

Fuzzy Logic Toolbox™

User's Guide



MATLAB®

R2018b

 MathWorks®

How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

Fuzzy Logic Toolbox™ User's Guide

© COPYRIGHT 1995–2018 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

January 1995	First printing	
April 1997	Second printing	
January 1998	Third printing	
September 2000	Fourth printing	Revised for Version 2 (Release 12)
April 2003	Fifth printing	
June 2004	Online only	Updated for Version 2.1.3 (Release 14)
March 2005	Online only	Updated for Version 2.2.1 (Release 14SP2)
September 2005	Online only	Updated for Version 2.2.2 (Release 14SP3)
March 2006	Online only	Updated for Version 2.2.3 (Release 2006a)
September 2006	Online only	Updated for Version 2.2.4 (Release 2006b)
March 2007	Online only	Updated for Version 2.2.5 (Release 2007a)
September 2007	Online only	Revised for Version 2.2.6 (Release 2007b)
March 2008	Online only	Revised for Version 2.2.7 (Release 2008a)
October 2008	Online only	Revised for Version 2.2.8 (Release 2008b)
March 2009	Online only	Revised for Version 2.2.9 (Release 2009a)
September 2009	Online only	Revised for Version 2.2.10 (Release 2009b)
March 2010	Online only	Revised for Version 2.2.11 (Release 2010a)
September 2010	Online only	Revised for Version 2.2.12 (Release 2010b)
April 2011	Online only	Revised for Version 2.2.13 (Release 2011a)
September 2011	Online only	Revised for Version 2.2.14 (Release 2011b)
March 2012	Online only	Revised for Version 2.2.15 (Release 2012a)
September 2012	Online only	Revised for Version 2.2.16 (Release 2012b)
March 2013	Online only	Revised for Version 2.2.17 (Release 2013a)
September 2013	Online only	Revised for Version 2.2.18 (Release 2013b)
March 2014	Online only	Revised for Version 2.2.19 (Release 2014a)
October 2014	Online only	Revised for Version 2.2.20 (Release 2014b)
March 2015	Online only	Revised for Version 2.2.21 (Release 2015a)
September 2015	Online only	Revised for Version 2.2.22 (Release 2015b)
March 2016	Online only	Revised for Version 2.2.23 (Release 2016a)
September 2016	Online only	Revised for Version 2.2.24 (Release 2016b)
March 2017	Online only	Revised for Version 2.2.25 (Release 2017a)
September 2017	Online only	Revised for Version 2.3 (Release 2017b)
March 2018	Online only	Revised for Version 2.3.1 (Release 2018a)
September 2018	Online only	Revised for Version 2.4 (Release 2018b)

1

Getting Started

Fuzzy Logic Toolbox Product Description	1-2
Key Features	1-2
What Is Fuzzy Logic?	1-3
Description of Fuzzy Logic	1-3
Why Use Fuzzy Logic?	1-6
When Not to Use Fuzzy Logic	1-7
What Can Fuzzy Logic Toolbox Software Do?	1-8
Foundations of Fuzzy Logic	1-10
Overview	1-10
Fuzzy Sets	1-11
Membership Functions	1-14
Logical Operations	1-19
If-Then Rules	1-23
References	1-26
Fuzzy Inference Process	1-28
Fuzzify Inputs	1-29
Apply Fuzzy Operator	1-30
Apply Implication Method	1-31
Aggregate All Outputs	1-31
Defuzzify	1-33
Fuzzy Inference Diagram	1-34
Membership Function Gallery	1-37
Defuzzification Methods	1-40
Fuzzy vs. Nonfuzzy Logic	1-47

Fuzzy Inference System Modeling

2

Types of Fuzzy Inference Systems	2-2
What Is Mamdani-Type Fuzzy Inference?	2-4
What Is Sugeno-Type Fuzzy Inference?	2-5
Comparison of Sugeno and Mamdani Systems	2-12
Advantages of the Sugeno Systems	2-12
Advantages of the Mamdani Systems	2-12
Build Fuzzy Systems Using Fuzzy Logic Designer	2-14
Fuzzy Logic Toolbox Graphical User Interface Tools	2-14
The Basic Tipping Problem	2-16
The Fuzzy Logic Designer	2-17
The Membership Function Editor	2-22
The Rule Editor	2-30
The Rule Viewer	2-33
The Surface Viewer	2-35
Importing and Exporting Fuzzy Inference Systems	2-37
Build Fuzzy Systems at the Command Line	2-38
Build Fuzzy Systems Using Custom Functions	2-50
Build Fuzzy Inference Systems Using Custom Functions in Fuzzy Logic Designer	2-50
Specify Custom Membership Functions	2-51
Specify Custom Inference Functions	2-57
Use Custom Functions in Code Generation	2-63
Fuzzy Logic Image Processing	2-65

Adaptive Neuro-Fuzzy Modeling

3

Neuro-Adaptive Learning and ANFIS	3-2
When to Use Neuro-Adaptive Learning	3-2

Model Learning and Inference Through ANFIS	3-2
References	3-5
Comparison of anfis and Neuro-Fuzzy Designer	
Functionality	3-7
Training Data	3-7
Input FIS Object	3-7
Training Options	3-8
Display Options	3-9
Method	3-9
Output FIS Object for Training Data	3-10
Training Error	3-10
Step-Size	3-10
Checking Data	3-11
Output FIS Object for Checking Data	3-12
Checking Error	3-12
Train Adaptive Neuro-Fuzzy Inference Systems	3-13
Loading, Plotting, and Clearing the Data	3-14
Generating or Loading the Initial FIS Structure	3-15
Training the FIS	3-16
Validating the Trained FIS	3-16
Test Data Against Trained System	3-18
Checking Data Helps Model Validation	3-18
Checking Data Does Not Validate Model	3-29
Save Training Error Data to MATLAB Workspace	3-35
Predict Chaotic Time-Series	3-43
Chaotic Time-Series Prediction	3-51
Modeling Inverse Kinematics in a Robotic Arm	3-61
Adaptive Noise Cancellation Using ANFIS	3-72
Nonlinear System Identification	3-82
Gas Mileage Prediction	3-97

4

Fuzzy Clustering	4-2
What Is Data Clustering?	4-2
Fuzzy C-Means Clustering	4-2
Subtractive Clustering	4-3
References	4-3
Cluster Quasi-Random Data Using Fuzzy C-Means Clustering	4-4
Adjust Fuzzy Overlap in Fuzzy C-Means Clustering	4-8
Fuzzy C-Means Clustering	4-12
Fuzzy C-Means Clustering for Iris Data	4-16
Model Suburban Commuting Using Subtractive Clustering .	4-21
Modeling Traffic Patterns using Subtractive Clustering	4-33
Data Clustering Using Clustering Tool	4-47
Load and Plot Data	4-48
Cluster Data	4-48
Save Cluster Centers	4-50

Fuzzy Logic in Simulink

5

Simulate Fuzzy Inference Systems in Simulink	5-2
Simulate Fuzzy Inference System	5-2
Access Intermediate Fuzzy Inference Results	5-11
Simulation Modes	5-12
Map Command-Line Functionality to Fuzzy Logic Controller Block	5-12
Water Level Control in a Tank	5-15

Temperature Control in a Shower	5-23
Implement Fuzzy PID Controller in Simulink Using Lookup Table	5-30

Deployment

6

Deploy Fuzzy Inference Systems	6-2
Generate Code in Simulink	6-2
Generate Code in MATLAB	6-2
Generate Code for Fuzzy System Using Simulink Coder	6-4
Generate Structured Text for Fuzzy System Using Simulink PLC Coder	6-9
Generate Code for Fuzzy System Using MATLAB Coder	6-12

7 | Apps – Alphabetical List

8 | Functions – Alphabetical List

9 | Objects – Alphabetical List

10 | Blocks – Alphabetical List

A | Bibliography

| Glossary

Getting Started

- “Fuzzy Logic Toolbox Product Description” on page 1-2
- “What Is Fuzzy Logic?” on page 1-3
- “Foundations of Fuzzy Logic” on page 1-10
- “Fuzzy Inference Process” on page 1-28
- “Membership Function Gallery” on page 1-37
- “Defuzzification Methods” on page 1-40
- “Fuzzy vs. Nonfuzzy Logic” on page 1-47

Fuzzy Logic Toolbox Product Description

Design and simulate fuzzy logic systems

Fuzzy Logic Toolbox provides MATLAB® functions, apps, and a Simulink® block for analyzing, designing, and simulating systems based on fuzzy logic. The product guides you through the steps of designing fuzzy inference systems. Functions are provided for many common methods, including fuzzy clustering and adaptive neurofuzzy learning.

The toolbox lets you model complex system behaviors using simple logic rules, and then implement these rules in a fuzzy inference system. You can use it as a stand-alone fuzzy inference engine. Alternatively, you can use fuzzy inference blocks in Simulink and simulate the fuzzy systems within a comprehensive model of the entire dynamic system.

Key Features

- Fuzzy Logic Design app for building fuzzy inference systems and viewing and analyzing results
- Membership functions for creating fuzzy inference systems
- Support for AND, OR, and NOT logic in user-defined rules
- Standard Mamdani and Sugeno-type fuzzy inference systems
- Automated membership function shaping through neuroadaptive and fuzzy clustering learning techniques
- Ability to embed a fuzzy inference system in a Simulink model
- Ability to generate embeddable C code or stand-alone executable fuzzy inference engines

What Is Fuzzy Logic?

Description of Fuzzy Logic

In recent years, the number and variety of applications of fuzzy logic have increased significantly. 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 why use of fuzzy logic has grown, you must first understand 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. However, in a wider sense 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. Even in its more narrow definition, fuzzy logic differs both in concept and substance from traditional multivalued logical systems.

In Fuzzy Logic Toolbox software, fuzzy logic should be interpreted as FL, that is, fuzzy logic in its wide sense. The basic ideas underlying FL are explained in “Foundations of Fuzzy Logic” on page 1-10. 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 Artificial Intelligence (AI), what is missing in such systems is a mechanism for dealing with fuzzy consequents and fuzzy antecedents. In fuzzy logic, this mechanism is provided by 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 the toolbox, it is effectively one of its principal constituents. In most of the applications of fuzzy logic, a fuzzy logic solution is, in reality, a translation of a human solution into FDCL.

A trend that is growing in visibility relates to the use of fuzzy logic in combination with neurocomputing and genetic algorithms. More generally, fuzzy logic, neurocomputing,

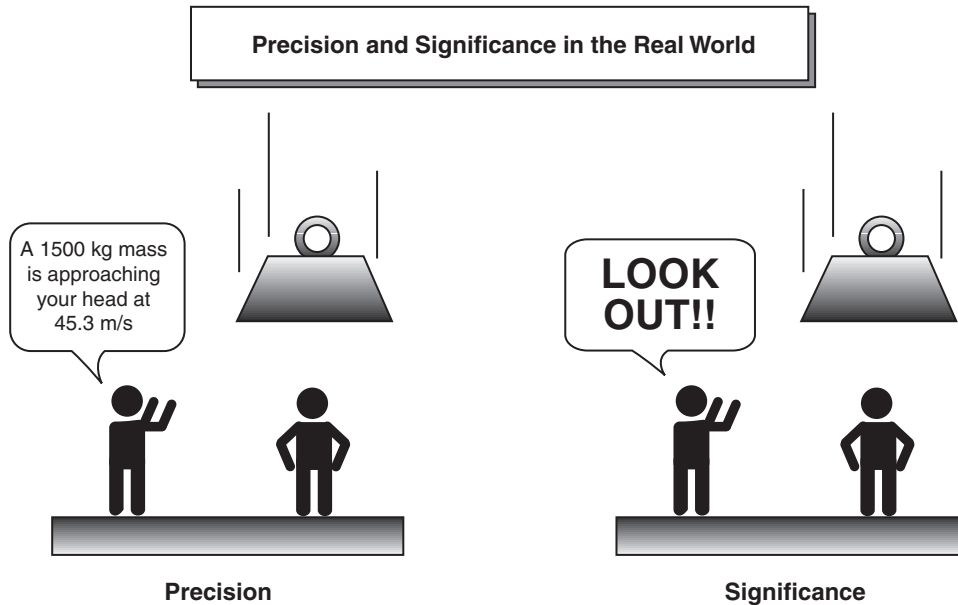
and genetic algorithms may be viewed as the principal constituents of what might be called soft computing. Unlike the traditional, hard computing, *soft computing* accommodates the imprecision of the real world. The guiding principle of soft computing is: Exploit the tolerance for imprecision, uncertainty, and partial truth to achieve tractability, robustness, and low solution cost. In the future, soft computing could 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 that has highest visibility at this juncture is that of fuzzy logic and neurocomputing, leading to 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 toolbox.

Fuzzy logic is all about the relative importance of precision: How important is it to be exactly right when a rough answer will do?

You can use Fuzzy Logic Toolbox software with MATLAB technical computing software as 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.

