuitive
- Widespread acceptance
- Better suited to human input

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

The example we'll look at is one of water level control. Picture a tank with a pipe flowing in and a pipe flowing out. We 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.



So 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 current 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.

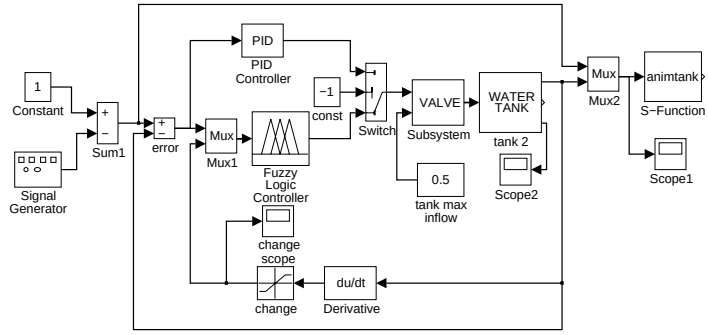
1. *If (level is okay) then (valve is no_change) (1)*
2. *If (level is low) then (valve is open_fast) (1)*
3. *If (level is high) then (valve is close_fast) (1)*

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

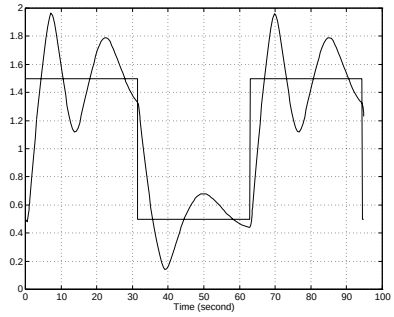
environment. A Simulink block diagram for this system is shown below. The Simulink block diagram for this system is `sltank`. Typing

```
sltank
```

at the command line, causes the system to appear.



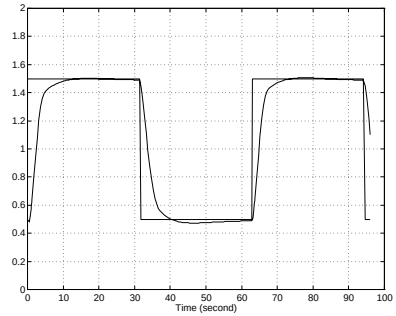
At the same time, the file `tank.fis` is loaded into the FIS matrix `tank`. Some experimentation shows that these three rules are not very good by themselves, since the water level tends to oscillate around the desired level.



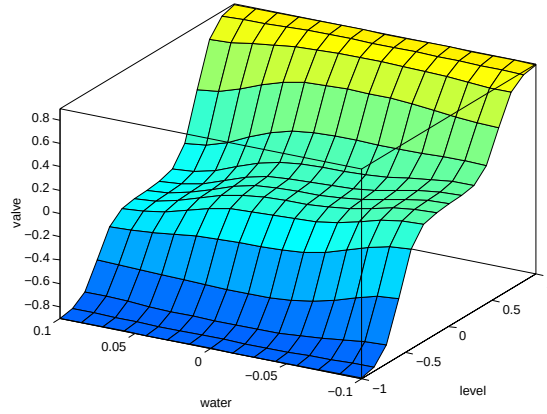
We need to add another input, the water level's rate of change, to slow down the valve when we get close to the right level.

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)*

With all five rules in operations, the step response 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. Notice that a PID controller would not have this latitude. The error, error-change, valve command 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 Toolbox, and so on), you can, for example, quickly do a fair comparison of a fuzzy controller versus a linear controller versus a neural network controller.

To load the system from the disk, type

```
a=readfis('tank.fis');
```

You can look at the five rules in this system from the command line by typing

```
showrule(a)
```

Or if you want to use the standard GUI tool for reviewing them

```
ruleedit(a)
```

The command `showrule` is the function that is called by `ruleedit` when it displays the rules.

Building Your Own Simulations

To build your own Simulink systems that use fuzzy logic, simply copy the Fuzzy Logic Controller block out of this system (or any of the other demo Simulink systems available with the toolbox) and place it in your own block diagram. You can also open the Simulink system called `fuzblock`, which contains the Fuzzy Logic Controller block all by itself. Make sure that the fuzzy inference system (FIS) matrix 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.

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.

ANFIS

ANFIS stands for Adaptive Neuro-Fuzzy Inference System. Fundamentally, ANFIS is about taking a fuzzy inference system (FIS) and tuning it with a backpropagation algorithm based on some collection of input-output data. This allows your fuzzy systems to learn.

A network structure facilitates the computation of the gradient vector for parameters in a fuzzy inference system. Once the gradient vector is obtained, we can apply a number of optimization routines to reduce an error measure (usually defined by the sum of the squared difference between actual and desired outputs). This process is called learning by example in the neural network literature.

Some Constraints

Since ANFIS is much more complex than the fuzzy inference systems discussed so far, you are not able to use all the available fuzzy inference system options. Specifically, ANFIS only supports Sugeno systems subject to the following constraints:

- First order Sugeno-type systems
- Single output derived by weighted average defuzzification
- Unity weight for each rule

An error occurs if your FIS matrix for ANFIS learning does not comply with these constraints.

Moreover, ANFIS is highly specialized for speed and 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 make do with the ones provided.

An Example

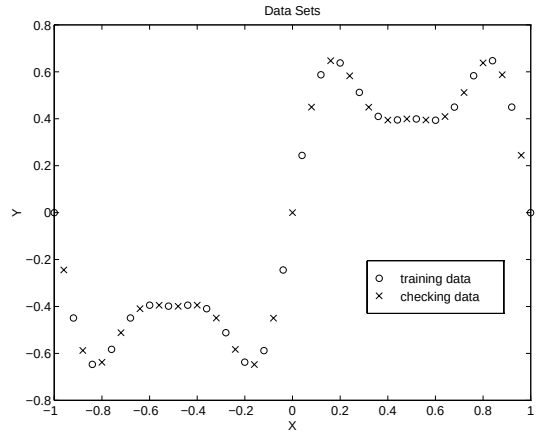
To start ANFIS learning, first you need to have a *training data set* that contains desired input/output data pairs of the target system to be modeled. Sometimes you also want to have an optional *checking data set* that can check the generalization capability of the resulting fuzzy inference system. Usually these data sets are collected based on observations of the target system and

then stored in separate files. Suppose the data sets are generated via the following MATLAB commands:

```
% Number of total data pairs
numPts = 51;
x = linspace(–1,1,numPts)';
y = 0.6*sin(pi*x) + 0.3*sin(3*pi*x) + 0.1*sin(5*pi*x);
data = [x y]; % total data set
trnData = data(1:2:numPts,:); % training data set
chkData = data(2:2:numPts,:); % checking data set
```

Now plot the data set.

```
plot(trnData(:,1),trnData(:,2),'o', ...
                    chkData(:,1),chkData(:,2),'x')
```



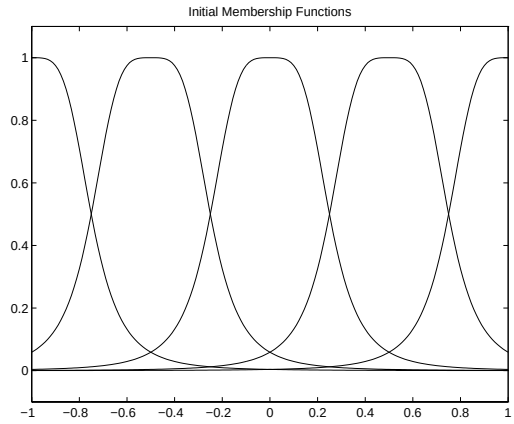
You still need to specify a fuzzy inference system for training. If you have preferable membership functions with specific parameters or shapes, use `fuzzy` to create a fuzzy inference system and store it as a FIS matrix in the workspace. On the other hand, if you do not have any ideas of what the initial membership functions should look like, use the command `genfis1` instead. This command will examine the training data set and then generate a FIS

matrix based on the given numbers and types of membership functions. For example

```
numMFs = 5; % number of MFs
mfType = 'gbellmf'; % MF type is generalized bell
fismat = genfis1(trnData,numMFs,mfType);
```

This generates a FIS matrix called `fismat`. To view the membership functions, type

```
[x,mf]=plotmf(fismat,'input',1);
plot(x,mf)
title('Initial Membership Functions')
```



You can see that `genfis1` places these initial membership functions so they are equally spaced with enough overlap within the input range. To start the training for 40 epochs, invoke the MEX-file `anfis`.

```
numEpochs = 40;
[fismat1,trnErr,ss,fismat2,chkErr] = ...

anfis(trnData,fismat,numEpochs,NaN,chkData);
```

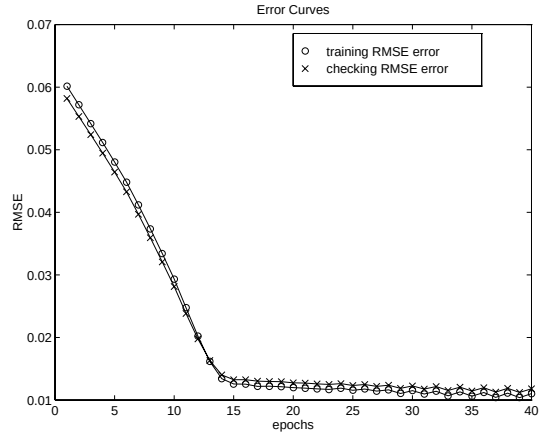
Note that there is a NaN (the IEEE symbol for “not a number”) in the input arguments; it simply acts as a place holder for the display options. When it sees a NaN, `anfis` will take default values for the display options. After you type the above command, information appears in the MATLAB command windows.

After the 40 epochs of batch learning, we can use `evalfis` to verify the learning results:

```
trnOut = evalfis(trnData(:,1),fismat1);
trnRMSE = norm(trnOut – trnData(:,2))/sqrt(length(trnOut));
```

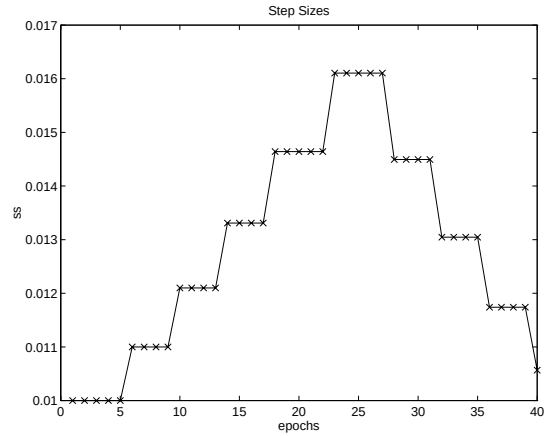
This RMSE (root mean squared error) for training data should match the number appeared on screen after the `anfis` command. Note that the output arguments `fismat1` and `fismat2` are the FIS matrices corresponding to minimal training and checking errors, respectively. To plot error curves, type

```
epoch = 1:numEpochs;
plot(epoch,trnErr,'o',epoch,chkErr,'x')
hold on; plot(epoch,[trnErr chkErr]); hold off
```



To plot step size, type

```
plot(epoch,ss,'-',epoch,ss,'x')
xlabel('epochs'), ylabel('ss'), title('Step Sizes')
```



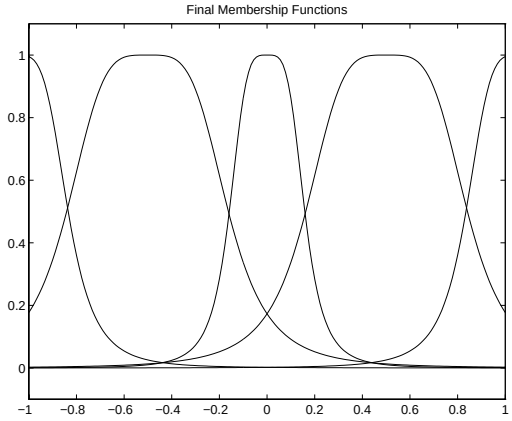
Note that the step size is updated according to the following heuristic guidelines:

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

To plot the final membership functions, type

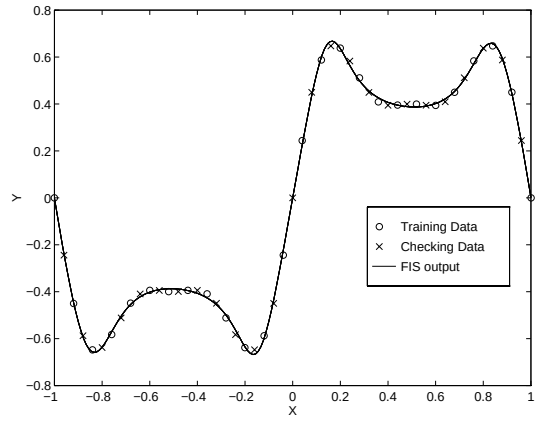
```
[x,mf]=plotmf(fismat1,'input',1);
plot(x,mf)
title('Final Membership Functions');
```



Compare these membership functions with those before training and you will see how the final membership functions are trying to catch the local features of the training data set.