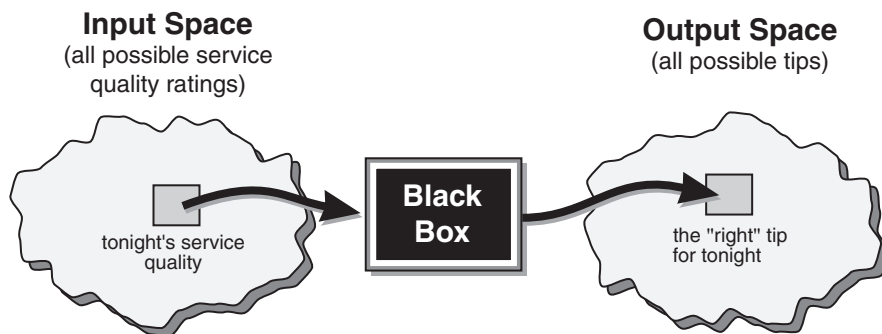
In this sense, fuzzy logic is both old and new because, although the modern and methodical science of fuzzy logic is still young, the concepts of fuzzy logic relies on age-old skills of human reasoning.



Fuzzy logic is a convenient way to map an input space to an output space. Mapping input to output is the starting point for everything. Consider the following examples:

- With information about how good your service was at a restaurant, a fuzzy logic system can tell you what the tip should be.
- With your specification of how hot you want the water, a fuzzy logic system can adjust the faucet valve to the right setting.
- With information about how far away the subject of your photograph is, a fuzzy logic system can focus the lens for you.
- With information about how fast the car is going and how hard the motor is working, a fuzzy logic system can shift gears for you.

A graphical example of an input-output map is shown in the following figure.



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

Determining the appropriate amount of tip requires mapping inputs to the appropriate outputs. Between the input and the output, the preceding figure shows a black box that can contain any number of things: fuzzy systems, linear systems, expert systems, neural networks, differential equations, interpolated multidimensional lookup tables, or even a spiritual advisor, just to name a few of the possible options. Clearly the list could go on and on.

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

Why Use Fuzzy Logic?

Here is a list of general observations about fuzzy logic:

- Fuzzy logic is conceptually easy to understand.

The mathematical concepts behind fuzzy reasoning are very simple. Fuzzy logic is a more intuitive approach without the far-reaching complexity.

- Fuzzy logic is flexible.

With any given system, it is easy to layer on more functionality 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 Adaptive Neuro-Fuzzy Inference Systems (ANFIS), which are available in Fuzzy Logic Toolbox software.

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

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

- Fuzzy logic can be blended with conventional control techniques.

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

- Fuzzy logic is based on natural language.

The basis for fuzzy logic is the basis for human communication. This observation underpins many of the other statements about fuzzy logic. Because fuzzy logic is built on the structures of qualitative description used in everyday language, fuzzy logic is easy to use.

The last statement is perhaps the most important one and deserves more discussion. Natural language, 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.

When Not to Use Fuzzy Logic

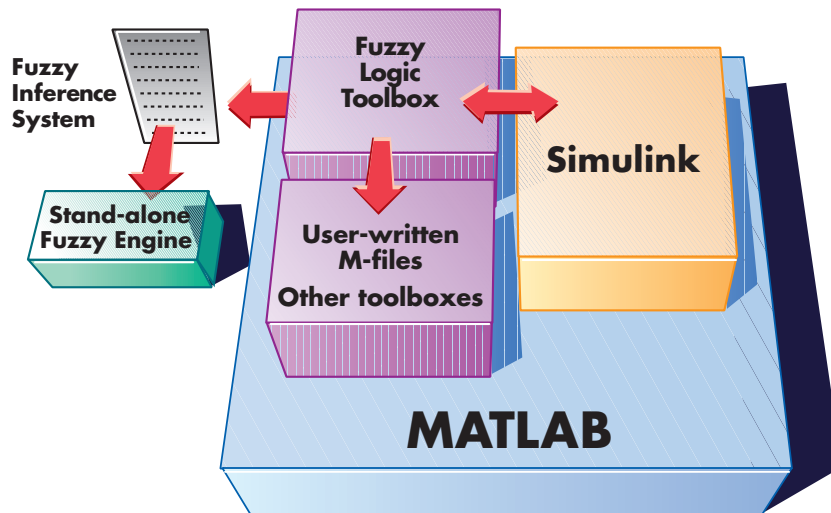
Fuzzy logic is not a cure-all. When should you not use fuzzy logic? The safest statement is the first one made in this introduction: fuzzy logic is a convenient way to map an input space to an output space. If you find it's not convenient, try something else. If a simpler solution already exists, use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

What Can Fuzzy Logic Toolbox Software Do?

You can create and edit fuzzy inference systems with Fuzzy Logic Toolbox software. You can create these systems using graphical tools or command-line functions, or you can generate them automatically using either clustering or adaptive neuro-fuzzy techniques.

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

The toolbox also lets you run your own stand-alone C programs directly. This is made possible by a stand-alone Fuzzy Inference Engine that reads the fuzzy systems saved from a MATLAB session. You can customize the stand-alone engine to build fuzzy inference into your own code. All provided code is ANSI[®] compliant.



Because of the integrated nature of the MATLAB environment, you can create your own tools to customize the toolbox or harness it with another toolbox, such as the Control System Toolbox™, Deep Learning Toolbox™, or Optimization Toolbox™ software.

See Also

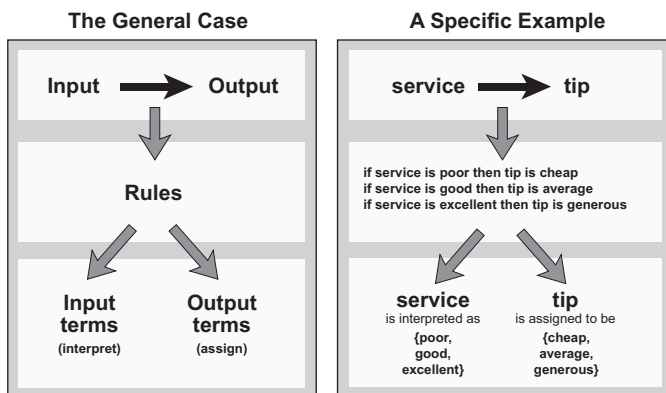
More About

- “Foundations of Fuzzy Logic” on page 1-10
- “Fuzzy vs. Nonfuzzy Logic” on page 1-47

Foundations of Fuzzy Logic

Overview

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 you can build a system that interprets rules, you must define all the terms you plan on using and the adjectives that describe them. To say that the water is hot, you need to define the range that the water's temperature can be expected to vary as well as what we mean by the word *hot*. The following diagram provides a roadmap for the fuzzy inference process. It shows the general description of a fuzzy system on the left and a specific fuzzy system on the right.



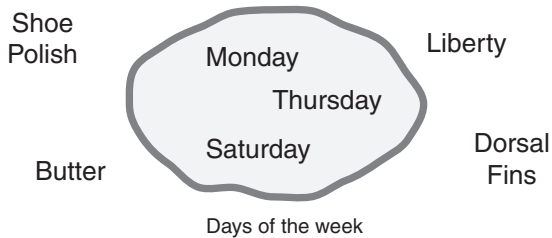
To summarize the concept of fuzzy inference depicted in this figure, *fuzzy inference is a method that interprets the values in the input vector and, based on some set of rules, assigns values to the output vector.*

This topic guides you through the fuzzy logic process step by step by providing an introduction to the theory and practice of fuzzy logic.

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 the definition of 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.

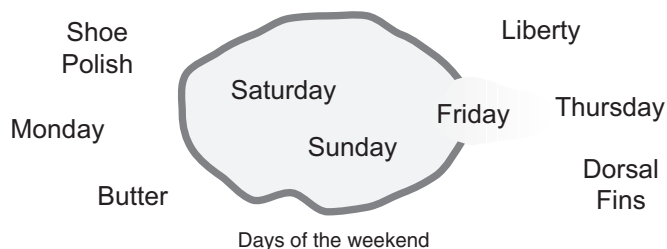


This type of set is called a classical set because it has been around for a long time. It was Aristotle who first formulated the Law of the Excluded Middle, which says X must either be in set A or in set not-A. Another version of this law is:

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

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

Now, consider the set of days comprising a weekend. The following diagram attempts to classify 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. Thus, in the preceding diagram, Friday tries its best to "straddle on the fence." Classical or normal sets would not tolerate this kind of classification. Either something is in or it is out. Human experience suggests something different, however, straddling the fence is part of life.

Of course individual perceptions and cultural background must be taken into account when you define what constitutes the weekend. Even the dictionary is imprecise, defining the weekend as the period from Friday night or Saturday to Monday morning. You are entering the realm where sharp-edged, yes-no logic stops being helpful. Fuzzy reasoning becomes valuable exactly when you work with 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.

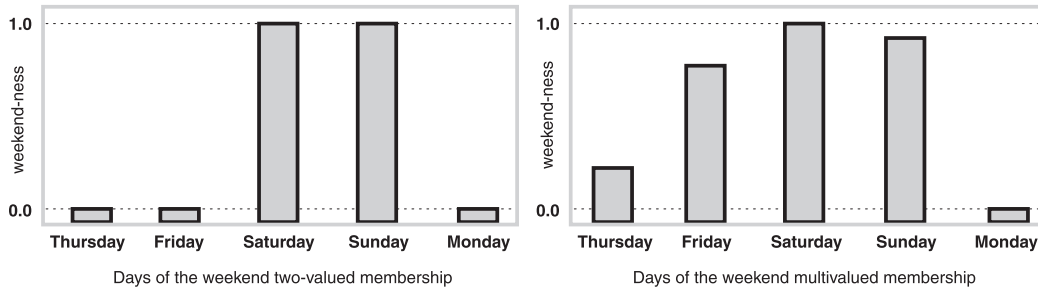
Any statement can be fuzzy. The major advantage that fuzzy reasoning offers is the ability to reply to a yes-no question with a not-quite-yes-or-no answer. Humans do this kind of thing all the time (think how rarely you get a straight answer to a seemingly simple question), but it is a rather new trick for computers.

How does it work? Reasoning in fuzzy logic is just a matter of generalizing the familiar yes-no (Boolean) logic. If you give true the numerical value of 1 and false the numerical value of 0, this value indicates 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)

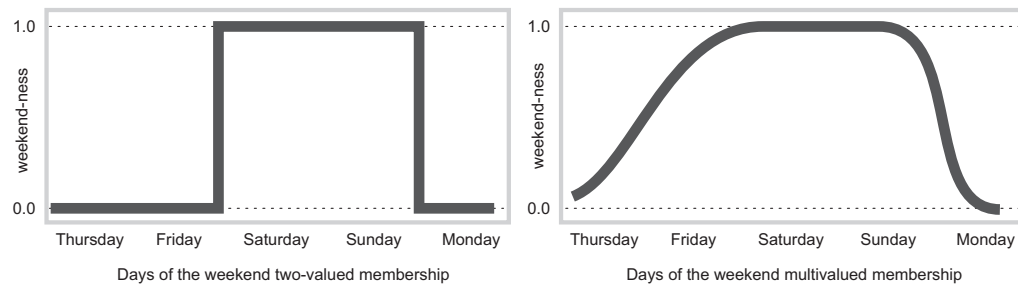
Q: Is Friday a weekend day?
 A: 0.8 (for the most part yes, but not completely)
 Q: Is Sunday a weekend day?
 A: 0.95 (yes, but not quite as much as Saturday).

The following plot on the left shows the truth values for weekend-ness if you 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 you 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 you 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. Thus, 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.

To return to the example, now consider a continuous scale time plot of weekend-ness shown in the following plots.

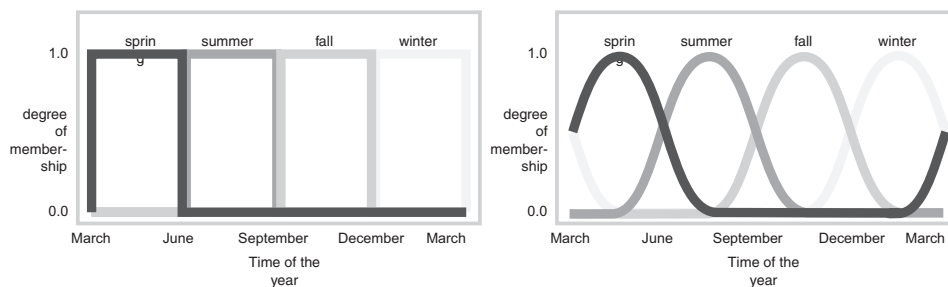


By making the plot continuous, you are defining the degree to which any given instant belongs in the weekend rather than an entire day. In the plot on the left, notice that at midnight on Friday, just as the second hand sweeps past 12, the weekend-ness truth value

jumps discontinuously from 0 to 1. This is one way to define the weekend, and while it may be useful to an accountant, it may not really connect with your own real-world experience of weekend-ness.

The plot on the right shows a smoothly varying curve that accounts for the fact that all of Friday, and, to a small degree, parts of Thursday, partake of the quality of weekend-ness and thus deserve partial membership in the fuzzy set of weekend moments. The curve that defines the weekend-ness of any instant in time is a function that maps the input space (time of the week) to the output space (weekend-ness). Specifically it is known as a *membership function*. See “Membership Functions” on page 1-14 for a more detailed discussion.

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, you get sharp boundaries as shown on the left in the figure that follows. But what you experience as the seasons vary more or less continuously as shown on the right in the following figure (in temperate northern hemisphere climates).

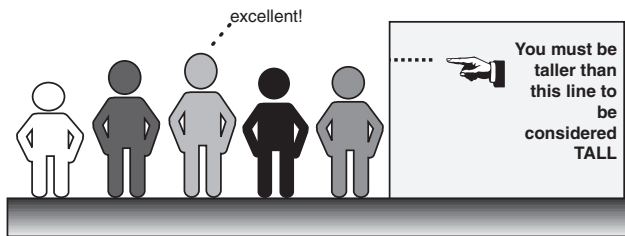


Membership Functions

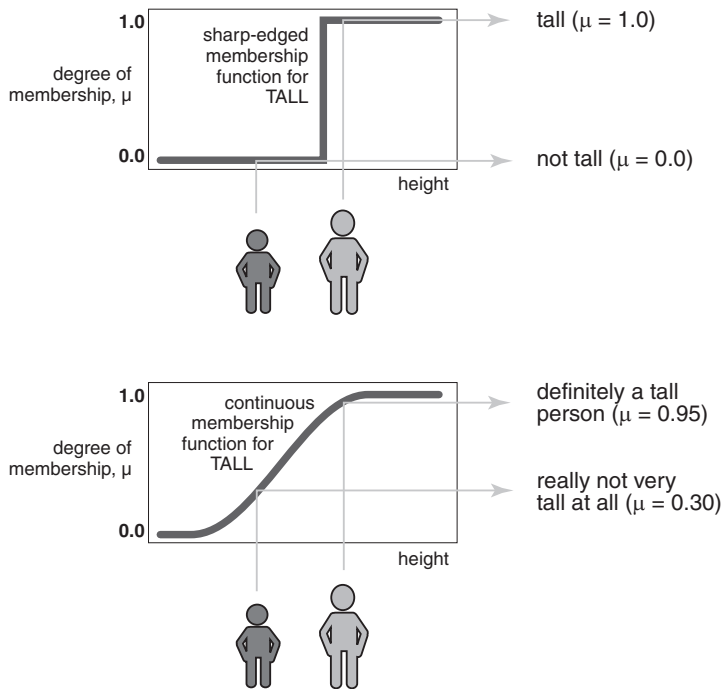
A *membership function* (MF) is a curve that defines how each point in the input space is mapped to a membership value (or degree of membership) between 0 and 1. The input space is sometimes referred to as the *universe of discourse*, a fancy name for a simple concept.

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 three feet to nine 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, you might say all people taller than six feet are officially considered tall. However, 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.



If the kind of distinction shown previously is unworkable, then what is the right way to define the set of tall people? Much as with the plot of weekend days, the figure following 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 you say "She's tall," the membership function tall should already take into account whether you are 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 Fuzzy Logic Toolbox Software

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.

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

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

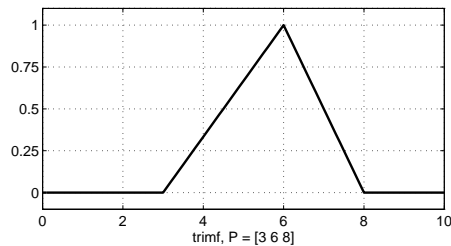
$\mu_A(x)$ is called the membership function (or MF) of x in A . The membership function maps each element of X to a membership value between 0 and 1.

The toolbox includes 11 built-in membership function types. These 11 functions are, in turn, built from several basic functions:

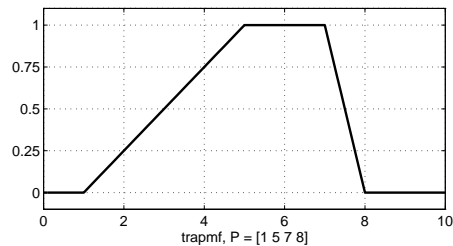
- Piece-wise linear functions
- Gaussian distribution function
- Sigmoid curve
- Quadratic and cubic polynomial curves

For detailed information on any of the membership functions mentioned next, see the corresponding reference page.

The simplest membership functions are formed using straight lines. Of these, the simplest is the *triangular* membership function, and it has the function name `trimf`. This function is 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

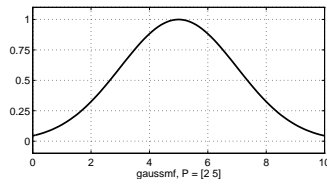


trapmf

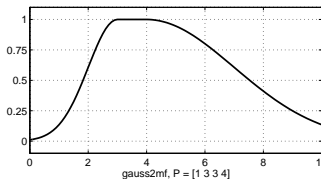
Two membership functions are built on the *Gaussian* distribution curve: a simple Gaussian curve and a two-sided composite of two different Gaussian curves. The two functions are `gaussmf` and `gauss2mf`.

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

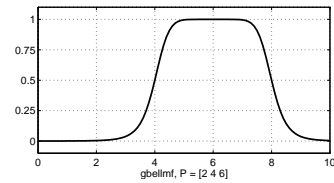
Gaussian membership function, so it can approach a non-fuzzy set if the free parameter is tuned. Because of their smoothness and concise notation, Gaussian and bell membership functions are popular methods for specifying fuzzy sets. Both of these curves have the advantage of being smooth and nonzero at all points.



gaussmf

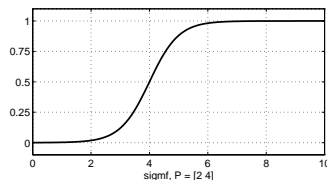


gauss2mf

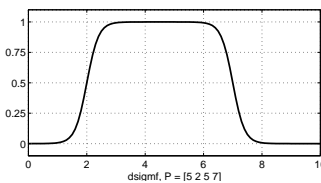


gbellmf

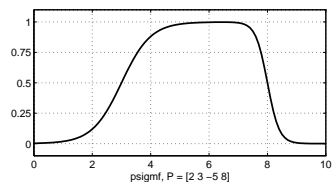
Although the Gaussian membership functions and bell membership functions achieve smoothness, they are unable to specify asymmetric membership functions, which are important in certain applications. Next, you define the *sigmoidal* membership function, which is either open left or right. Asymmetric and 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`, you also have the difference between two sigmoidal functions, `dsigmf`, and the product of two sigmoidal functions `psigmf`.



sigmf

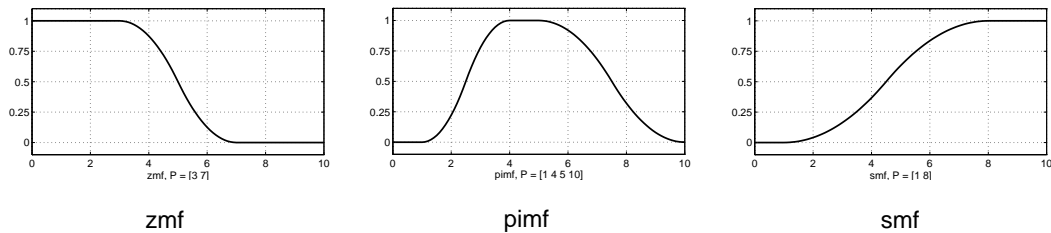


dsigmf



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.



There is a very wide selection to choose from when you're selecting a membership function. You can also create your own membership functions with the toolbox. However, if a list based on expanded membership functions seems too complicated, 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 expansive membership functions are not necessary for good fuzzy inference systems. Finally, remember that more details are available on all these functions in the reference section.

Summary of Membership Functions

- Fuzzy sets describe vague concepts (e.g., fast runner, hot weather, weekend days).
- A fuzzy set admits the possibility of partial membership in it. (e.g., 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. (e.g., 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

Now that you understand the fuzzy inference, you need to see how fuzzy inference connects with logical operations.

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

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, because in fuzzy logic the truth of any statement is a matter of degree, can these truth tables be altered? The input values can be real numbers between 0 and 1. What function preserves 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 $A \text{ AND } B$, where A and B are limited to the range $(0,1)$, by using the function $\min(A,B)$. Using the same reasoning, you can replace the OR operation with the *max* function, so that $A \text{ OR } B$ becomes equivalent to $\max(A,B)$. Finally, the operation NOT A becomes equivalent to the operation $1 - A$. Notice how the previous truth table 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

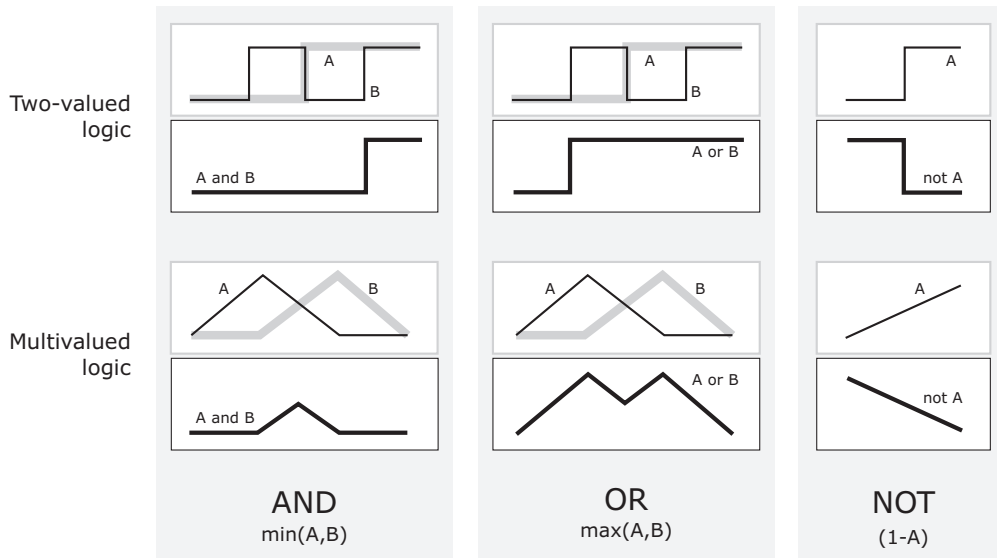
OR

A	$1 - A$
0	1
1	0

NOT

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

The next figure uses a graph to show the same information. In this figure, the truth table is converted 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 preceding two-valued truth tables, 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 you have defined.



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

Additional Fuzzy Operators

In this case, you defined only one particular correspondence between two-valued and multivalued logical operations for AND, OR, and NOT. This correspondence is by no means unique.

In more general terms, you are defining what are known as the fuzzy intersection or conjunction (AND), fuzzy union or disjunction (OR), and fuzzy complement (NOT). The classical operators for these functions are: 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. Fuzzy Logic Toolbox software uses the classical operator for the fuzzy complement as shown in the previous figure, but also enables you to customize the AND and OR operators.

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

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

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

A T -norm operator is a binary mapping $T(\dots)$ with the following properties:

- Boundary — $T(0,0) = 0, T(a,1) = T(1,a) = a$
- Monotonicity — $T(a,b) \leq T(c,d)$ if $a \leq c$ and $b \leq d$
- 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 pair-wise groupings.

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

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

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

A T -conorm (or S -norm) operator is a binary mapping $S(\dots)$ with the following properties:

- Boundary — $S(1,1) = 1, S(a,0) = S(0,a) = a$
- Monotonicity — $S(a,b) \leq S(c,d)$ if $a \leq c$ and $b \leq d$
- Commutativity — $S(a,b) = S(b,a)$
- Associativity — $S(a,S(b,c)) = S(S(a,b),c)$

Several parameterized T -norms and dual T -conorms have been proposed in the past, such as those of Yager [10], Dubois and Prade [1], Schweizer and Sklar [7], and Sugeno [8]. Each of these provides a way to vary the gain on the function so that it can be very restrictive or very permissive.

If-Then Rules

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

A single fuzzy if-then rule assumes the form

If x is A , then y is B

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

If service is good then tip is average

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

If service == good, then tip = average

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

Interpreting an if-then rule involves two steps:

- Evaluation of the antecedent — *Fuzzifying* the inputs and applying any necessary *fuzzy operators*.

- Application of the result to the consequent.

The second step is known as *implication*. For an if-then rule, the antecedent, p , *implies* the consequent, q . In binary logic, if p is true, then q is also true ($p \rightarrow q$). In fuzzy logic, if p is true to some degree of membership, then q is also true to the same degree ($0.5p \rightarrow 0.5q$). In both cases, if p is false, then the value of q is undetermined.