To plot the fuzzy inference system outputs, type

```
anfis_y = evalfis(x,fismat1);
plot(trnData(:,1),trnData(:,2),'o', ...
    chkData(:,1),chkData(:,2),'x', ...
    x,anfis_y,'-');
```



The final result is a good fit for the original data.

More on ANFIS

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

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

where `trnOpt`, `dispOpt`, and `chkData` are optional. All the output arguments are also optional.

Training Data

The training data `trnDat` is a required argument to `anfis`. Each row of `trnData` is a desired input/output pair of the target system to be modeled; it 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 the number of columns is equal to the number of inputs plus one.

Input FIS Matrix

The input FIS matrix `fismat` can be obtained either from the FIS Editor (where you have to specify all the details) or `genfis1` (where you only need to give numbers and types of membership functions). This FIS matrix contains both the structure (which specifies number of rules in the FIS, number of membership functions for each input, etc.) and parameters (which specify the shapes of membership functions). Remember that ANFIS learning employs the gradient descent for updating membership function parameters, so the learning process will drop into a local minimum if it finds one. Therefore the more the initial membership functions resemble the optimal ones, the more likely the training will converge to the optimal point in the parameter space. Human expertise about the target system to be modeled can help when setting up these initial membership function parameters in the FIS matrix.

Note that `genfis1` produces a FIS matrix with a grid partition and it causes an explosion of the number of rules when the number of input is moderately large, that is, more than four or five. This *curse of dimensionality* is inherent to all fuzzy inference systems with a grid partition. To get around this, an alternative is to generate a FIS matrix with a scatter partition. This can be done using the clustering algorithm discussed in the next chapter.

ANFIS applies the least-squares method to identify the consequent parameters (the coefficients of the output equations of each rules) at each epoch, so the initial values of consequent parameters in `fismat` are not used in the learning process at all.

Training Options

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

- `trnOpt(1)`: epoch number, default 10.
- `trnOpt(2)`: 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.

If any element of `trnOpt` is NaN (Not a Number) 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.

The step-size update strategy was touched on in the early part of this section. Usually we want the step-size profile to be a curve which goes uphill initially, reaches some maximum, and then goes downhill till the end of training. This ideal step-size profile is achieved by adjusting the initial step-size and the increase and decrease rates (`trnOpt(3)` to `trnOpt(5)`). The default values are set as the best guess to deal with a wide range of learning tasks. For any specific application, you are encouraged to modify these step-size options in order to find their optimal values.

Display Options

Display option `dispOpt` is a vector of either ones or zeros that specifies what information to display before, during, and after the training process:

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

The default mode is verbose, that is, all available information will be displayed. If any element of `dispOpt` is NaN (not a number) or missing, the default value will be taken.

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 usually distinct from those of the training data.

The checking data is important for learning tasks where the input number is large and/or the data itself is noisy. In general we are not looking for a fuzzy inference system that can best fit the training data. Instead, we are looking for a fuzzy inference system trained on the training data that can respond to the checking data in a satisfactory manner. This cross-validation gives an unbiased estimate of the minimal error measure that can be achieved in the training.

The parameter that corresponds to the minimal checking error is returned in the output argument `fismat2`. This is the output FIS matrix that should be used if the checking data is supplied for the learning.

Output FIS Matrix for Training Data

`fismat1` is the output FIS matrix for minimal training error. This is the FIS matrix that should be used for further calculation if there is no checking data used for cross validation.

Training Error

The training error `trnError` records the RMSE (root mean squared error) for the training data set at each epoch. `fismat1` is the snapshot of the FIS matrix when the training error measure is at its minimum.

Step Size

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.

Output FIS Matrix for Checking Data

`fismat2` is the output FIS matrix for minimal checking error. This is the FIS matrix that should be used for further calculation if there is a checking data used for cross validation.

Checking Error

The checking error `chkError` records the RMSE (root mean squared error) for the checking data at each epoch. `fismat2` is the snapshot of the FIS matrix when the checking error is at its minimum.

Reference

For a detailed discussion of ANFIS architecture, its learning rules and other related issues, you may want to read the paper [Jan93] listed in the "References" section at the end of this chapter.

Fuzzy Clustering

Clustering of numerical data forms the basis of many classification and system modeling algorithms. The purpose of clustering is to distill natural groupings of data from a large data set, producing 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-style fuzzy inference system that models the data behavior.

Fuzzy C-Means Clustering

Fuzzy c-means (FCM) is a data clustering technique where each data point belongs to a cluster to a degree specified by a membership grade. This technique was originally introduced by Jim Bezdek in 1981 [Bez81] as an improvement on earlier clustering methods. The idea is fairly simple: how do you lump together data points that populate some multidimensional space into a specific number of different clusters?

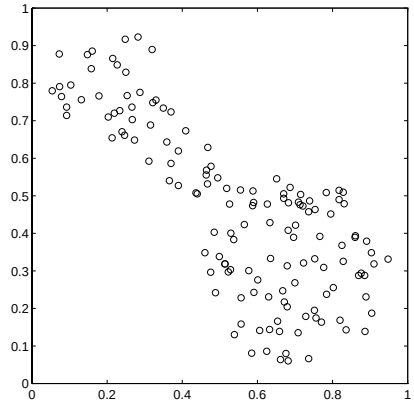
We start with the concept of *cluster centers* that mark the mean location of each cluster. Initially these cluster centers are very inaccurately placed. Additionally, every data point has a membership grade for each cluster. By iteratively updating the cluster centers and the membership grades for each data point, we can watch the cluster centers move to the "right" location. 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.

The final output of fuzzy c-means is not a fuzzy inference system but rather a list of cluster centers and several membership grades for each data point. You can use the information returned by the fuzzy c-means routine to help you build a fuzzy inference system.

An Example: 2-D Clusters

Let's use some quasi-random two-dimensional data to illustrate how fuzzy c-means 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 `fcm` function 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
```

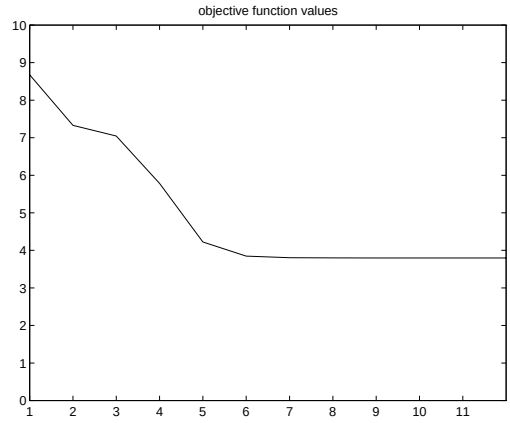
and so on until the objective function is no longer decreasing much at all.

The variable `center` contains the 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.

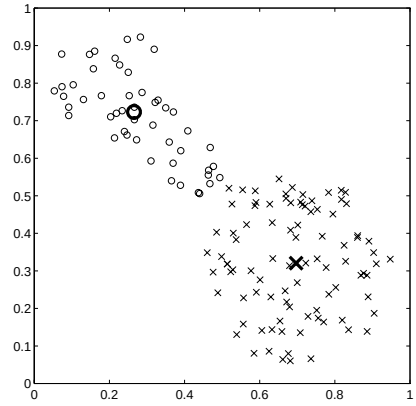
The `fcm` function is actually an iteration loop built on top of several other routines, namely `initfcm`, which initializes the problem; `distfcm`, which is used for distance calculations; and `stepfcm`, which steps through one iteration.

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. Cluster centers are shown by the large characters.



Subtractive Clustering

Suppose we don't even have a clear idea how many clusters there should be for a given set of data. *Subtractive clustering* 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 (like fuzzy c-means) and model identification methods (like ANFIS). The `subclust` function finds the clusters by using the subtractive clustering method.

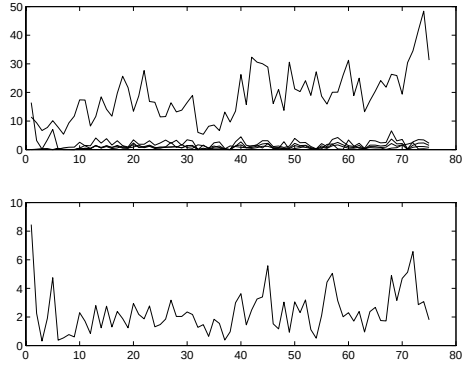
The `genfis2` function builds upon the `subclust` function to provide a fast, one-pass method to take input-output training data and generate a Sugeno-style fuzzy inference system that models the data behavior.

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)
```



Several vectors now exist in the workspace. Of the original 100 data points, we will use 75 as training data (`datin` and `datout`) and 25 as checking data (`chkdatin` and `chkdatout`). The `genfis2` function generates a model from data by 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. A small cluster radius will usually lead to finding many small clusters in the data (resulting in many rules); a large cluster radius will usually lead to finding a few large clusters in the data (resulting in fewer rules). Here we use a cluster radius of 0.5 and run the `genfis2` function.

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

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

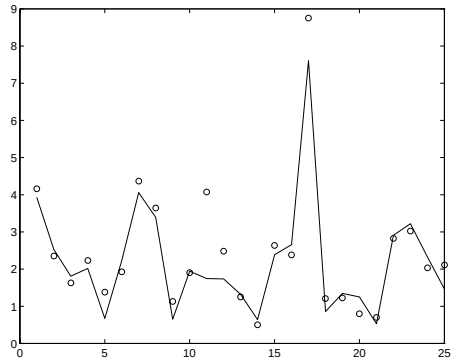
```
fuzout=evalfis(datin,fismat);
trnRMSE=norm(fuzout-datout)/sqrt(length(fuzout))
trnRMSE =
                    0.5276
```

The variable `trnRMSE` is the root mean square error of the system generated by the training data. To check the model, we use the checking data.

```
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 checking data. A plot of the checking 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.

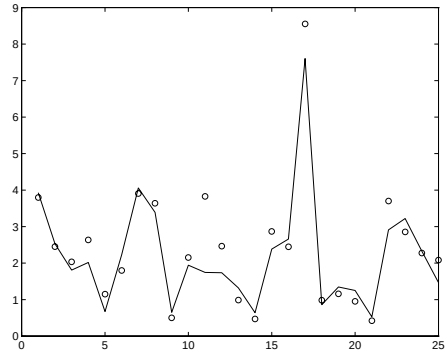
```
fismat2=anfis([datin datout],fismat,[50 0 0.1]);
```

Messages go by as the training progresses, after which we can 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
```

So the model has improved a lot with respect to the training data, and a little with respect to the checking data. Here is a plot of the improved checking 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 form a good "scatter" partition of the input space, in contrast to a grid partition of the input space. This overcomes the problem with combinatorial explosion of rules when the input data has high dimension (the dreaded curse of dimensionality).