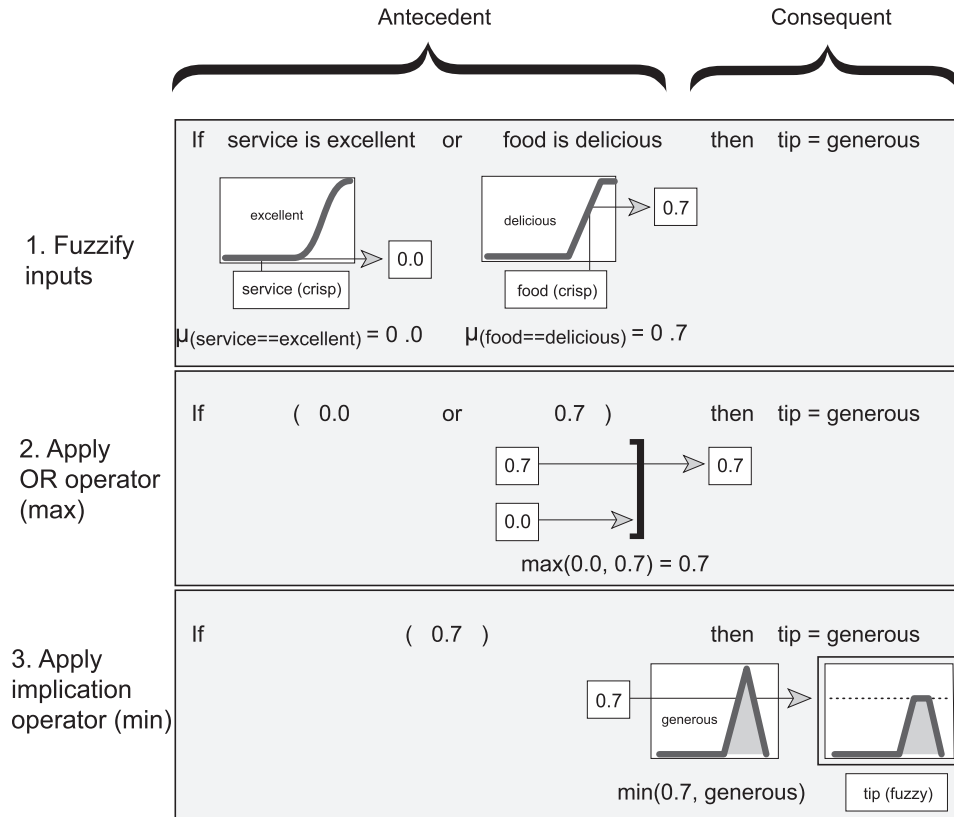
The antecedent of a rule can have multiple parts.

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

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

If temperature is cold, then hot water valve is open and cold water valve is shut

In this case, all consequents are affected equally by the result of the antecedent. How is the consequent affected by the antecedent? The consequent specifies a fuzzy set be assigned to the output. The *implication function* then modifies that fuzzy set to the degree specified by the antecedent. The most common ways to modify the output fuzzy set are truncation using the `min` function (where the fuzzy set is truncated as shown in the following figure) or scaling using the `prod` function (where the output fuzzy set is squashed). Both are supported by the toolbox, but you use truncation for the examples in this section.



Summary of If-Then Rules

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

- 1 Fuzzify inputs:** Resolve all fuzzy statements in the antecedent to a degree of membership between 0 and 1. If there is only one part to the antecedent, then this is the degree of support for the rule.
- 2 Apply fuzzy operator to multiple part antecedents:** If there are multiple parts to the antecedent, apply fuzzy logic operators and resolve the antecedent to a single number between 0 and 1. This is the degree of support for the rule.
- 3 Apply implication method:** Use the degree of support for the entire rule to shape the output fuzzy set. The consequent of a fuzzy rule assigns an entire fuzzy set to the

output. This fuzzy set is represented by a membership function that is chosen to indicate the qualities of the consequent. If the antecedent is only partially true, (i.e., is assigned a value less than 1), then the output fuzzy set is truncated according to the implication method.

In general, one rule alone is not effective. Two or more rules that can play off one another are needed. The output of each rule is a fuzzy set. The output fuzzy sets for each rule are then aggregated into a single output fuzzy set. Finally the resulting set is defuzzified, or resolved to a single number. "Build Fuzzy Systems Using Fuzzy Logic Designer" on page 2-14 shows how the whole process works from beginning to end for a particular type of fuzzy inference system called a *Mamdani type*.

References

- [1] Dubois, D. and H. Prade, *Fuzzy Sets and Systems: Theory and Applications*, Academic Press, New York, 1980.
- [2] Kaufmann, A. and M.M. Gupta, *Introduction to Fuzzy Arithmetic*, V.N. Reinhold, 1985.
- [3] Lee, C.-C., "Fuzzy logic in control systems: fuzzy logic controller-parts 1 and 2," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 20, No. 2, pp 404-435, 1990.
- [4] 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.
- [5] Mamdani, E.H., "Advances in the linguistic synthesis of fuzzy controllers," *International Journal of Man-Machine Studies*, Vol. 8, pp. 669-678, 1976.
- [6] 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.
- [7] Schweizer, B. and A. Sklar, "Associative functions and abstract semi-groups," *Publ. Math Debrecen*, 10:69-81, 1963.
- [8] 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, NY, 1977.
- [9] Sugeno, M., *Industrial applications of fuzzy control*, Elsevier Science Pub. Co., 1985.

- [10] Yager, R., "On a general class of fuzzy connectives," *Fuzzy Sets and Systems*, 4:235-242, 1980.
- [11] 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.
- [12] Zadeh, L.A., "Fuzzy sets," *Information and Control*, Vol. 8, pp. 338-353, 1965.
- [13] 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.
- [14] 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.
- [15] Zadeh, L.A., "Fuzzy Logic," *Computer*, Vol. 1, No. 4, pp. 83-93, 1988.
- [16] Zadeh, L.A., "Knowledge representation in fuzzy logic," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, pp. 89-100, 1989.

See Also

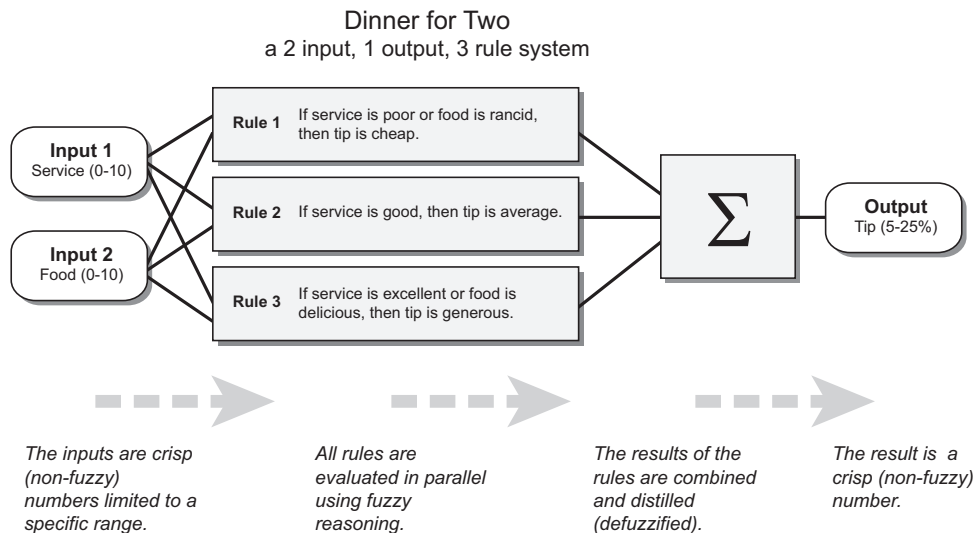
More About

- "What Is Fuzzy Logic?" on page 1-3
- "Fuzzy Inference Process" on page 1-28
- "Fuzzy vs. Nonfuzzy Logic" on page 1-47

Fuzzy Inference Process

Fuzzy inference is the process of formulating the mapping from a given input to an output using fuzzy logic. The mapping then provides a basis from which decisions can be made, or patterns discerned. The process of fuzzy inference involves all the pieces that are described in “Membership Functions” on page 1-14, “Logical Operations” on page 1-19, and “If-Then Rules” on page 1-23.

This section describes the fuzzy inference process and uses the example of the two-input, one-output, three-rule tipping problem from “The Basic Tipping Problem” on page 2-16. The basic structure of this example is shown in the following diagram:



Information flows from left to right, from two inputs to a single output. The parallel nature of the rules is an important aspect of fuzzy logic systems. Instead of sharp switching between modes based on breakpoints, logic flows smoothly from regions where one rule or another dominates.

Fuzzy inference process comprises of five parts:

- Fuzzification of the input variables on page 1-29
- Application of the fuzzy operator (AND or OR) in the antecedent on page 1-30
- Implication from the antecedent to the consequent on page 1-31

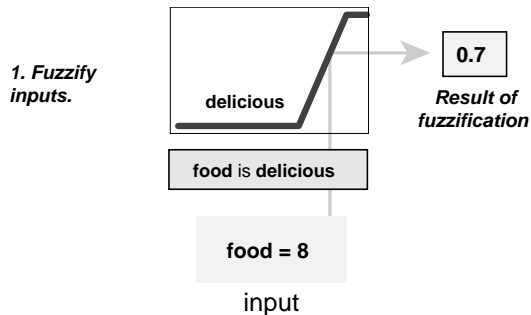
- Aggregation of the consequents across the rules on page 1-31
- Defuzzification on page 1-33

A fuzzy inference diagram on page 1-34 displays all parts of the fuzzy inference process — from fuzzification through defuzzification.

Fuzzify Inputs

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

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



In this manner, each input is fuzzified over all the qualifying membership functions required by the rules.

Apply Fuzzy Operator

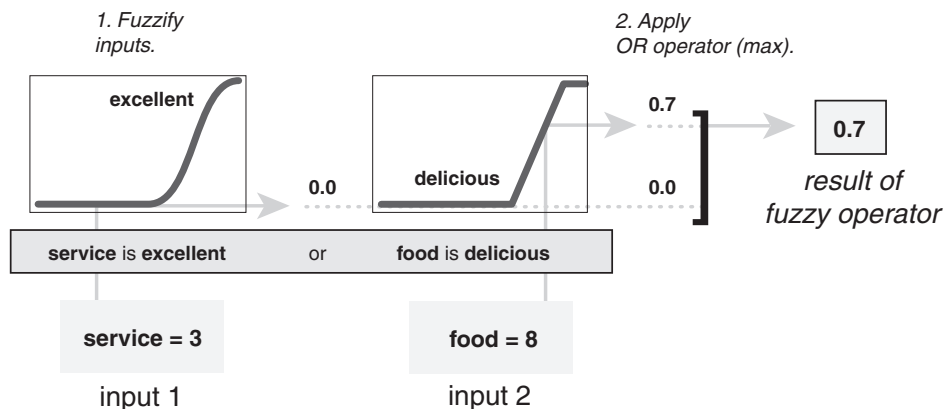
After the inputs are fuzzified, you know the degree to which each part of the antecedent is satisfied for each rule. If the antecedent of a rule has more than one part, the fuzzy operator is applied to obtain one number that represents the result of the rule antecedent. This number is then applied to the output function. The input to the fuzzy operator is two or more membership values from fuzzified input variables. The output is a single truth value.

As is described in “Logical Operations” on page 1-19 section, any number of well-defined methods can fill in for the AND operation or the OR operation. In the 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.

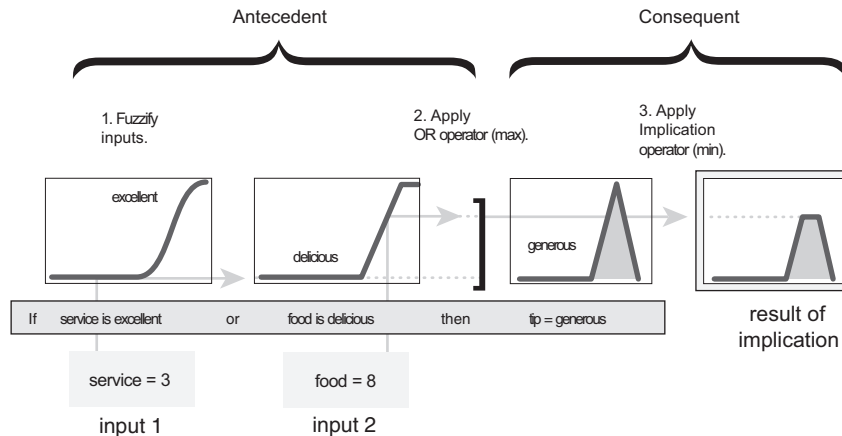
The following figure shows the OR operator *max* at work, evaluating the antecedent of the rule 3 for the tipping calculation. The two different pieces of the antecedent (service is excellent and food is delicious) yielded the fuzzy membership values 0.0 and 0.7 respectively. The fuzzy OR operator simply selects the maximum of the two values, 0.7, and the fuzzy operation for rule 3 is complete. The probabilistic OR method would still result in 0.7.