Overfitting

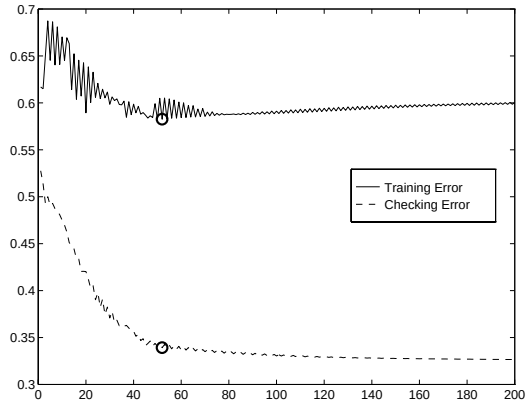
Now let's go on to consider what happens if we continue to exhaustively train this system using the ANFIS algorithm.

```
[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 RMS error versus training data, and the RMS error versus 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 like good news. 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 at all. 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.



This is indeed a case of overfitting. The smallest error against the checking data occurs at epoch 52 after which the checking data error trends upward even as ANFIS keeps working to minimize the error against the training data all the way to epoch 200.

References

The fuzzy c-means algorithm is described in [Bez81], while a full description of the `subclust` algorithm and the underlying clustering method can be found in the paper by Chiu [Chi94]. Both are listed in the "References" section at the end of this chapter.

Stand-Alone Code

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 fuzzy inference engine can read a 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
```

(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
```

It responds with 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 a FIS file that specifies the fuzzy inference system under consideration.

For tutorial purposes, consider the FIS file `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:

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

This will generate 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, 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, you need to change the file `fis.c`, which contains all the necessary functions to perform fuzzy inference process.
- For the Mac, the compiled command `fismain` tries to find `fismain.in` and `fismain.fis` as input data and FIS description file, respectively. The output is stored in a file `fismain.out`. These file names are defined within Mac-specific #define symbols in `fismain.c` and can be changed if necessary.

Applications and Demos

All the demos described in this section can be launched with the help of the demo gateway program `fuzdemos`.

Ball Juggling

Ball juggling is an interesting discrete control problem. In this system, a ball follows a ballistic curve in a two-dimensional plane, while a flat board at ground tries to bounce the ball in order to control the next location of impact. The control goal is to bring the ball to a desired location such that it bounces vertically at the center of the board. We assume that there is no loss of energy in the system; the ball follows a perfect ballistic curve in the air and the collision with the board is elastic. The state equation for this system is

$$x_{k+1} = x_k + \frac{v^2}{g}\sin(2\theta_k + 4u_k)$$

$$\theta_{k+1} = \theta_k + 2u_k$$

where

k : count of impact

x : horizontal location of impact

θ : angle of ball trajectory w.r.t. horizontal axis at time of impact

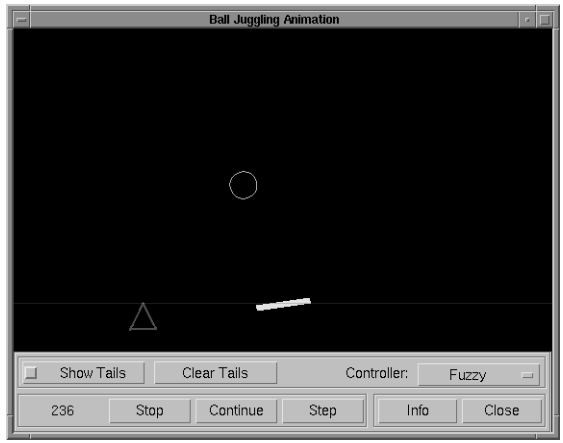
u : angle of board w.r.t. to horizontal axis, this is the control input

g : acceleration of gravity

v : ball velocity at impact

This demo does not require Simulink. To bring up the animation window, type

```
juggler
```



You will see an animation window, with a blue ball jumping up and down, a yellow board where the ball bounces, and a small red triangle indicating the target position. After every eight or nine bounces, the target position will move to a random location automatically, so you can constantly see how the board is controlling the ball. If you want to change the target position directly, you can do so by clicking the small red triangle and drag it to anywhere you like.

It is obvious that the ball can usually reach the target position at the first bounce and the board's angle will become zero right from the second bounce. However, if the target is too far away, the ball may bounce several times before hitting the target.

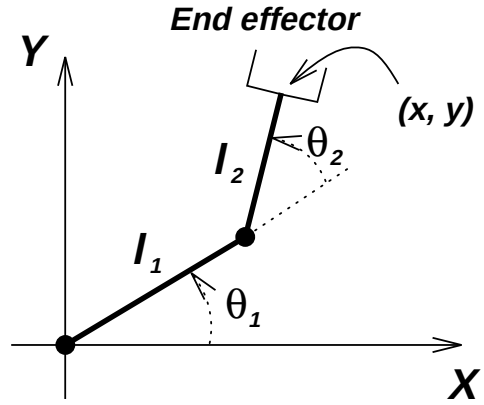
This demo provides an option for a human controller. To try it, set the **Controller** pop-up menu to Human.

Now you can control the tilt angle of the board by clicking at the little steering arrow at the upper right corner. The tilt angle is restricted to any angle between -45 and 45 degrees. If the tilt angle is not set correctly such that the rebounding ball has a negative project angle, then you will see a message like this:

```
Bouncing from the ground, project angle = 1.450950e+02
```

This implies that the ball has a negative project angle after hitting the board and it is bouncing upwards because it also hits the ground.

Inverse Kinematics of Two-Joint Robot Arm



A two-joint planar robot arm, as shown above, is fully specified when the joint angles θ_1 and θ_2 are known. In particular, the Cartesian position (x, y) of the end effector ("hand") can be derived from the joint angles by the following equations:

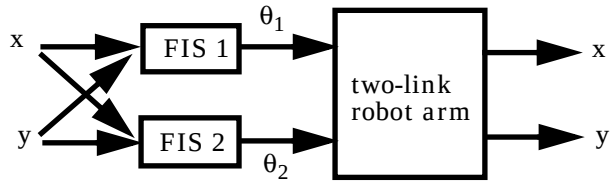
$$x = l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2)$$

$$y = l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2)$$

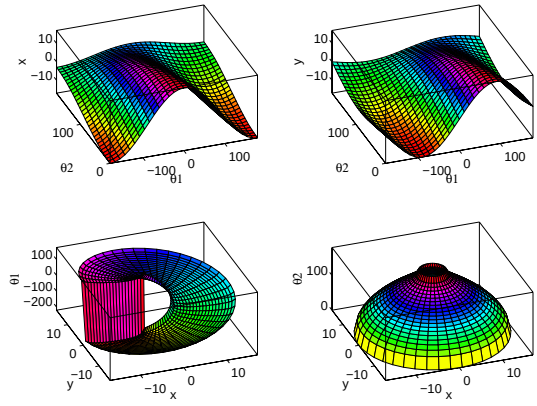
where l_1 and l_2 are the lengths of the rigid arms, respectively. However, in robotic applications, we often face the opposite problem, that is, given the desired position of the end effector, find the corresponding joint angles. This is the so-called *inverse kinematics* problem. This demo will use the adaptive neuro-fuzzy inference system (ANFIS) to solve this kind of problem.

The forward kinematics from the joint angles θ_1 and θ_2 to the end-point Cartesian position (x, y) are quite straightforward, as shown in the above equations. However, the inverse mappings from (x, y) to (θ_1, θ_2) are not too clear. In this particular case, it is possible to solve the inverse mappings algebraically. However, here we assume the solutions are not available and we will train two fuzzy inference systems to approximate these two mappings. In other words, we want to design two fuzzy systems FIS 1 and FIS 2 such that

the overall composite function of the following block diagram is an identity mapping.



Suppose that l_1 is 10, l_2 is 7, and the value of θ_2 is restricted to [0, π]. The following figure demonstrates the mapping surfaces from (θ_1, θ_2) to (x, y) (the first row) and from (x, y) to (θ_1, θ_2) (the second row). These four plots are created by the MATLAB command `invsurf`.

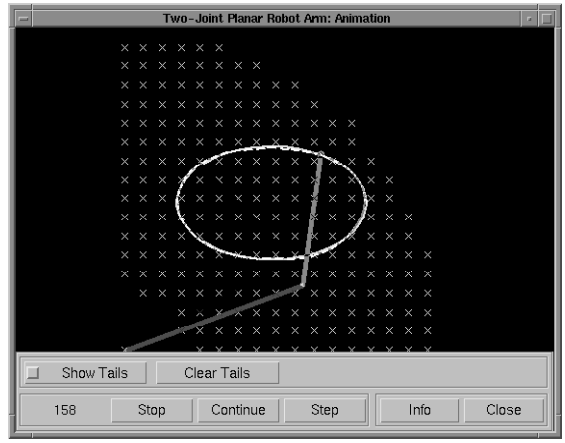


When $\sqrt{x^2 + y^2}$ is greater than l_1+l_2 or less than $|l_1-l_2|$, there is no corresponding (θ_1, θ_2) and the corresponding regions are called *unreachable workspace*. For θ_1 and θ_2 in the unreachable workspace, their values are assigned to NaNs; the effect is shown clearly in the second row of the above plots.

To further simplify our discussion, we assume the end-point position is limited to the first quadrant of the x-y plane. From the first quadrant, we collect 229 training data pairs of the format (x, y: θ_1, θ_2), respectively, for the training of two fuzzy inference systems. We use three membership functions on each input; thus the number of rules is nine and the number of parameters is 45 for each FIS. After 50 epochs of training, the results are stored in two FIS files

`inv1.fis` and `inv2.fis`. To see animation of how well these two fuzzy inference systems work, type

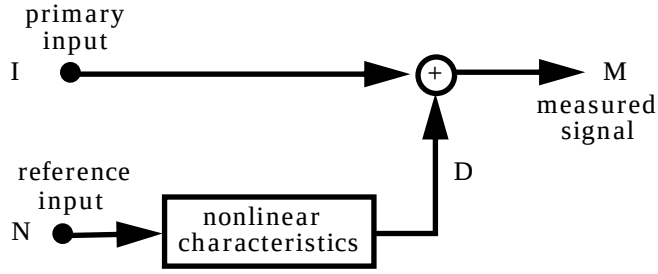
```
invkine
```



In the animation window, an ellipse is chosen as the reference path and the dashed line shows how the end-point follows the desired path based on the inverse mappings. The 229 crosses indicate the locations of the training data. You can even move the ellipse by clicking inside it and dragging it to a new location. As long as the ellipse is inside the region covered by training data, the end-point can follow the path satisfactorily. However, if part or all of the ellipse is out of the region, the end-point will sometimes take a wild trajectory.

This example is only used to demonstrate the concept; the results are not necessarily optimized. Better performance can be obtained through either extensive training or a denser data set. Note that this example corresponds to a case of off-line design of open-loop control; other design approaches can force the end-point to follow the desired trajectory more closely if closed loop control is permitted.

Adaptive Noise Cancellation



Adaptive noise cancellation is one interesting application of ANFIS. The basic situation for adaptive noise cancellation is shown above, where the information signal I comes from the primary input, while the noise source N comes from the reference input. At the receiving end, the measured signal M is equal to the sum of I and D, where D is a distorted version of N due to some nonlinear characteristics f. In symbols,

$M(k) = I(k) + D(k) = I(k) + f(N(k), N((k-1), \ldots))$

Our task is to eliminate D from M and recover the original information signal I.

If the nonlinear characteristic f is known exactly, it would be easy to recover the original information signal by subtracting D from M. However, f is usually unknown in advance and it could be time-varying due to changes in external environments. Moreover, the spectrum of D might overlap that of I, which invalidates the use of filtering techniques in frequency domain.

To estimate the distorted noise signal D, we need to model the nonlinear characteristic f. We use ANFIS to model this nonlinear function. Before training ANFIS, we need to collect training data pairs, but the desired output D is not available since it is combined additively into the measured signal M. Fortunately, we can take M as a contaminated version (which is contaminated by the information signal I) of the desired output and proceed training as usual; the difference between M and D (that is, the information signal I) will hopefully

average out during the training process. Therefore for this scheme to work, the following conditions must hold:

- The noise source N must be available and free of the information signal I.
- The information signal I must be zero-mean and uncorrelated either linearly or nonlinearly with the noise source N.
- The order of the passage characteristics must be known. This determines the number of inputs for ANFIS.

Note that if the passage characteristic is linear, then we can use a linear model instead and the whole setting is the linear adaptive noise canceling proposed by Widrow [WidS85].

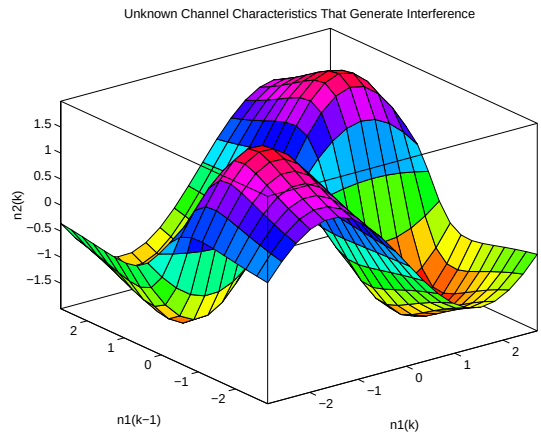
Now we can return to the MATLAB demo. To start the demo, type

```
sshow noisedm
```

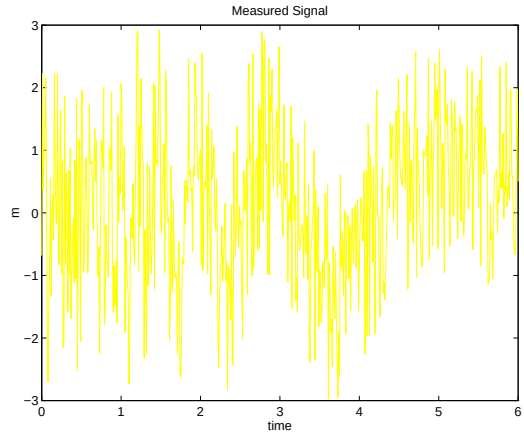
and push the **Start** button when the window opens.

In this demo, we assume the channel characteristic is

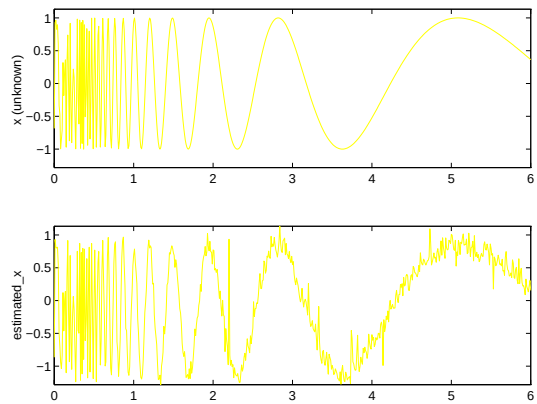
$$D(k) = f(N(k), N(k-1)) = 4*\sin(N(k))*N(k-1)/(1+N^2(k-1))$$



Here is the measured signal M



And we use a 4-rule ANFIS for training 10 epochs on 601 training data pairs. Without extensive training, the ANFIS can already do a fairly good job; the original information signal and the recovered one by ANFIS are shown side by side in the figure below.



Chaotic Time Series Prediction

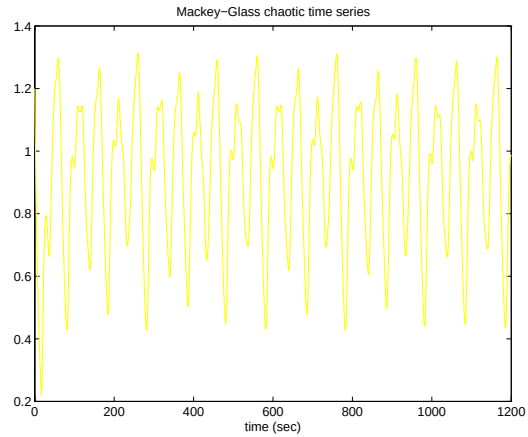
The demo `mgtsdemo` shows how to train an ANFIS for predicting a time series defined by the Mackey-Glass (MG) time-delay differential equation:

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

This turns out to be a chaotic time series without a clearly defined period; it will not converge or diverge, and it is very 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, τ = 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);
```



The task of time series prediction is to use known values of the time series up to the point x = t to predict the value at some point in the future x = t+P. The standard method for this type of prediction is to create a mapping from D points spaced Δ apart, that is, (x(t-(D-1)Δ,..., x(t-Δ), 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 Δ = P = 6. In other words, the training data for ANFIS is of the following format:

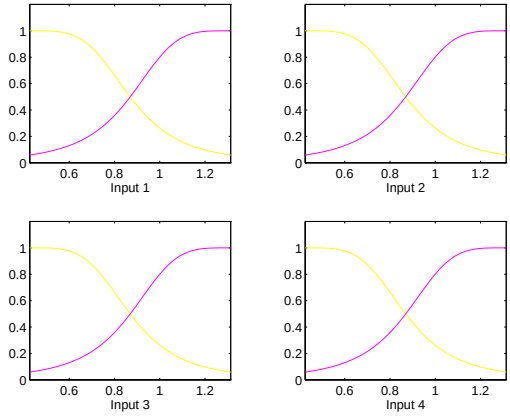
```
[x(t-18) x(t-12) x(t-6) x(t) x(t+6)]
```

From t = 118 to 1117, we can extract 1000 data pairs of the above format. We use the first 500 data pairs for training ANFIS, while the others are used for validating the identified fuzzy model. This results in two data matrices, `trnData` and `chkData`; both are 500-by-5 matrices.

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

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

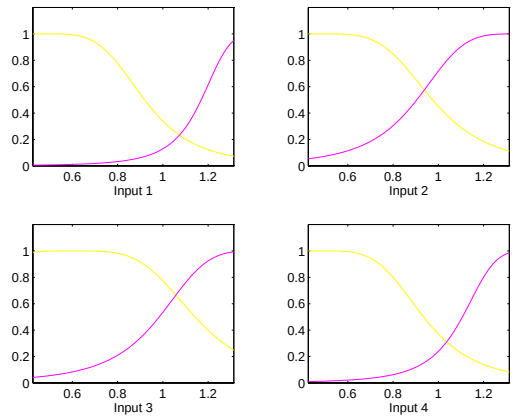
Since we did not specify numbers and types of membership functions used in the FIS, default values are assumed and we have two generalized bell membership functions on each input. The generated FIS matrix contains 2^4 = 16 fuzzy rules with 104 parameters, including 80 linear parameters and 24 nonlinear 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 of fitting parameters. In this case, the ratio between data and parameters is about five (500/104). The function `genfis1` also tries to generate initial membership functions that are equally spaced and cover the whole input space; 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. The membership functions after training, as shown below, do not change drastically. From this, we can guess most of the fitting is done with the linear parameters, while the nonlinear parameters are mostly for fine tuning.



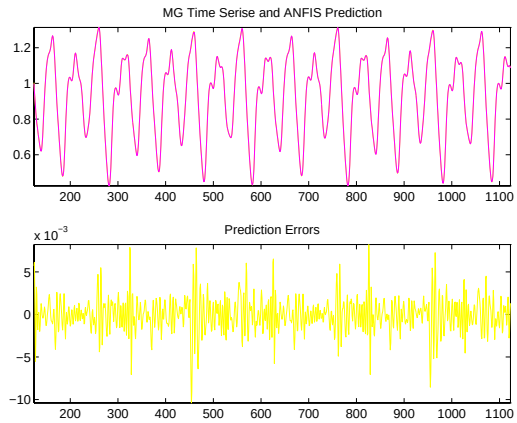
To plot the error curves, type

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

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

To compare the original MG time series and ANFIS prediction side by side, try

```
anfis_output = evalfis([trnData; chkData], fismat1);
index = 125:1124;
subplot(211), plot(time(index), [x(index) anfis_output]);
subplot(212), plot(time(index), x(index) – anfis_output);
```



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

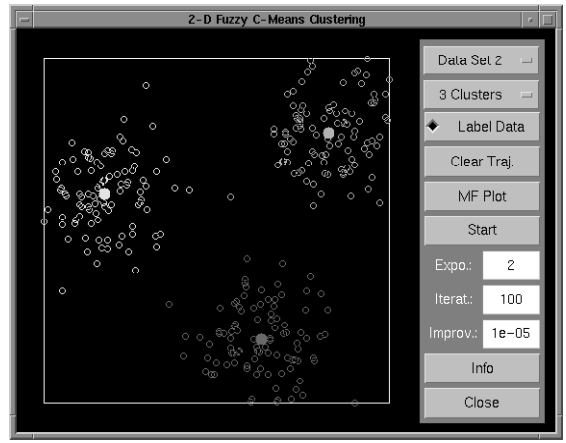
Comparative study shows that because of its sparing use of parameterization, ANFIS has better generalization capability on this problem when compared to auto-regressive models, cascade-correlation neural networks, back-propagation neural networks, radial basis function networks, and other polynomial prediction methods. More details on this respect can be found in Jang's paper listed in the "References" section at the end of this chapter. [Jan93].

Fuzzy C-Means Clustering Demos

Fuzzy c-means (FCM) is a data clustering technique where each data point belongs to a cluster to a degree specified by a membership grade. To try out FCM with 2-D data, type

```
fcmdemo
```

This brings up a window on screen, with a scatter plot of the data set to be clustered and quite a few GUI controls. The default data set obviously falls into three clusters; by clicking **Start**, you see how the three cluster centers move to the "right" positions.



After the clustering process is done, you can click **Clear Traj** to clear the trajectories and get a better view of the cluster centers. You can now click **Start** again to see the repeatability of FCM.

If you set **Label Data**, each data point will have the same color as its cluster center (defined as the cluster with highest membership grade). If **Label Data** is set before the clustering process, you see how clusters are moving and settling; the effect is most pronounced when FCM is applied to data set 4 with four clusters.

Label Data can only let you see the results due to maximal membership grades. To view the membership grade of a cluster, select a cluster (by clicking mouse near a cluster center) and then press **MF Plot**. MATLAB uses the command `griddata` to construct a MF surface on a grid base.

You can select other data sets with different numbers of clusters. Other parameters for FCM includes

Expo.: exponent for membership grades

Iterat.: maximum number of iterations

Improv.: minimum amount of improvement between two iterations

The clustering process stops when the maximum number of iterations is reached, or when the minimum amount of improvement cannot be achieved.

This demo provides a simple and easy way to try out FCM for 2-D data. For data of higher dimensions, usually it's harder to visualize the clustering process. Another simple program that deals with higher-dimensional data is `irisfcm`, which uses FCM to cluster the IRIS data. By typing

irisfcm

at the command line, you can see how the cluster centers move on projected 2-D surfaces.

Note: The remaining demos make use of Simulink. If you do not have access to Simulink, you can still load the `.fis` files associated with these demos and examine the systems using the standard GUI tools, but the animations and simulations illustrated below will not run.

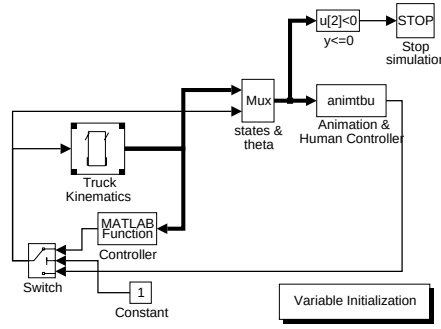
Truck Backer-Upper (Simulink only)

The truck backer-upper (TBU) problem has become a standard problem in the fuzzy logic field. The problem is to design a controller (driver) that can back up a truck into a loading dock from any initial position that has enough clearance from the back wall. The front wheels of the truck can reach any angles between -45 and 45 degrees, but only backing up is allowed; there is no going forward.

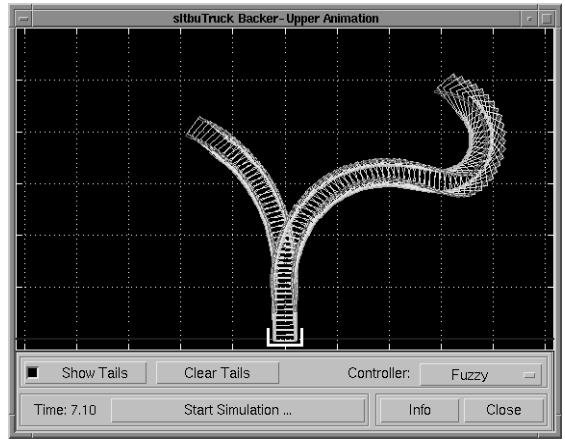
To bring up the Simulink window for this demo, try

```
sltbu
```

A Simulink window will appear on your screen.



Start the simulation by choosing **Start** from the **Simulation** menu. You will see an animation window for the TBU problem, which contains the top view of a small truck, the loading dock indicated as three small circles, a steering handle at the lower right corner, and several UI controls.



You should now see the truck (driven by a fuzzy controller) backing up to the loading dock. The simulation stops whenever the rear end of the truck touches the back wall. To move the truck, click the mouse inside the truck and drag till it reaches a desired location. To rotate the truck, click at any corners of the truck and drag till it has a desired orientation. If you want to revert to the initial conditions for this demo, click the Variable Initialization block in the

Simulink window. Now you can start the simulation from the Simulink window as before, or just click the **Start** button in the animation window.

The default controller is a fuzzy controller. However, you can try to back the truck yourself to see how well you do compared to the fuzzy controller. To do this, set the Controller pop-up menu to Human. Move the truck to a desired initial condition and start the simulation as before. Now you can control the front steering wheel by clicking the mouse on the little steering handle at the lower right corner of the animation window. This type of "human control" is usually not easy at the first shot and requires some practice.

Other UI controls include:

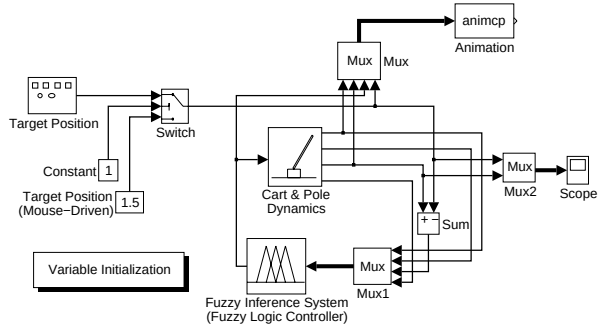
- **Show Trails** to select either to show the trails or not.
- **Clear Trails** to clear animation trails.