Apply Implication Method

Before applying the implication method, you must determine the rule weight. Every rule has a *weight* (a number from 0 through 1), which is applied to the number given by the antecedent. Generally, this weight is 1 (as it is for this example) and thus has no effect on the implication process. However, you can decrease the effect of one rule relative to the others by changing its weight value to something other than 1.

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



Note Sugeno systems always use the product implication method.

Aggregate All Outputs

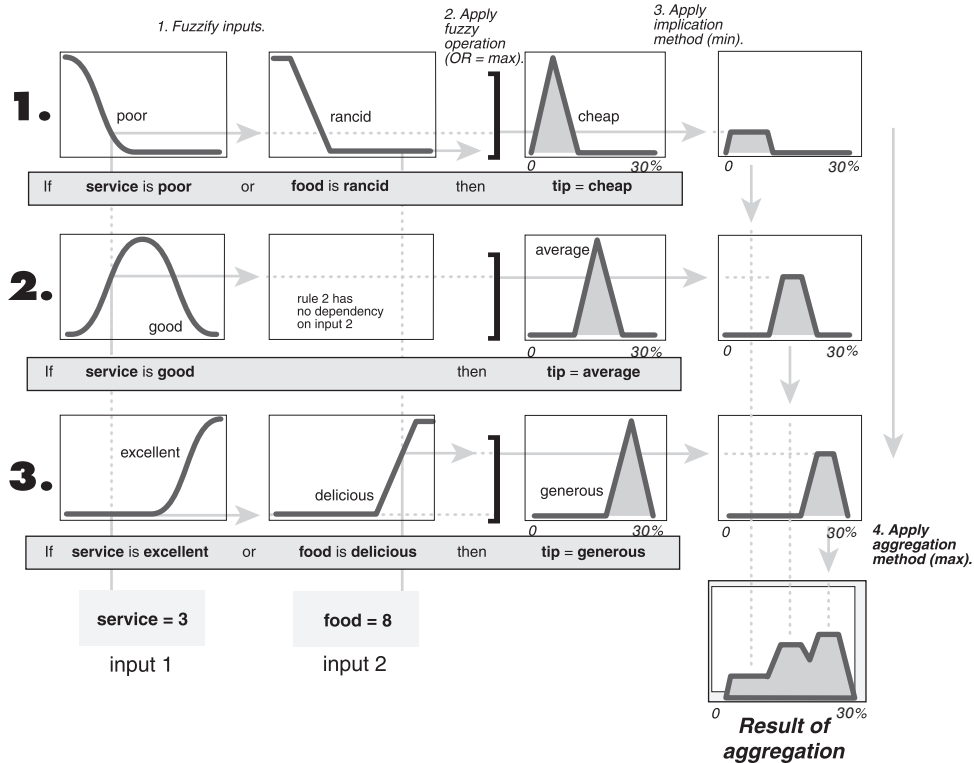
Since decisions are based on testing all the rules in a FIS, the rule outputs must be combined in some manner. Aggregation is the process by which the fuzzy sets that

represent the outputs of each rule are combined into a single fuzzy set. Aggregation only occurs once for each output variable, which is before the final defuzzification step. 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.

As long as the aggregation method is commutative, then the order in which the rules are executed is unimportant. Three built-in methods are supported:

- `max` (maximum)
- `probor` (probabilistic OR)
- `sum` (sum of the rule output sets)

In the following diagram, all three rules are displayed to show how the rule outputs are aggregated into a single fuzzy set whose membership function assigns a weighting for every output (tip) value.



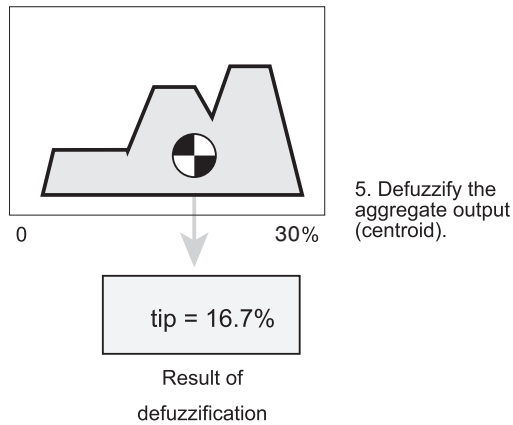
Note Sugeno systems always use the sum aggregation method.

Defuzzify

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

There are five built-in defuzzification methods supported: centroid, bisector, middle of maximum (the average of the maximum value of the output set), largest of maximum, and

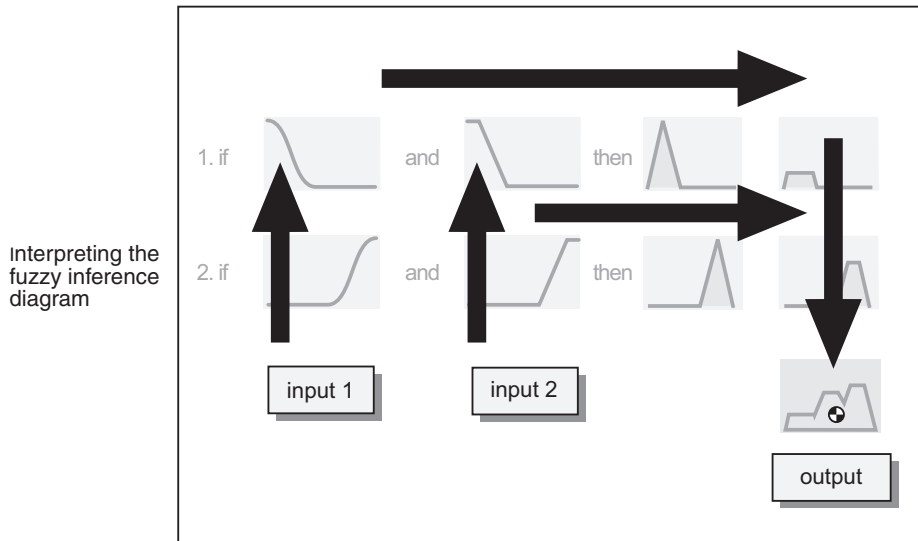
smallest of maximum. Perhaps the most popular defuzzification method is the centroid calculation, which returns the center of area under the curve, as shown in the following:



While the aggregate output fuzzy set covers a range from 0% though 30%, the defuzzified value is between 5% and 25%. These limits correspond to the centroids of the cheap and generous membership functions, respectively.

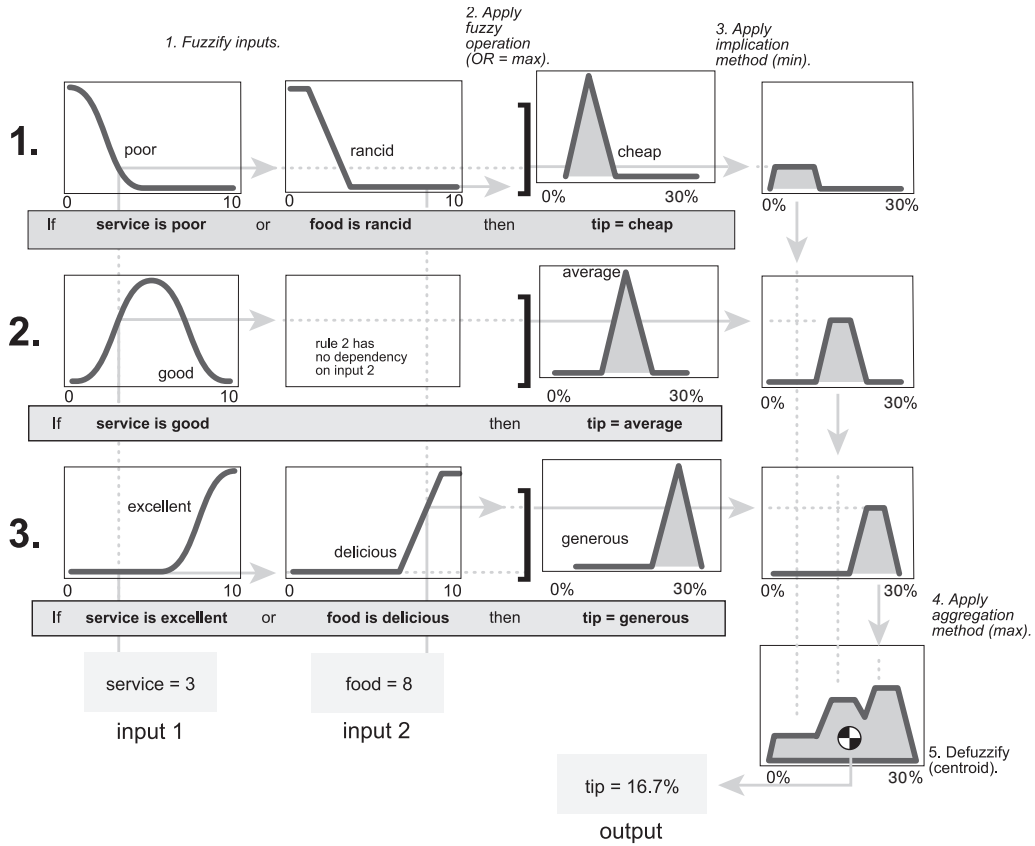
Fuzzy Inference Diagram

The fuzzy inference diagram is the composite of all the smaller diagrams presented so far in this section. It simultaneously displays all parts of the fuzzy inference process you have examined. Information flows through the fuzzy inference diagram as shown in the following figure.



In this figure, the flow proceeds up from the inputs in the lower left, across each row, and then down the rule outputs in the lower right. This compact flow shows everything at once, from linguistic variable fuzzification all the way through defuzzification of the aggregate output.

The following figure shows the actual full-size fuzzy inference diagram. Using a fuzzy inference diagram, you can learn a lot about how the system operates. For instance, for the particular inputs in this diagram, you can see that the implication method is truncation with the *min* function. The *max* function is used for the fuzzy OR operation. Rule 3 (the bottom-most row in the diagram shown previously) has the strongest influence on the output. The Rule Viewer described in "The Rule Viewer" on page 2-33 is an implementation of the fuzzy inference diagram.



See Also

More About

- "Foundations of Fuzzy Logic" on page 1-10

Membership Function Gallery

This example shows how to display 11 membership functions supported in the Fuzzy Logic Toolbox[®].

Define membership functions.

```
mf = [...
    fismf('trapmf',[-19 -17 -12 -7]) ...
    fismf('gbellmf',[3 4 -8]) ...
    fismf('trimf',[-9 -1 2]) ...
    fismf('gaussmf',[3 5]) ...
    fismf('gauss2mf',[3 10 5 13]) ...
    fismf('smf',[11 17]) ...
    fismf('zmf',[-18 -10]) ...
    fismf('psigmf',[2 -11 -5 -4]) ...
    fismf('dsigmf',[5 -3 1 5]) ...
    fismf('pimf',[0 7 11 15]) ...
    fismf('sigmf',[2 15]) ...
];
```

For more information on the different membership functions and their parameters, see their respective function reference pages.

Evaluate the membership functions.

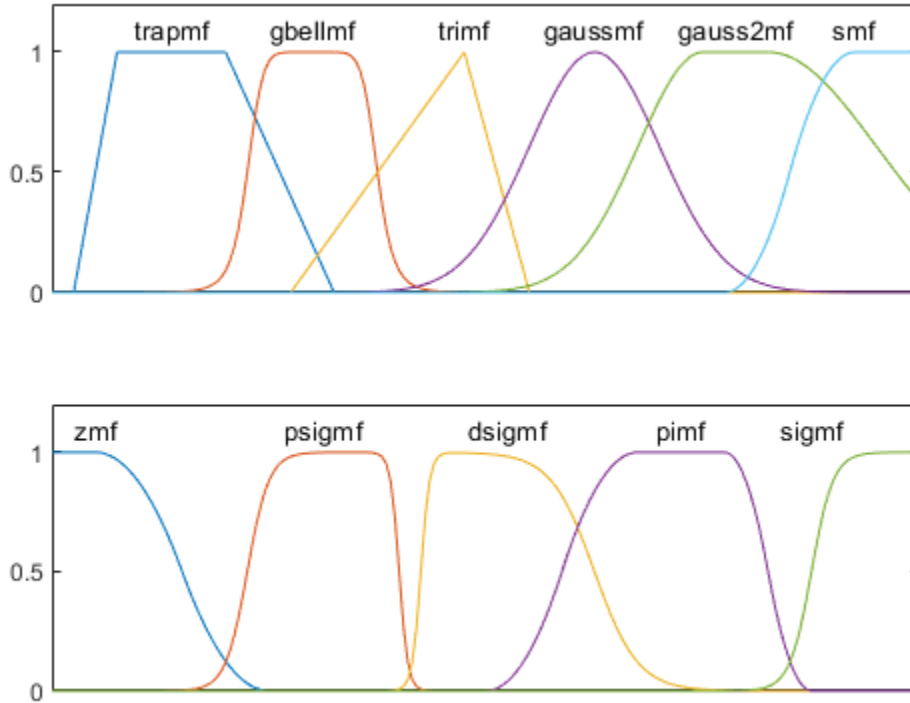
```
x = linspace(-20,20,201);
y = evalmf(mf,x);
```