Inverted Pendulum (Simulink only)

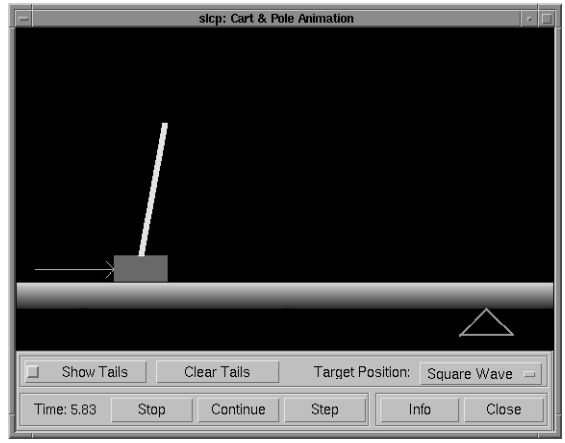
Another standard problem in neuro-fuzzy literature is the inverted pendulum control, also known as the cart-pole (CP) problem. The system under control consists of a rigid pole hinged to a cart through a free joint with one degree of freedom. The cart can be moved to its right or left depending on the force exerted on it. Our task is to design a control that generates appropriate force on the cart such that we can move the cart to a desired position while keeping the pole balanced.

To try the demo, type

```
slcp
```



This brings up the Simulink window for this demo. Start the simulation by choosing **Start** from the **Simulation** menu. Now you can see how the cart is following a desired position of a square wave by a fuzzy controller. The arrow on the cart indicate the magnitude and direction of the exerted force; the triangle is the desired cart position.



This demo actually lets you have five choices for the desired cart position: sinusoid wave, square wave, saw wave, random signal, and mouse driven signal. To change the signal for the target cart position, click the **Target Position** pop-up menu in the animation window and select the one you are interested in. You can pause the simulation by clicking the **Pause** button, after which you can either continue (**Continue** button) or single-step (**Step** button) through the simulation. Note that both **Continue** and **Step** buttons are hidden under the **Pause** button, which means you will not be able to do single-stepping of the simulation until you pause it first.

Other UI controls include:

- **Show trails** to show trails of animation.
- **Clear trails** to clear trails.

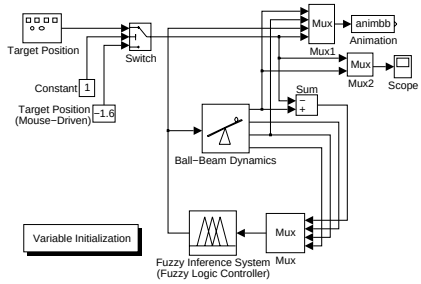
The FIS matrix for the fuzzy controller of this demo is specified in the file `slcp.fis`.

Ball and Beam (Simulink only)

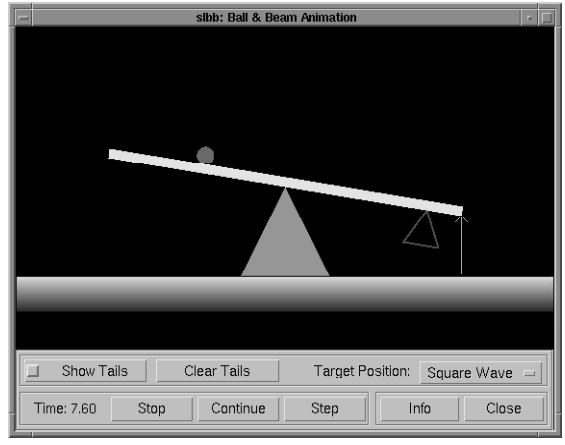
The ball-beam (BB) system consists of a ball rolling frictionlessly on a beam; a motor generates a torque to tilt the beam in order to send the ball to a desired location. A fuzzy controller is designed to generate an appropriate torque to achieve the control goal.

To start the demo, type

```
slbb
```



This brings up the Simulink window for this demo. Start the simulation by choosing **Start** from the **Simulation** menu. Now you can see how the ball is following a desired position of a square wave by a fuzzy controller. The arrow on the tips of the beam indicates the magnitude and direction of the exerted torque; the small hollow triangle is the desired ball position.



The GUI layout of the animation window is almost the same as that of the cart-pole demo. Again we have five signals for desired cart positions. If the desired position is mouse-driven, you can click mouse inside the small triangle and drag it to a desired location. The simulation controls for this demo are very similar to the ones used in the Inverted Pendulum demo.

The FIS matrix for the fuzzy controller of this demo is specified in the file `slbb.fis`.

Glossary

This section is designed to briefly explain some of the specialized terms that appear when discussing fuzzy logic.

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

ANFIS - (Adaptive Neuro-Fuzzy Inference System) 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 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 where 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-style FIS.

Mamdani-style 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-style 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.

[Jan95] Jang, J.-S. R. and C.-T. Sun, "Neuro-Fuzzy and Soft Computing," 1995, (submitted for publication).

[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-part 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.

Fuzzy Musings

"It was the best of times, it was the worst of times..."

—from *A Tale of Two Cities* by Charles Dickens.

Dickens' famous story about the French revolution begins with enough contradictions to make a logician weep. How can great literature fly so rashly in the face of Aristotle? Let's rephrase the first statement in a more mathematical format to make the inconsistency as glaring as possible. Dickens is describing a setting in which

(times == best) ***and*** *(times == worst)*

That is, the book begins at the intersection of two mutually exclusive sets. Can such a long book be devoted entirely to the empty set? The resolution must lie somewhere else. Dickens is using words that can be construed by a rigid literalist as quasi-mathematical in meaning, but obviously it would be foolish to do so. He's not referring to the realm of Platonic forms, he's referring to the real world, full of dirt, sweat, and vagueness. And yet, these words do bear some relationship to their stricter mathematical cousins. They must, else where would the mathematical terms have come from? So what's going on here?

If we approach our lexical analysis of the first sentence with fuzzy reasoning in mind, suddenly we find there is room for a book after all. The intersection of good times and bad times is not necessarily empty. On the other hand, if Dickens had truly been using two-valued logic, the result would have been empty and the book would have gone unwritten. In other words, fuzzy logic permitted the novel to exist!

This may all sound like facile word play, but there is a serious point to be made here. If we really want to analyze mathematically the language that we use every day, the language that lets us grapple with complexity and the full range of human experience, we can be sure that two-valued Aristotelian logic will let us down. Syllogisms like "All fish swim, salmon are fish, therefore salmon swim" will only get you so far. Fuzzy logic brings more of human experience to bear, and therefore is more useful in translating what we know about the world into useful engineering.

3

Reference

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

GUI Tools

Function	Purpose
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.
psigmf	Product of two sigmoid 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 diagram.
plotmf	Display all membership functions for one variable.
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 Sugeno-type FIS (MEX only).
`fcm`	Find clusters with fuzzy c-means clustering.
`genfis1`	Generate FIS matrix using generic method.
`genfis2`	Generate FIS matrix using subtractive clustering.
`subclust`	Find cluster centers with subtractive clustering.

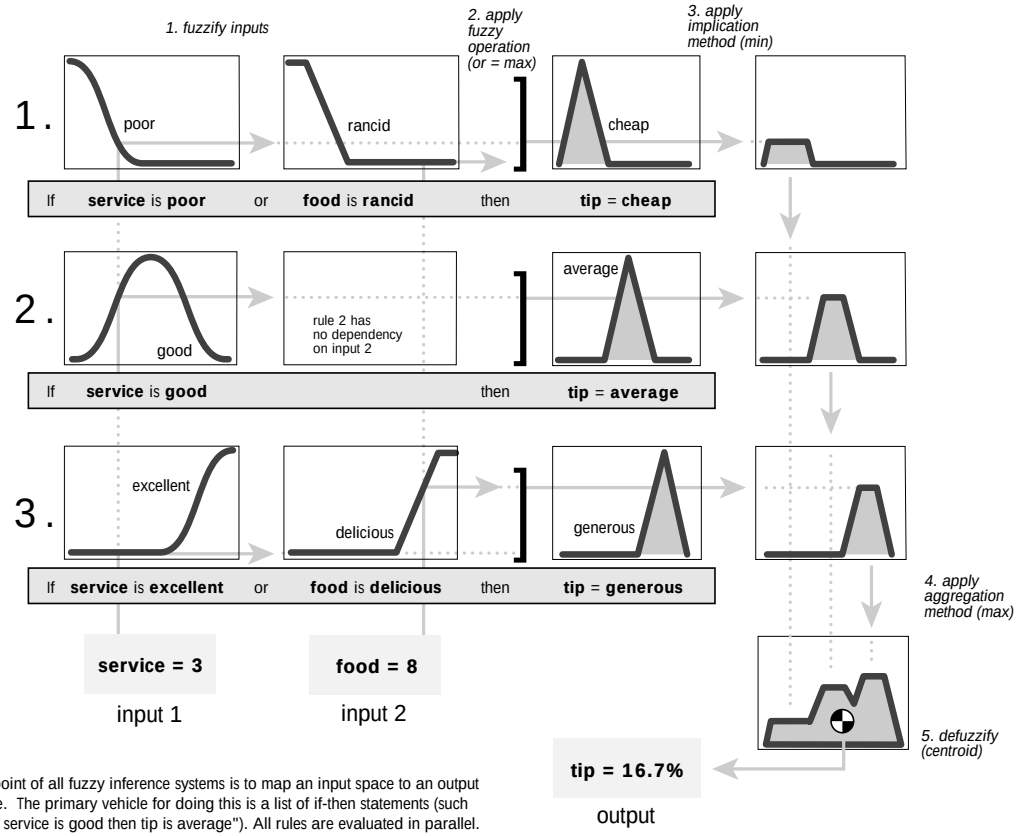
Simulink Blocks

Function	Purpose
`fuzblock`	Fuzzy logic controller block.
`sffis`	Fuzzy inference S-function.

Demos

Function	Purpose
`defuzzdm`	Defuzzification methods.
`fcmdemo`	Fuzzy c-means clustering demo (2-D).
`fuzdemos`	List of all Fuzzy Logic Toolbox demos.
`invkine`	Inverse kinematics of a robot arm.
`irisfcm`	Fuzzy c-means clustering demo (4-D).
`noisedm`	Adaptive noise cancellation.
`slbb`	Ball and beam control (Simulink only).
`slcp`	Inverted pendulum control (Simulink only).
`sltank`	Water level control (Simulink only).
`sltbu`	Truck backer-upper (Simulink only).

Fuzzy Inference Quick Reference



The point of all fuzzy inference systems is to map an input space to an output space. The primary vehicle for doing this is a list of if-then statements (such as "if service is good then tip is average"). All rules are evaluated in parallel. Shown above is the basic structure of a fuzzy inference system. There are five distinct parts to the process:

1. Fuzzify the inputs. (service is good)
Fuzzification is the process of assigning a degree of truth (between 0 = FALSE and 1 = TRUE) to statements about the input variables (all those statements in the IF part, or antecedent, of the rule). The membership functions associated with the input variables determine this degree of truth. Any statement in the antecedent evaluates to a number between 0 and 1.

2. Apply the fuzzy operator (service is poor or food is rancid)
If the antecedent is made up of multiple statements joined by connectives (AND or OR), then the fuzzy operator resolves the overall antecedent based on the connective used. The fuzzy operator always resolves a multiple statement antecedent into a number between 0 and 1.

3. Apply the implication operator (then tip = cheap)
The consequent, or THEN part of the rule, is a shape defined by the area under the output variable membership function curve. Whereas the antecedent statement is a mapping from a single input value to a single truth value, the consequent statement is the assignment of an entire fuzzy set to the output variable. The value (between 0 and 1) of the antecedent truncates or shapes the fuzzy set specified in the consequent by means of the implication operator.

4. Aggregate the output across all rules
Steps 1, 2, and 3 occur for all rules, so each rule has a fuzzy set to contribute to each output. Joining all these sets into a single output membership function is known as aggregation and it is mediated by the aggregation operator.

5. Defuzzify the aggregate output fuzzy set
The aggregate membership function for each output variable must be reduced to a single value. The defuzzification function returns this value given the sometimes oddly shaped aggregate.

Purpose

Add membership function to FIS.