Plot the evaluated membership functions with labels.

```
subplot(2,1,1);
plot(x,y(1:6,:));
axis([min(x) max(x) 0 1.2]);
text((mf(1).Parameters(2)+mf(1).Parameters(3))/2,1.1,mf(1).Type,...
    'horizon','center');
text(mf(2).Parameters(3),1.1,mf(2).Type,...
    'horizon','center');
text(mf(3).Parameters(2),1.1,mf(3).Type,...
    'horizon','center');
text(mf(4).Parameters(2),1.1,mf(4).Type,...
    'horizon','center');
text((mf(5).Parameters(2)+mf(5).Parameters(4))/2,1.1,mf(5).Type,...
    'horizon','center');
```

```
text(mf(6).Parameters(2), 1.1,mf(6).Type,...
     'horizon', 'center');
h_gca = gca;
h_gca.XTick = [];

subplot(2,1,2);
plot(x,y(7:11,:));
axis([min(x) max(x) 0 1.2]);
text(mf(7).Parameters(1),1.1,mf(7).Type,...
     'horizon', 'center');
text((mf(8).Parameters(2)+mf(8).Parameters(4))/2,1.1,mf(8).Type,...
     'horizon', 'center');
text((mf(9).Parameters(2)+mf(9).Parameters(4))/2,1.1,mf(9).Type,...
     'horizon', 'center');
text((mf(10).Parameters(2)+mf(10).Parameters(3))/2,1.1,mf(10).Type,...
     'horizon', 'center');
text(mf(11).Parameters(2),1.1,mf(11).Type,...
     'horizon', 'center');
h_gca = gca;
h_gca.XTick = [];
```

See Also

More About

- “Foundations of Fuzzy Logic” on page 1-10
- “Fuzzy Inference Process” on page 1-28

Defuzzification Methods

This example shows how to display five defuzzification methods supported in the Fuzzy Logic Toolbox™.

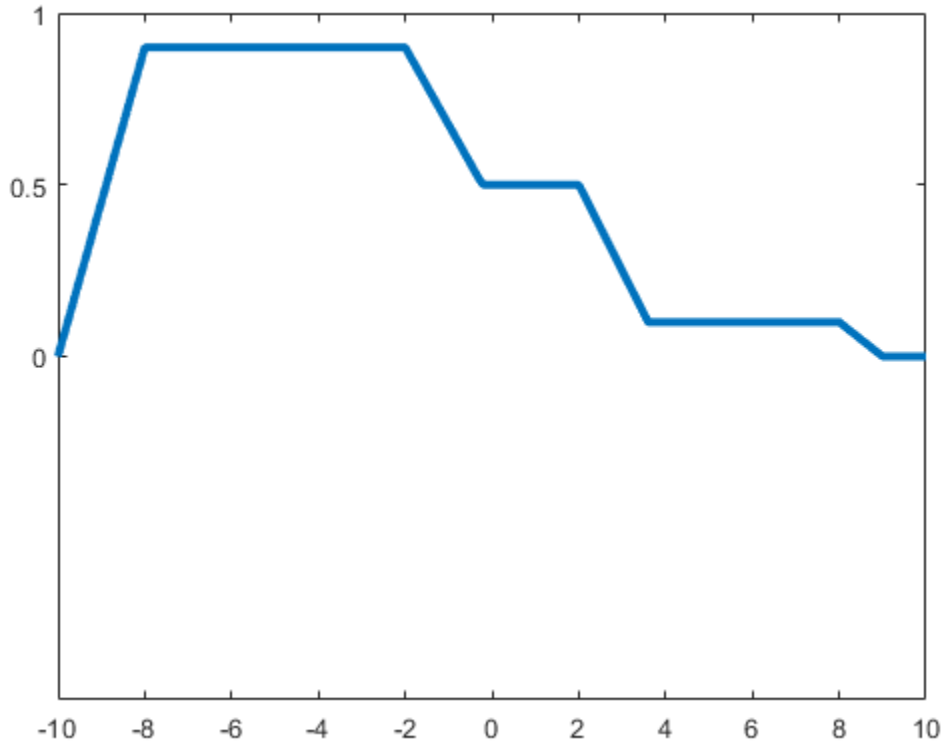
Problem Setup

Suppose you have the following region to be defuzzified. What are some of the methods you might choose?

```
x = -10:0.1:10;

mf1 = trapmf(x,[-10 -8 -2 2]);
mf2 = trapmf(x,[-5 -3 2 4]);
mf3 = trapmf(x,[2 3 8 9]);
mf1 = max(0.5*mf2,max(0.9*mf1,0.1*mf3));

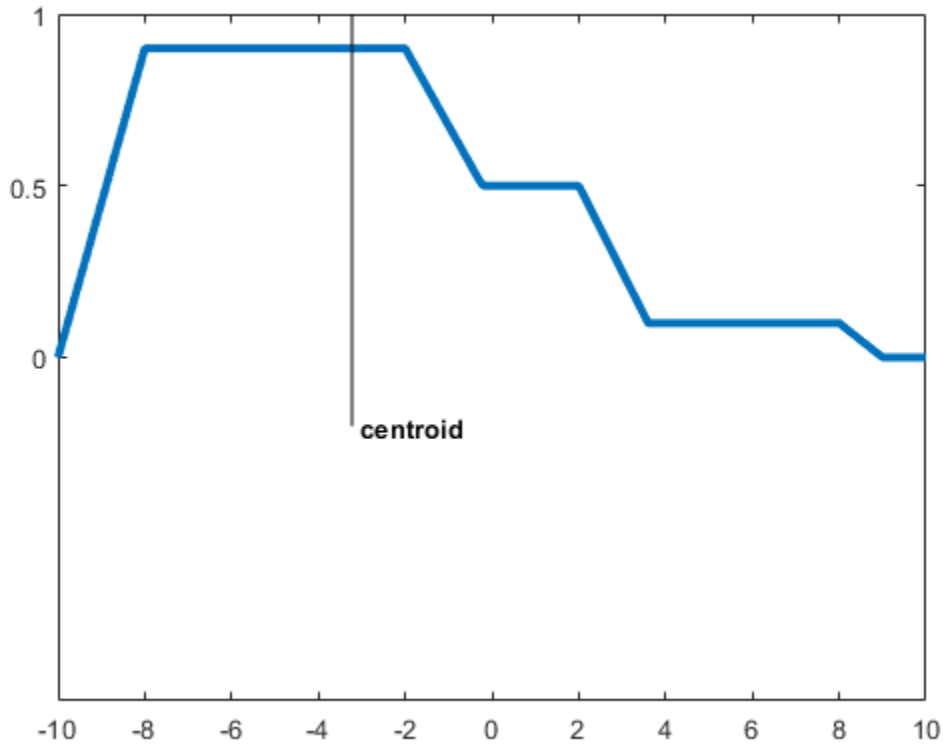
figure('Tag','defuzz');
plot(x,mf1,'LineWidth',3);
h_gca = gca;
h_gca.YTick = [0 .5 1] ;
ylim([-1 1]);
```



Centroid

Centroid defuzzification returns the center of area under the curve. If you think of the area as a plate of equal density, the centroid is the point along the x axis about which this shape would balance.

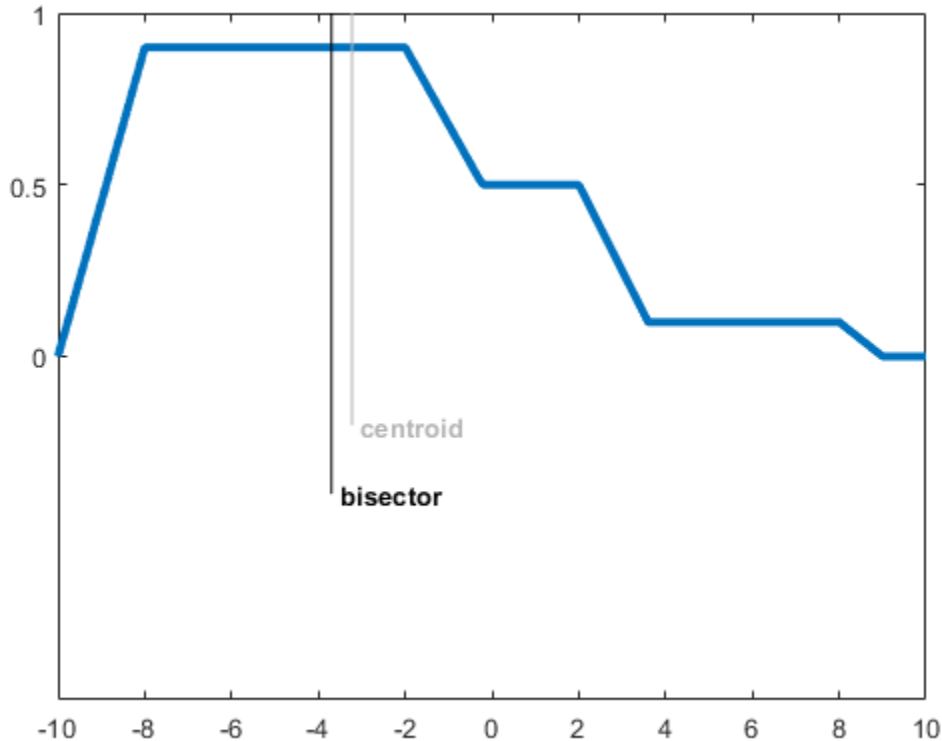
```
x1 = defuzz(x,mf1,'centroid'); % #ok<*NOPTS>  
h1 = line([x1 x1],[-0.2 1.2],'Color','k');  
t1 = text(x1,-0.2,' centroid','FontWeight','bold');
```



Bisector

The bisector is the vertical line that will divide the region into two sub-regions of equal area. It is sometimes, but not always coincident with the centroid line.

```
x2 = defuzz(x,mf1,'bisector');  
  
gray = 0.7*[1 1 1];  
h1.Color = gray;  
t1.Color = gray;  
h2 = line([x2 x2],[-0.4 1.2],'Color','k');  
t2 = text(x2,-0.4,' bisector','FontWeight','bold');
```



Middle, Smallest, and Largest of Maximum

MOM, SOM, and LOM stand for Middle, Smallest, and Largest of Maximum, respectively. These three methods key off the maximum value assumed by the aggregate membership function. In this example, because there is a plateau at the maximum value, they are distinct. If the aggregate membership function has a unique maximum, then MOM, SOM, and LOM all take on the same value.

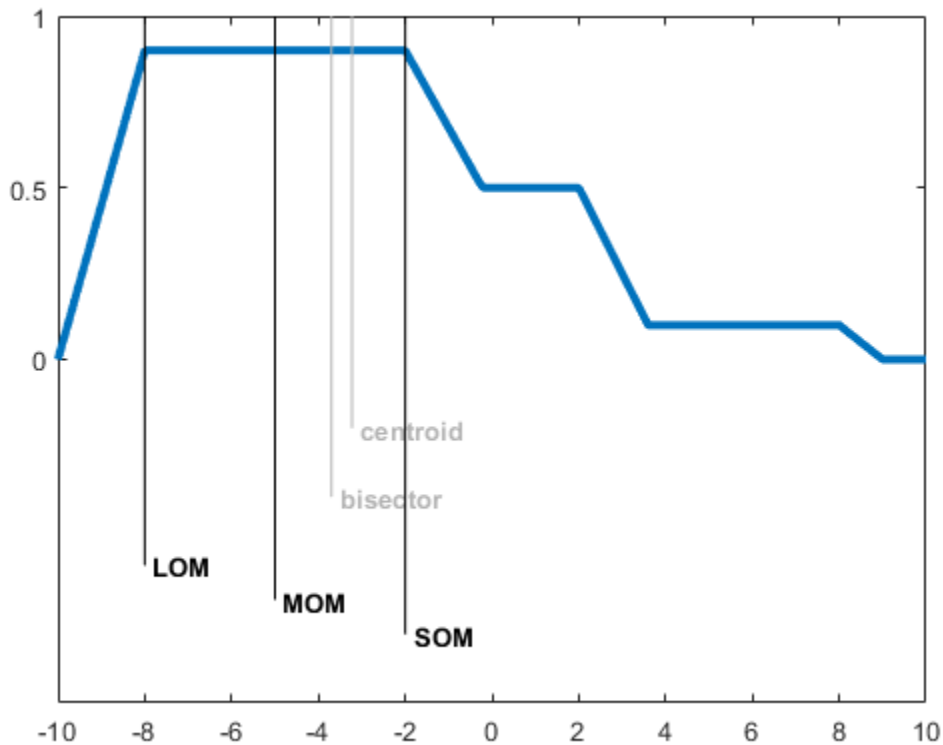
```
x3 = defuzz(x,mf1,'mom')
```

```
x3 = -5
```

```
x4 = defuzz(x,mf1,'som')
```

```
x4 = -2
```