Synopsis

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

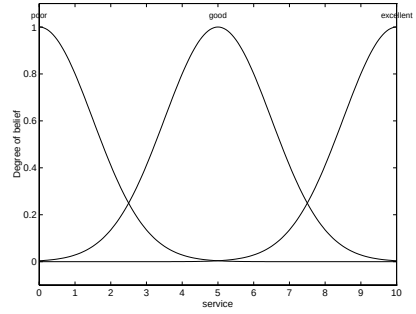
Description

A membership function can only be added to a variable that is already part of the system. You cannot add a membership function to input variable number two of a system if only one input has been defined. Membership functions are given indices 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.

Examples

The function requires six input arguments. Here is an example of how it might be used:

```
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 rule to FIS.

Synopsis

```
a = addrule(a,rulelist)
```

Description

The variable `rulelist` can be a list of one or more rows, each of which represents a given rule. The format that the rule list 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.

Examples

```
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 variable to FIS.

Synopsis

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

Description

Variables are given indices 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.

Examples

```
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`

anfis

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)
```

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 the least-squares method and the backpropagation gradient descent for linear and nonlinear parameters, respectively.

If no checking data is involved, `anfis` can be invoked with from one to four input arguments and it returns three output arguments:

```
                          [fismat1,error,stepsize] = ...
    anfis(trnData,fismat,trnOpt,dispOpt)
```

`trnData` is a training data matrix, where each row is a desired input-output data pair, with output at the last column.

`fismat` is a FIS matrix that specifies the structure and initial parameters for training. This FIS matrix can be generated from data directly using the command `genfis1`. If `fismat` is a single number or a vector, it is taken as the number of membership functions. Then both `trnData` and `fismat` are passed to `genfis1` to generate a valid FIS matrix before starting the training process.

`trnOpt` is a training option vector which specifies various options during training:

- `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)

If any element of `trnOpt` is `NaN` (not a number), then the default value is used. Default values can be changed directly by modifying this file. If `trnOpt` itself is missing, a null matrix, or a `NaN`, then it takes the default values.

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

The step size is decreased (by multiplying it with the 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.

`dispOpt` is a display options vector which specifies what message to display in the MATLAB command window during training:

`dispOpt(1)`: ANFIS information, such as numbers of linear and nonlinear parameters, and so on (default: 1)

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

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

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

The parsing rule of `dispOpt` is the same as `trnOpt`.

`fismat1` is the FIS matrix, which corresponds to the minimum training error. `error` is an array of root mean squared errors. `stepsize` is an array of step sizes.

If checking data is involved in the training process, then `anfis` should be invoked with five input arguments and it returns five output arguments:

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

Here none of `fismat`, `trnOpt` and `dispOpt` can be omitted. If the default values of `trnOpt` and/or `dispOpt` are taken, they should be specified either as NaNs or empty matrices. The additional input argument `chkData` specifies the checking data matrix; its format is the same as `trnData`.

`fismat1` is the FIS matrix that corresponds to the minimum training error. `error1` is an array of root mean squared training errors. `stepsize` is an array of step sizes. `fismat2` is the FIS matrix that corresponds to the minimum checking error. `error2` is an array of root mean squared checking errors.

Examples

```
x = (0:0.1:10)';
y = sin(2*x)./exp(x/5);
trnData = [x y];
numMFs = 5;
mfType = 'gbellmf';
epoch_n = 20;
in_fismat = genfis(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

Defuzzify membership function.

Synopsis

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

Description

`defuzz(x,mf,type)` returns a defuzzified value of `mf` positioned at `x`, using different defuzzification strategies. 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

Difference of two sigmoid membership functions.

Synopsis

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

Description

The sigmoid curve depends on two parameters *a* and *c* given by

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

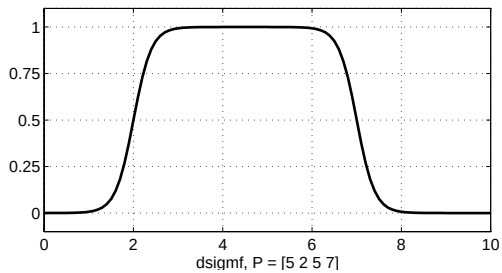
This function is simply the difference between two such curves

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

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

Examples

```
x=0:0.1:10;
y=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 calculation.

Synopsis

```
output = evalfis(input,fismat)
```

Description

This function computes the output vector `output` of the fuzzy inference system specified by the FIS matrix `fismat`. The function `evalfis` exists as both an M-file and a MEX-file. The MEX-file version is always used preferentially (if available) because of its speed advantage.

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 `output`, where each row is a output vector and L is the number of output variables.

Example

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

which generates the response

```
out =
   7.0169
   19.6810
```

See Also

`ruleview`, `gensurf`

evalmf

Purpose

Generic membership function evaluation.

Synopsis

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

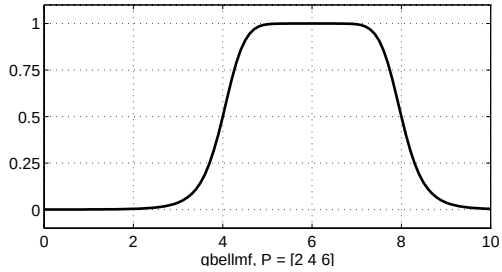
Description

As long as `mfType` is a legal membership function, and `mfParams` are appropriate parameters for that function, `evalmf` will evaluate any membership function.

If you want to create your own custom membership function, `evalmf` will still work, because it will "`eval`" the name of any membership function 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`

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. Input and output 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)

`center`: final cluster centers, where each row is a center

`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 `options` to control clustering parameters, stopping criteria, and/or 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 (not a number), the default value for that option is used instead. The clustering process stops when the maximum number of iteration is reached, or when the objective function improvement between two consecutive iteration 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','*','color','g');
line(data(index2,1),data(index2,2), ...
   'linestyle','*','color','r');
```

fuzblock

Purpose

Simulink fuzzy logic controller block.

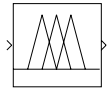
Synopsis

```
fuzblock
```

Description

This command brings up a Simulink system that contains exactly one block, the fuzzy logic controller. The dialog box associated with this block (found by double-clicking on the block) should contain the name of the FIS matrix in the workspace that corresponds to the desired fuzzy system.

If the fuzzy inference system has multiple inputs, these inputs should be multiplexed together before feeding them into the fuzzy controller block. Similarly, if the system has multiple outputs, these signals will be passed out of the block on one multiplexed line.



Fuzzy Logic Controller

See Also

`sffis`

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 the several Fuzzy Logic Toolbox demos, including the pole and cart demo, the truck backing demo, and others. The demos are all described in detail in Chapter 2, Tutorial.

fuzzy

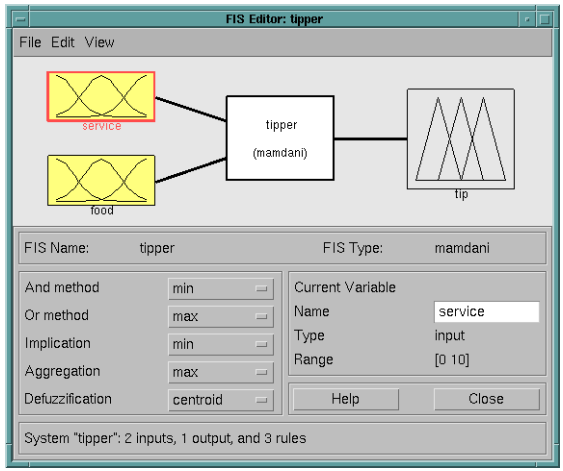
Purpose

Basic FIS editor.

Synopsis

```
fuzzy
fuzzy(fismat)
```

April
1997



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 Chapter 2, Tutorial, for more information about how to use `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 system. 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, which allows you to open related GUI tools, open and save systems, and so on.

- **File**

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

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

  Open from disk... Loads a system from a specified `.fis` file on disk.

  Save to disk Saves the current system to a `.fis` file on disk.

  Save to disk as... Saves the current system to disk with the option to rename or relocate the file.

  Open from workspace... Load a system from a specified FIS matrix variable in the workspace.

  Save to workspace... Saves the system to the currently named FIS matrix variable in the workspace.

  Save to workspace as... Saves the system to a specified FIS matrix variable in the workspace.

  Close window.

- **Edit**

  Add input Add another input to the current system.

  Add output Add another output to the current system.

  Remove variable Delete the current variable.

  Undo Undo the most recent change.

- **View**

  Edit MFs... Invoke the Membership Function Editor.

  Edit rules... Invoke the Rule Editor.

  View rules... Invoke the Rule Viewer.

  View output surface... 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 a custom operation.

Or method Choose max, probor (probabilistic or), or a custom operation.

Implication method Choose min, prod, or a custom operation. This selection is not available for Sugeno-style fuzzy inference.

Aggregation method Choose max, sum, probor, or 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 a custom operation. For Sugeno-style inference, choose between wtaver (weighted average) or wtsum (weighted sum).

See Also

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

Purpose

Two-sided Gaussian curve membership function.

Synopsis

```
y = gauss2mf(x,params)
y = gauss2mf(x,[sig1 c1 sig2 c2])
```

Description

The gaussian curve depends on two parameters *sig* and *c* as given by

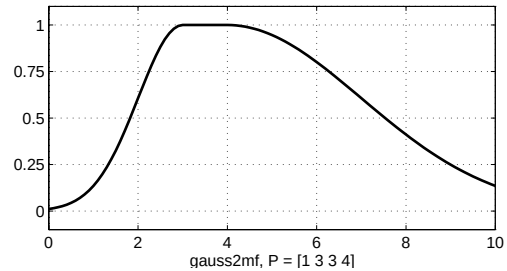
$$f(x;\sigma,c) = e^{\frac{-(x-c)^2}{2\sigma^2}}$$

The function `gauss2mf` is just a combination of two such curves. The first curve should be the leftmost curve. The region between *c1* and *c2* is constrained to be equal to 1. The parameters are listed in the order:

[*sig1*, *c1*, *sig2*, *c2*], *c1* < *c2*.

Examples

```
x=0:0.1:10;
y=gauss2mf(x,[1 3 3 4]);
plot(x,y)
xlabel('gauss2mf, P=[1 3 3 4]')
```



See Also

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

gaussmf

Purpose

Gaussian curve membership function.

Synopsis

```
y = gaussmf(x,params)
y = gaussmf(x,[sig c])
```

Description

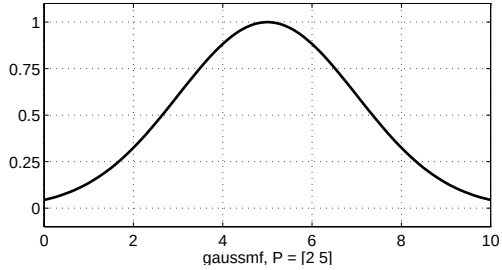
The gaussian curve depends on two parameters *sig* and *c* as given by

$$f(x;\sigma,c) = e^{\frac{-(x-c)^2}{2\sigma^2}}$$

The parameters are listed in the order: [*sig*, *c*].

Examples

```
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 curve membership function.

Synopsis

```
y = gbellmf(x,params)
y = gbellmf(x,[a b c])
```

Description

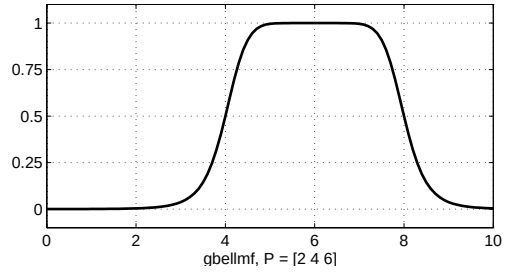
The generalized bell curve depends on three paramters *a*, *b*, and *c* as given by

$$f(x;a,b,c) = \frac{1}{1+\left|\frac{x-c}{a}\right|^{2b}}$$

where the parameter *b* is usually positive. The parameter *c* locates the center of the curve.

Examples

```
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 FIS matrix using generic method.

Synopsis

```
fismat = genfis1(data)
fismat = genfis1(data,numMFs,mfType)
```

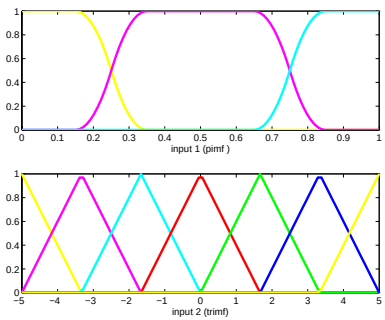
Description

`genfis1(data,numMFs,mfType)` generates a FIS matrix from training data `data`, using grid partition style. `numMFs` is a vector specifying the number of membership functions on all inputs. `mfType` is a string array where each row specifies the membership function type of an input variable.

If `numMFs` is a number and/or `mfType` is a single string, they will be used for all inputs. Default value for `numMFs` is 2; default string for `mfType` is '`gbellmf`'.

Examples

```
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 FIS matrix using subtractive clustering.