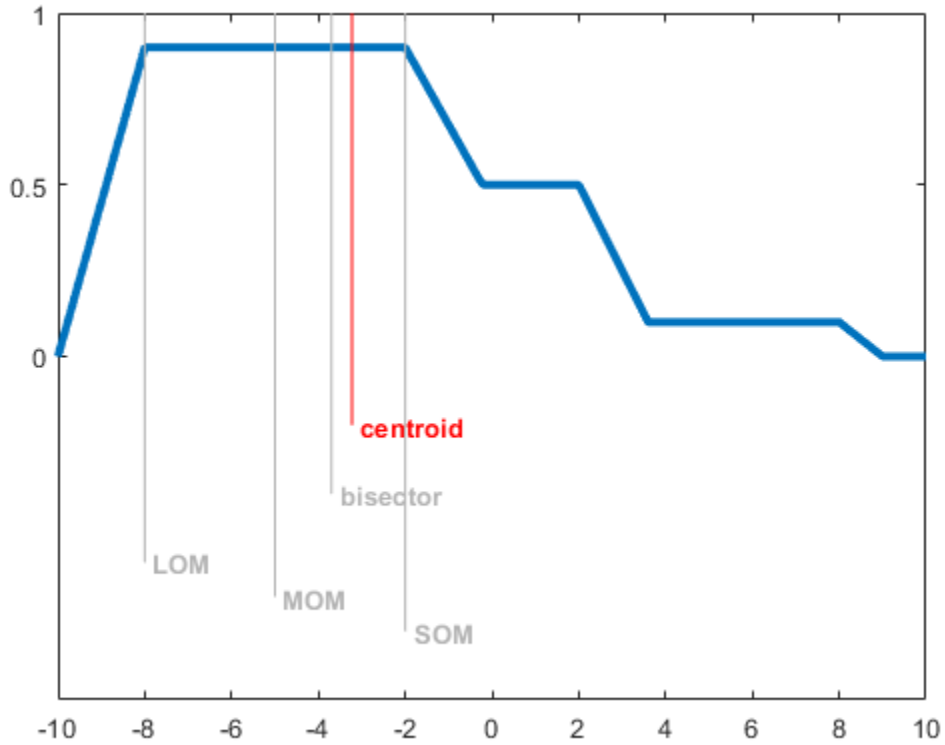
```
x5 = defuzz(x,mf1,'lom')  
  
x5 = -8  
  
h2.Color = gray;  
t2.Color = gray;  
  
h3 = line([x3 x3],[-0.7 1.2],'Color','k');  
t3 = text(x3,-0.7,'MOM','FontWeight','bold');  
h4 = line([x4 x4],[-0.8 1.2],'Color','k');  
t4 = text(x4,-0.8,'SOM','FontWeight','bold');  
h5 = line([x5 x5],[-0.6 1.2],'Color','k');  
t5 = text(x5,-0.6,'LOM','FontWeight','bold');
```



Picking a Method

Which of these methods is the right one? There's no simple answer. But if you want to get started quickly, generally the centroid method is good enough. Later you can always change your defuzzification method to see if another method works better.

```
h3.Color = gray;  
t3.Color = gray;  
h4.Color = gray;  
t4.Color = gray;  
h5.Color = gray;  
t5.Color = gray;  
h1.Color = 'red';  
t1.Color = 'red';
```



See Also

More About

- “Foundations of Fuzzy Logic” on page 1-10
- “Fuzzy Inference Process” on page 1-28

Fuzzy vs. Nonfuzzy Logic

Basic Tipping Problem

To illustrate the value of fuzzy logic, examine both linear and fuzzy approaches to the following problem:

What is the right amount to tip your waitperson?

First, work through this problem the conventional (nonfuzzy) way, writing MATLAB® commands that spell out linear and piecewise-linear relations. Then, look at the same system using fuzzy logic.

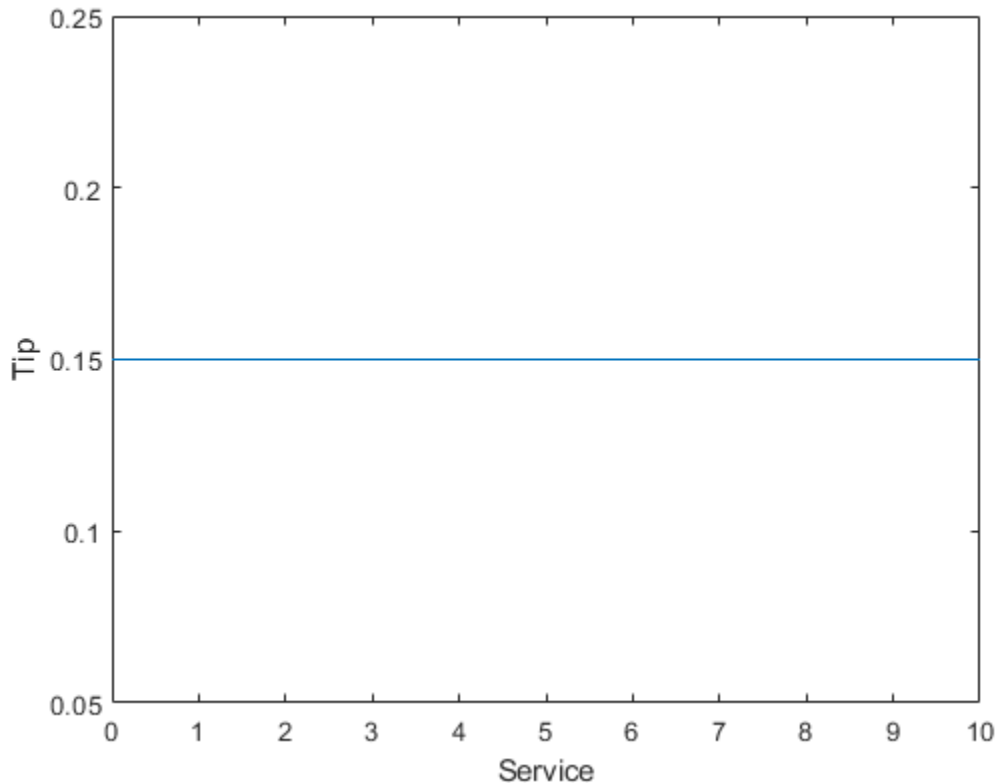
Basic Tipping Problem. Given a number from 0 through 10 that represents the quality of service at a restaurant (where 10 is excellent), what should the tip be?

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

Nonfuzzy Approach

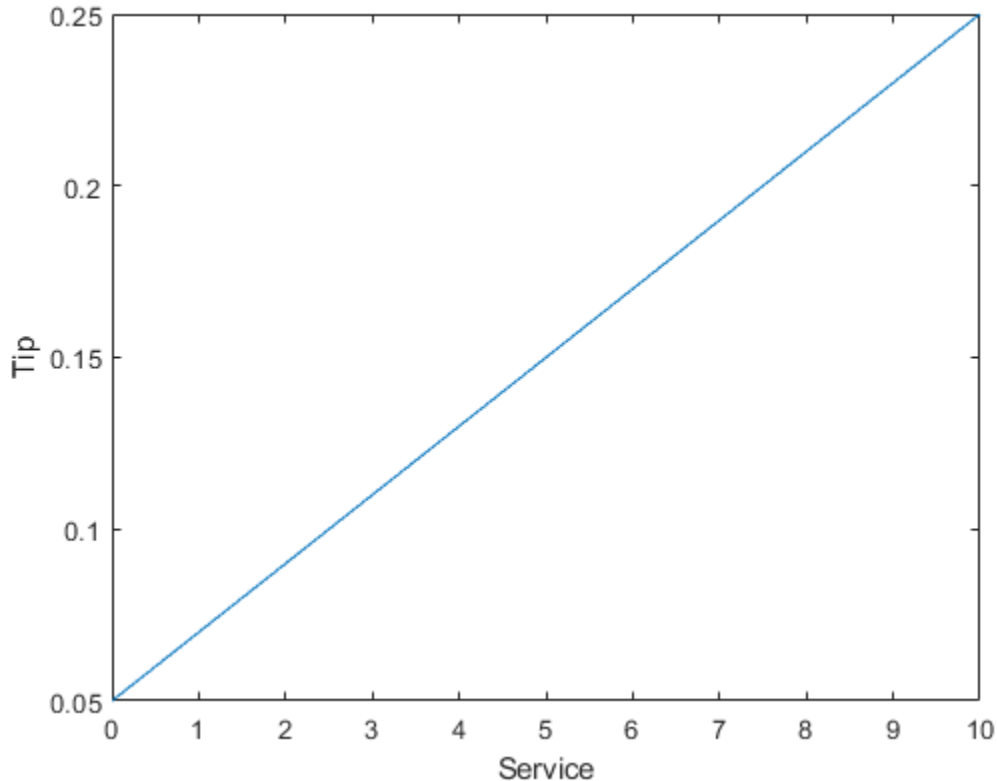
Begin with the simplest possible relationship. Suppose that the tip always equals 15% of the total bill.

```
service = 0:.5:10;  
tip = 0.15*ones(size(service));  
plot(service,tip)  
xlabel('Service')  
ylabel('Tip')  
ylim([0.05 0.25])
```



This relationship does not account for the quality of the service, so you must add a term to the equation. Since service is rated on a scale from 0 through 10, you the tip increase linearly from 5% if the service is bad to 25% if the service is excellent. Now the relation looks like the following plot:

```
tip = (.20/10)*service+0.05;  
plot(service,tip)  
xlabel('Service')  
ylabel('Tip')  
ylim([0.05 0.25])
```



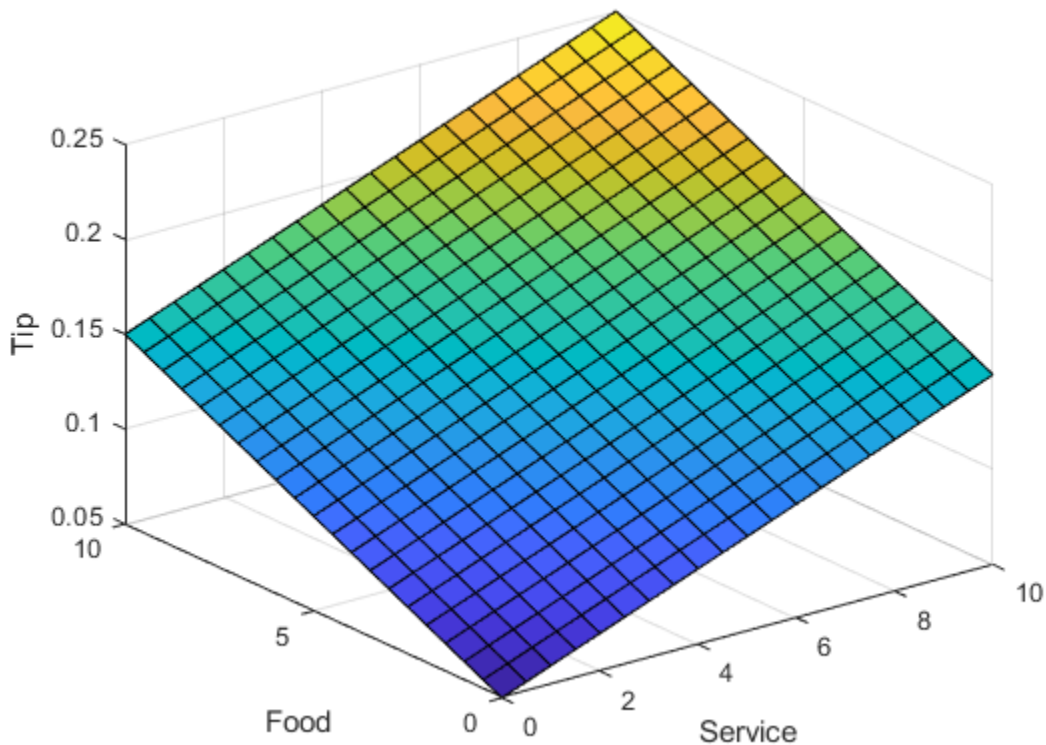
The formula does what you want it to do, and is straight forward. However, you may want the tip to reflect the quality of the food as well. This extension of the problem is defined as follows.