Synopsis

```
fismat = genfis2(Xin,Xout,radii,xBounds,options)
```

Description

Given a set of input and output data, this function extracts 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 a FIS matrix that contains the resultant fuzzy rulebase. `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 2 columns and `Xout` has 1 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 2xN matrix that specifies how to map the data in `Xin` and `Xout` into a unit hyperbox, where N is the data dimension. 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 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 the `subclust` function.

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`

Purpose

Generate FIS output surface.

Synopsis

```
gensurf(fis)
gensurf(fis,inputs,output)
gensurf(fis,inputs,output,grids,refinput)
```

Description

`gensurf(fis)` will generate a plot of the output surface of a fuzzy system using the first two inputs and the first output.

`gensurf(fis,inputs,output)` will generate a plot using the inputs (one or two) and output (only one is allowed) given by the vector `inputs` and the scalar `output`.

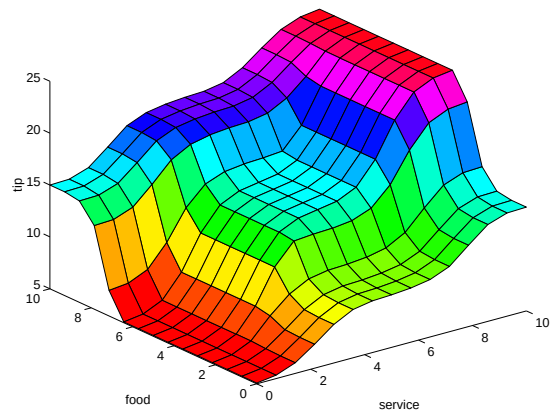
`gensurf(fis,inputs,output,grids)` allows you to specify the number of grids in the X and Y directions. If `grids` is a two element vector, the grids in the X and Y directions can be set independently.

`gensurf(fis,inputs,output,grids,refinput)` can be used if there are more than two outputs. `refinput` then specifies the nonvarying inputs to the system.

`[x,y,z]=gensurf(...)` returns the variables that define the output surface and suppresses automatic plotting.

Examples

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



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 matrix. With this one function you can learn about every part of the fuzzy inference system.

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`

mf2mf

Purpose

Translate parameters between functions.

Synopsis

```
outParams = mf2mf(inParams,inType,outType)
```

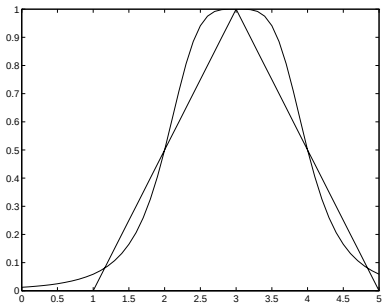
Description

This function does its best to translate parameters among the various membership function types. Occasionally this translation will result 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 function tries to match the $\mu = 0.5$ crossover points for both the new and old membership functions.

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`

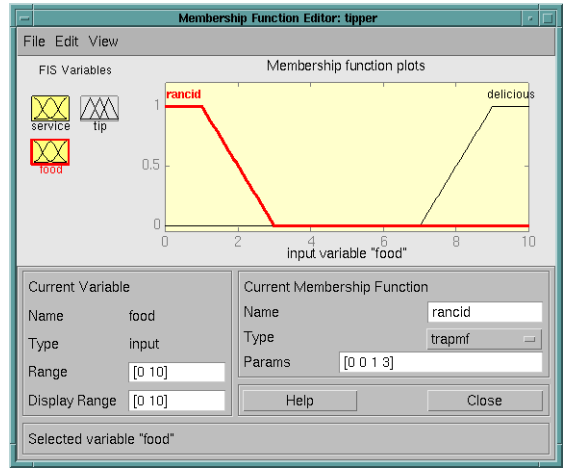
Purpose

Membership function editor.

Synopsis

```
mfedit(a)
```

Description



The Membership Function Editor allows you to inspect and modify all the membership functions in your fuzzy system. For each membership function you can change the name, the type, and the parameters. Eleven basic membership functions are provided for you to choose from, although of course you can always create your own specialized versions. Refer to Chapter 2, Tutorial, for more information about how to use `mfedit`.

The Diagram

Select the current variable with the Variable Palette on the left side of the diagram (under the heading “FIS Variables”). 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 the Reference entry `fuzzy` for more information.

- **Edit**

  Add MF... Add membership functions to the current variable.

  Add custom MF... Add a customized membership function to the current variable.

  Remove current MF Delete the current membership function.

  Remove all MFs Delete all membership functions of the current variable.

  Undo Undo the most recent change.

- **View**

  Edit FIS properties... Invoke the FIS Editor.

  Edit rules... Invoke the Rule Editor.

  View rules... Invoke the Rule Viewer.

  View output surface... 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. In general, any membership function can be converted to any other. Customized membership functions, however, can never be converted.

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 matrices. `newfis` has up to seven input arguments, and the output argument is a FIS matrix. The seven input arguments correspond to: name, type, AND method, OR method, implication method, aggregation method, and defuzzification method.

Examples

The following example shows what the defaults are for each of the methods:

```
>> a=newfis('newsys');
>> getfis(a)
        Name = newsys
        Type = mamdani
        NumInputs = 0
        InLabels =
        NumOutputs = 0
        OutLabels =
        NumRules   0
        AndMethod   min
        OrMethod   max
        ImpMethod   min
        AggMethod   max
        DefuzzMethod   centroid
ans =
     []
```

See Also

`readfis`, `writefis`

parsrule

Purpose

Parse fuzzy rules.

Synopsis

```
fis2 = parsrule(fis,txtRuleList,ruleFormat)
```

Description

This function parses the text that defines the rules for a fuzzy system and returns a FIS matrix with the appropriate rule list in place. If the original input matrix `fis` has any rules initially, they are replaced in the new matrix `fis2`. Three different rule formats are supported: verbose, symbolic, and indexed.

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

Pi-shaped curve membership function.

Synopsis

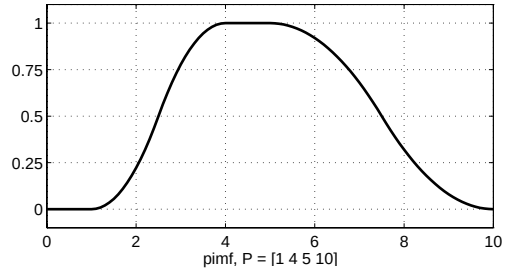
```
y = pimf(x,params)
y = pimf(x,[a b c d])
```

Description

This spline-based curve is so named because of its shape. 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 fuzzy inference system.

Synopsis

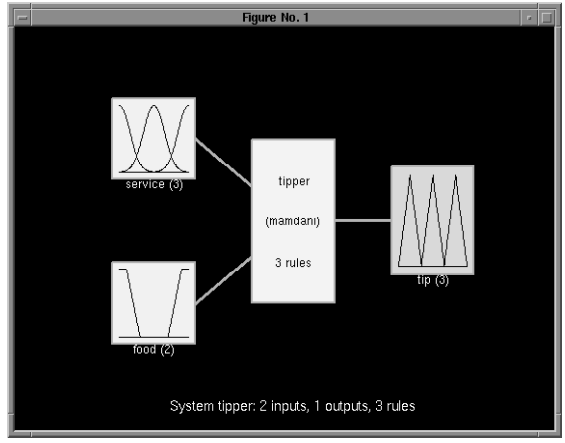
```
plotfis(fismat)
```

Description

This function displays a high level diagram of a fuzzy inference system. Inputs and their membership functions are shown on the left and outputs and their membership functions are shown on the right.

Examples

```
a = readfis('tipper')
plotfis(a)
```



See Also

`plotfis`, `evalmf`

Purpose

Plot membership functions for a variable.

Synopsis

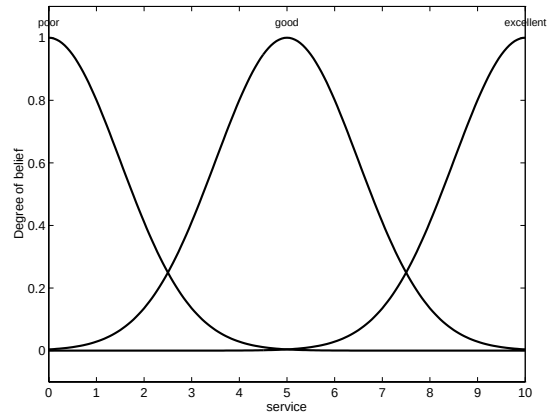
```
plotmf(fismat,varType,varIndex)
```

Description

This function plots all of the membership functions associated with a given variable.

Examples

```
a = readfis('tipper')
plotmf(a,'input',1)
```



See Also

`evalmf`, `plotfis`

psigmf

Purpose

Product of two sigmoid curves membership functions.

Synopsis

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

Description

The sigmoid curve depends on two parameters *a* and *c* as given by

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

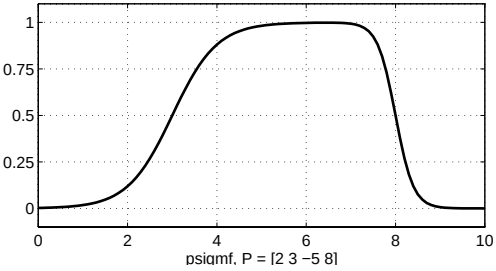
This function is simply the product or two such curves

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

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 FIS from disk.

Synopsis

```
fismat = readfis('filename')
```

Description

Read a fuzzy inference system from a `.fis` file on 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.

The extension `.fis` is assumed for *filename* if it is not already present.

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
```

See Also

`writefis`

rmmf

Purpose

Remove membership function from FIS.

Synopsis

```
a = rmmf(a,'varType',varIndex,'mf',mfIndex)
```

Description

For removing membership functions. You cannot remove a membership function currently in use in the rule list.

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)
        Name = temperature
        NumMFs = 1
        MFLabels =
                cold
        Range = [0 100]
b = rmmf(a,'input',1,'mf',1);
getfis(b,'input',1)
        Name = temperature
        NumMFs = 0
        MFLabels =
        Range = [0 100]
```

See Also

`addmf`, `addrule`, `addvar`, `rmvar`

Purpose

Remove variable from FIS.

Synopsis

```
rmvar(a,'varType',varIndex)
```

Description

For removing fuzzy variables. You cannot remove a fuzzy variable currently in use in the rule list. This command will automatically alter the rule list to keep its size consistent with the current number of variables.

Examples

```
a = newfis('mysys');
a = addvar(a,'input','temperature',[0 100]);
getfis(a)
        Name = mysys
        Type = mamdani
        NumInputs = 1
        InLabels =
                temperature
        NumOutputs = 0
        OutLabels =
        NumRules = 0

b = rmvar(a,'input',1);
getfis(b)
        Name = mysys
        Type = mamdani
        NumInputs = 0
        InLabels =
        NumOutputs = 0
        OutLabels =
        NumRules = 0
```

See Also

`addmf`, `addrule`, `addvar`, `rmmf`

ruleedit

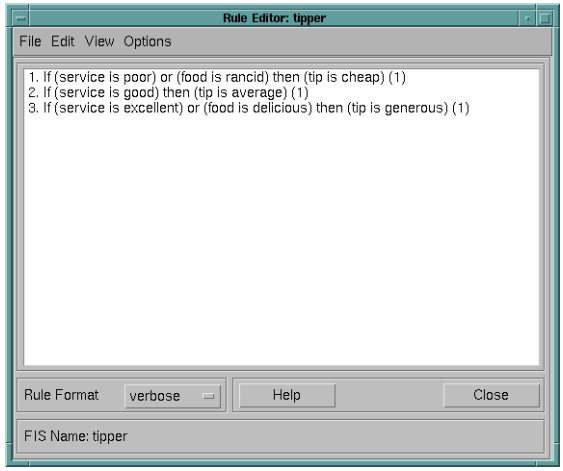
Purpose

Rule editor and parser.

Synopsis

`ruleedit(a)`

Description



The Rule Editor, like the Membership Function Editor, is used to modify the FIS matrix. It can also be used simply to inspect the current rules being used by a system. In general, you simply type your rules into the text field, and when you're ready to parse the rules press **Ctrl-Return**. Refer to Chapter 2, Tutorial, for more information about how to use `ruleedit`.

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 the Reference entry `fuzzy` for more information.

- **Edit**

  Undo Undo the most recent change.

- **View**

  Edit FIS properties... Invoke the FIS Editor.

  Edit membership functions... Invoke the Membership Function Editor.

  View rules... Invoke the Rule Viewer.

  View output surface... Invoke the Surface Viewer.

Rule Formats Pop-up Menu

There is a pop-up menu in the Rule Editor that allows you to choose which rule display format you prefer. Three different formats are available:

- **verbose**, which uses the words “if” and “then” and so on to create actual sentences.
- **symbolic**, which simply substitutes some symbols for the words described above. For example, “if A and B then C” becomes “A & B => C.”
- **indexed**, which is the simplest of the three formats. Aside from some punctuation used to ease interpretation, this format exactly mirrors how the rule is stored in the FIS matrix.

See the `addrule` and `showrule` commands for more information about the composition of rules.

See Also

`addrule`, `fuzzy`, `mfedit`, `parsrule`, `ruleview`, `showrule`, `surfview`

ruleview

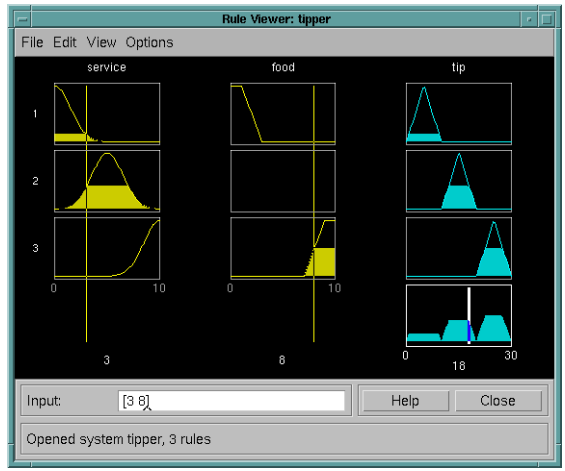
Purpose

Rule viewer and fuzzy inference diagram.

Synopsis

```
ruleview(a)
```

Description



The Rule Viewer is a “living” version of the fuzzy inference diagram. Much like the Surface Viewer, it is a read-only tool. 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 Chapter 2, Tutorial, 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 the Reference entry `fuzzy` for more information.

- **Edit**

  No options under **Edit**.

- **View**

  Edit FIS properties... Invoke the FIS Editor.

  Edit rules... Invoke the Rule Editor.

  View output surface... Invoke the Surface Viewer.

- **Options**

  Rule display format 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. This menu item allows you to set the format in which the rule appears.

See Also

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

Purpose

Set fuzzy system properties.

Synopsis

```
a = setfis(a,'propname',newprop)
a = setfis(a,'vartype',varindex,'propname',newprop)
a = setfis(a,'vartype',varindex,'mf',mfindex, ...
                          'propname',newprop);
```

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 matrix, a particular variable belonging to that FIS matrix, or a particular membership function belonging to one of those variables.

Examples

Called with three arguments

```
a = readfis('tipper');
a2 = setfis(a,'numinputs',3);
getfis(a2,'numinputs')
ans =
   3
```

The following properties of any fuzzy system can be altered with a three argument call to `setfis`: `name`, `type`, `numinputs`, `numoutputs`, `numrules`, `andmethod`, `ormethod`, `impmethod`, `aggmethod`, `defuzzmethod`

If used with five arguments, `setfis` will update any of several variable properties.

```
a2 = setfis(a,'input',1,'name','help');
getfis(a2,'input',1,'name')
ans =
   help
```

The following properties of any fuzzy system can be altered with a five argument call to `setfis`: `name`, `bounds`

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
```

The following properties of any fuzzy system can be altered with a seven argument call to `setfis`: `name`, `type`, `params`

See Also

`getfis`

sffis

Purpose

Fuzzy inference S-function for Simulink.

Synopsis

```
output = sffis(t,x,u,flag,fismat)
```

Description

This MEX-file is used by Simulink to do 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 the simulation which it then continues to use until the simulation is complete.

The input to the fuzzy system comes in through the argument `u`. If, for example, there are two inputs to `fismat` then `u` will be a two element vector.

See Also

`evalfis`, `fuzblock`

Purpose

Display annotated FIS.

Synopsis

```
showfis(fismat)
```

Description

`showfis(fismat)` prints a version of the variable `fismat` annotated row by row, allowing you to see the significance and contents of each row.

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. RuleList         [1 1 1 1 2]
    43.                  [2 0 2 1 1]
    44.                  [3 2 3 1 2]
```

See Also

`getfis`

showrule

Purpose

Display FIS rules.

Synopsis

```
showrule(a,indexList,format)
```

Description

This command is used to display the rules associated with a given system. It can return the rule in any of three different formats: verbose (the default), symbolic, and membership function index referencing. The first argument is the FIS matrix, the second argument is the rule number, and the third argument, if supplied, is the return format. One rule or a vector of rules can be provided to this function.

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`

Purpose

Sigmoid curve membership function.

Synopsis

```
y = sigmf(x,params)
y = sigmf(x,[a c])
```

Description

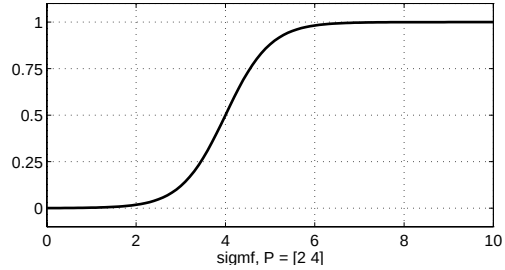
The sigmoid curve depends on two parameters *a* and *c* as given by

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

Depending on the sign of the parameter *a*, a sigmoidal membership function is inherently open right or 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 more on this in the 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`

smf

Purpose

S-curve membership function.

Synopsis

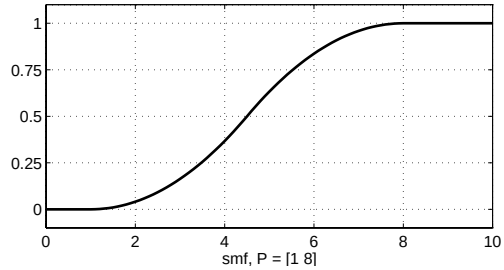
```
y = smf(x,params)
y = smf(x,[a b])
```

Description

This spline-based curve is so named because of its 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`

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 potential for each data point based on the density of surrounding data points. The algorithm selects the data point with the highest potential as the first cluster center and then destroys the potential of data points near the first cluster center. The algorithm then selects the data point with the highest remaining potential as the next cluster center and destroys the potential of data points near this new cluster center. This process of acquiring a new cluster center and destroying the potential of surrounding data points repeats until the potential of all data points falls below a threshold. The subtractive clustering method is an extension of the Mountain clustering method proposed by R. Yager [Yag92].

The matrix `X` contains the data to be clustered; each row of `X` is a data point. The variable `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. 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 2xN matrix that specifies how to map the data in `X` into a unit hyperbox, where N is the data dimension.

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 parameters are:

- options(1) = squashFactor: This is used to multiply the radii values to determine the neighborhood of a cluster center within which the existence of other cluster centers are to be discouraged. (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 very strong potential of 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.

surfview

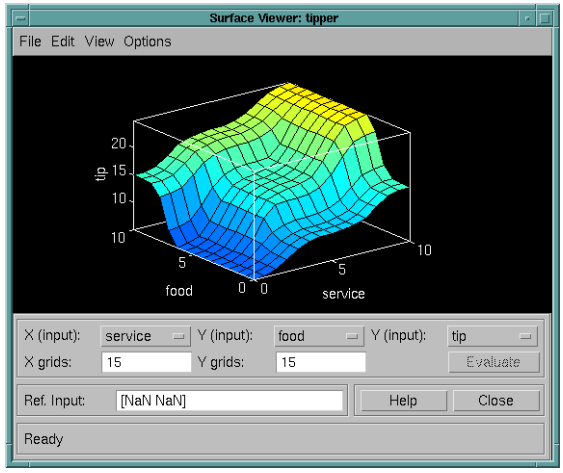
Purpose

Output surface viewer.

Synopsis

```
surfview(a)
```

Description



The Surface Viewer is a GUI tool that lets you examine the output surface of a fuzzy inference system for any one or two inputs. Since it does not alter the fuzzy system or its associated FIS matrix in any way, it is a read-only editor. Using the pop-up menus, you select which input variables you want to form the two input axes (X and Y) as well the output variable that you want to form the output (or Z) axis. Then push the **Evaluate** button to perform the calculation and plot the output surface.

By clicking on the plot axes and dragging the mouse, you can actually 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, in the reference input section the constant values associated with any unspecified inputs.

Refer to the Tutorial section 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 the Reference entry `fuzzy` for more information.

- **Edit**

  No options under Edit.

- **View**

  Edit FIS properties... Invoke the FIS Editor.

  Edit membership functions... Invoke the Membership Function Editor.

  Edit rules... Invoke the Rule Editor.

  View rules... Invoke the Rule Viewer.

- **Options**

  Plot Choose among eight different kinds of plot styles.

  Color Map Choose among several different color schemes.

  Always evaluate Check this menu item if you want 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).

See Also

`fuzzy`, `gensurf`, `mfedit`, `ruleedit`, `ruleview`

trapmf

Purpose

Trapezoidal membership function.

Synopsis

```
y = trapmf(x,params)
y = trapmf(x,[a b c d])
```

Description

The trapezoidal curve depends on four parameters as given by

$$f(x;a,b,c,d)=\begin{cases}0, & x\le a\\ \dfrac{x-a}{b-a}, & a\le x\le b\\ 1, & b\le x\le c\\ \dfrac{d-x}{d-c}, & c\le x\le d\\ 0, & d\le x\end{cases}$$

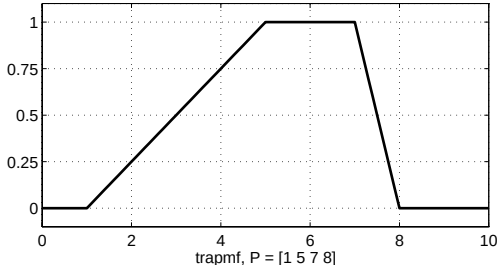
or more compactly by

$$f(x;a,b,c,d)=\max(\min(\frac{x-a}{b-a},1,\frac{d-x}{d-c}),0)$$

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 membership function.

Synopsis

```
y = trimf(x,params)
y = trimf(x,[a b c])
```

Description

The triangular curve depends on three parameters as given by

$$f(x;a,b,c)=\begin{cases}0, & x\le a\\ \dfrac{x-a}{b-a}, & a\le x\le b\\ \dfrac{c-x}{c-b}, & b\le x\le c\\ 0, & c\le x\end{cases}$$

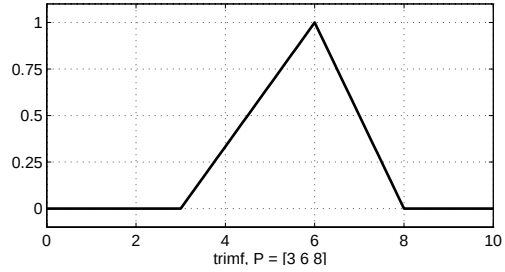
or more compactly by

$$f(x;a,b,c)=\max\left(\min\left(\frac{x-a}{b-a},\frac{c-x}{c-b}\right),0\right)$$

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`

writefis

Purpose

Save FIS to disk.

Synopsis

```
writefis(fismat)
writefis(fismat,filename)
writefis(fismat,filename,'dialog'
```

Description

Save fuzzy inference system as a `.fis` file on disk. `writefis(fismat)` brings up a `uiputfile` dialog box to assist with the naming and directory location of the file.

`writefis(fismat,`*filename*`)` writes a `.fis` file corresponding to the FIS matrix `fismat` to a disk file called *filename*. No dialog box is used and the file is saved to the current directory.

`writefis(fismat,`*filename*`,'dialog')` brings up a `uiputfile` dialog box with the default name *filename* supplied.

The extension `.fis` is added to *filename* if it is not already present.

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`

Purpose

Z-shaped membership function.

Synopsis

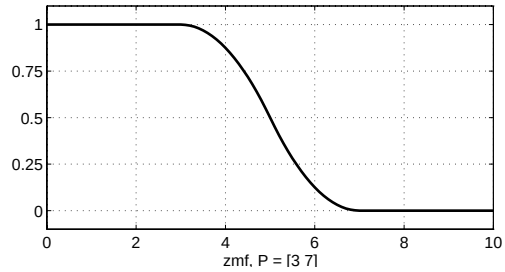
```
y = zmf(x,params)
y = zmf(x,[a b])
```

Description

This spline-based function is so named because of its shape. The parameters *a* and *b* locate the extremes of the sloped portion of the curve.

Examples

```
x=0:0.1:10;
y=trimf(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