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

See how the formula is affected now that you have added another variable.

```
food = 0:.5:10;  
[F,S] = meshgrid(food,service);  
tip = (0.20/20).*(S+F)+0.05;  
surf(S,F,tip)
```

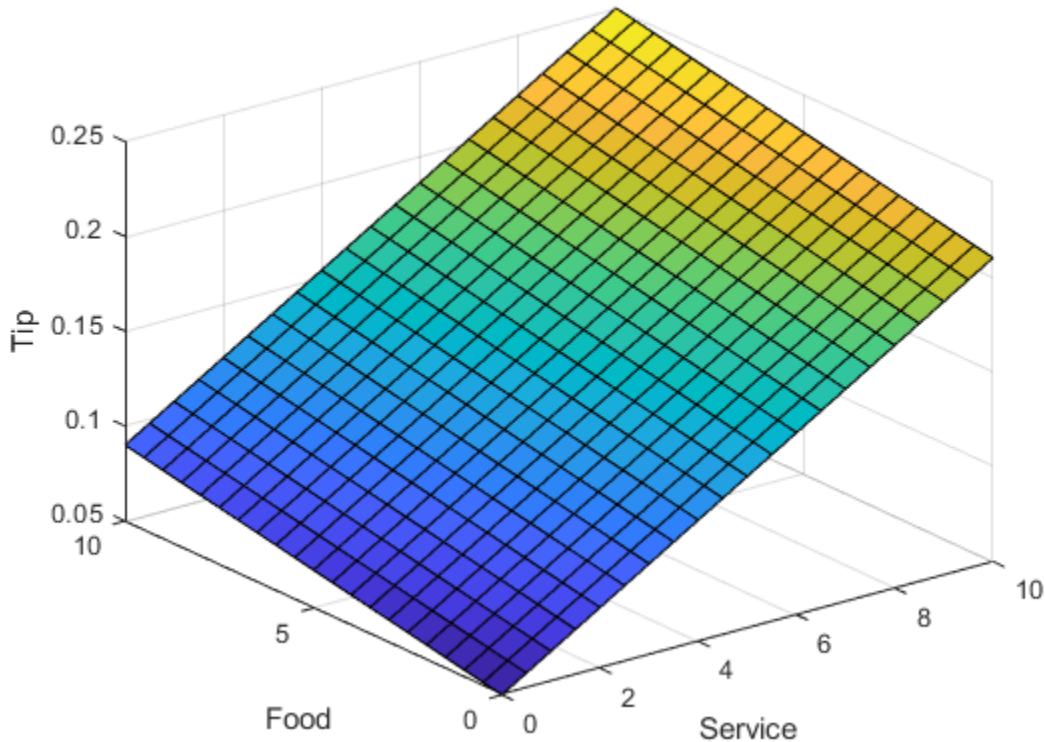
```
xlabel('Service')  
ylabel('Food')  
zlabel('Tip')
```



In this case, the results look satisfactory, but when you look at them closely, they do not seem right. Suppose that you want the service to be a more important factor than the food quality. Specify that service accounts for 80% of the overall tipping grade and the food makes up the other 20%.

```
servRatio = 0.8;  
tip = servRatio*(0.20/10*S+0.05) + ...  
      (1-servRatio)*(0.20/10*F+0.05);  
surf(S,F,tip)  
xlabel('Service')
```

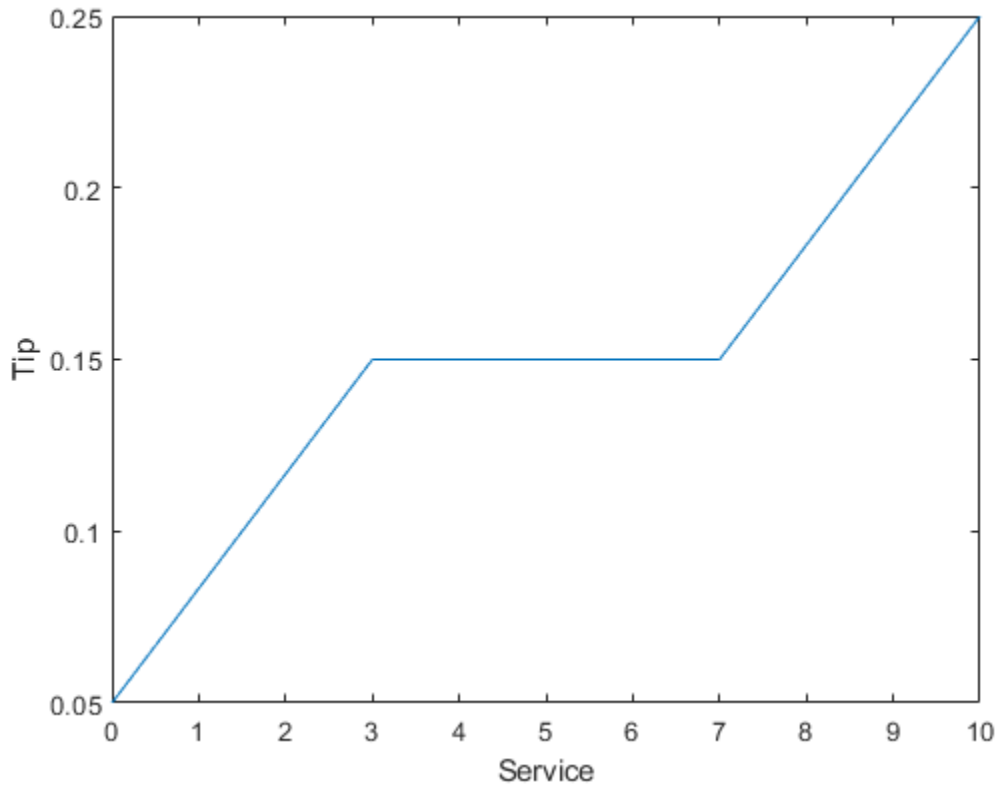
```
ylabel('Food')
zlabel('Tip')
```



The response is still some how too uniformly linear. Suppose that you want more of a flat response in the middle, that is, you want to give a 15% tip in general, but want to also specify a variation if the service is exceptionally good or bad. This factor, in turn, means that the previous linear mappings no longer apply. You can still use the linear calculation with a piecewise linear construction. Now, return to the one-dimensional problem of just considering the service. You can create a simple conditional tip assignment using logical indexing.

```
tip = zeros(size(service));
tip(service<3) = (0.10/3)*service(service<3)+0.05;
tip(service>=3 & service<7) = 0.15;
```

```
tip(service>=7 & service<=10) = ...  
    (0.10/3)*(service(service>=7 & service<=10)-7)+0.15;  
plot(service,tip)  
xlabel('Service')  
ylabel('Tip')  
ylim([0.05 0.25])
```



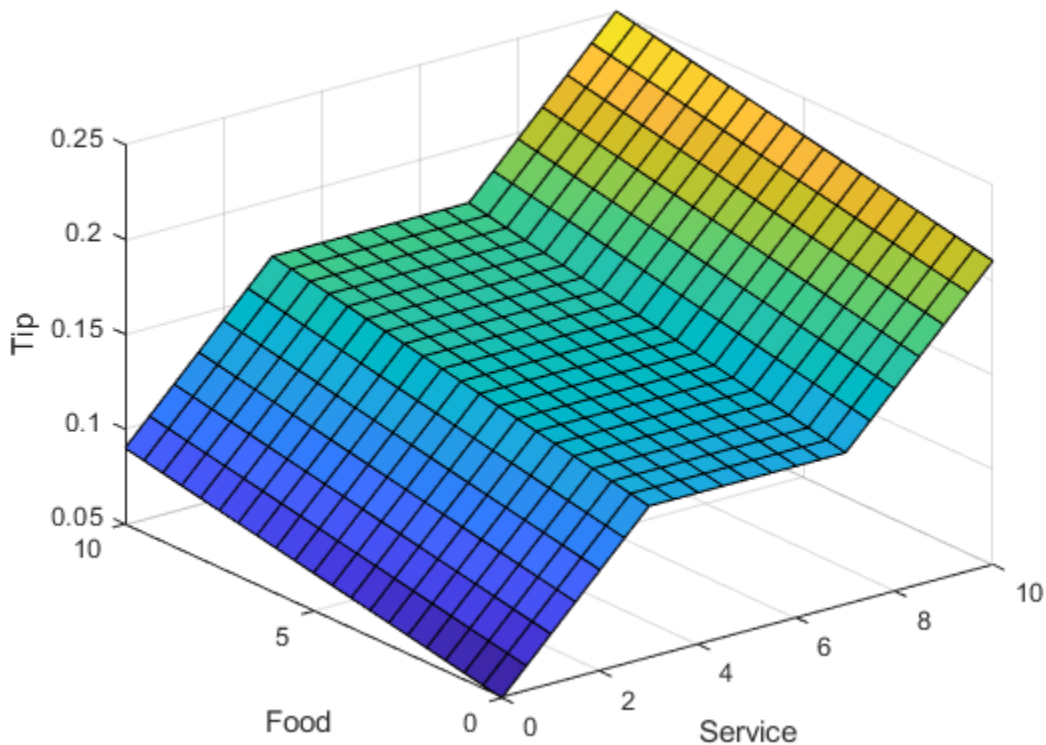
Suppose that you extend this approach to two dimensions, where you account for food quality again.

```
servRatio = 0.8;  
tip = zeros(size(S));  
tip(S<3) = ((0.10/3)*S(S<3)+0.05)*servRatio + ...  
    (1-servRatio)*(0.20/10*S(S<3)+0.05);
```

```

tip(S>=3 & S<7) = (0.15)*servRatio + ...
    (1-servRatio)*(0.20/10*F(S>=3 & S<7)+0.05);
tip(S>=7 & S<=10) = ((0.10/3)*(S(S>=7 & S<=10)-7)+0.15)*servRatio + ...
    (1-servRatio)*(0.20/10*F(S>=7 & S<=10)+0.05);
surf(S,F,tip)
xlabel('Service')
ylabel('Food')
zlabel('Tip')

```



The plot looks good, but the function is surprisingly complicated. It is even not apparent how the algorithm works to someone who did not see the original design process.

Fuzzy Logic Approach

In general, you want to capture the essentials of this problem, leaving aside all the factors that could be arbitrary. If you make a list of what really matters in this problem, you could end up with the following rule descriptions.

Tipping Problem Rules - Service Factor

- If service is poor, then tip is cheap
- If service is good, then tip is average
- If service is excellent, then tip is generous

The order in which the rules are presented here is arbitrary. It does not matter which rules come first. To include the effect of food quality on the tip, add the following two rules.

Tipping Problem Rules - Food Factor

- If food is rancid, then tip is cheap
- If food is delicious, then tip is generous

You can combine the two different lists of rules into one list of three rules like so.

Tipping Problem Rules - Both Service and Food Factors

- If service is poor or the food is rancid, then tip is cheap
- If service is good, then tip is average
- If service is excellent or food is delicious, then tip is generous

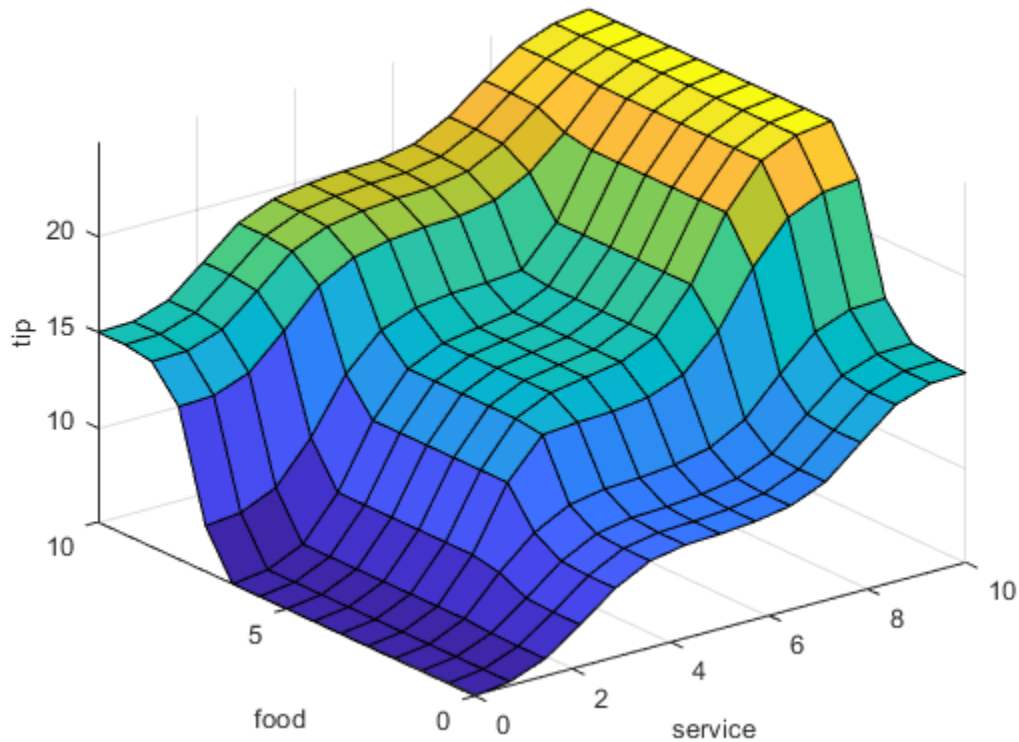
These three rules are the core of your solution and they correspond to the rules for a fuzzy logic system. When you give mathematical meaning to the linguistic variables (what is an average tip, for example) you have a complete fuzzy inference system. The methodology of fuzzy logic must also consider:

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

Problem Solution

The following plot represents the fuzzy logic system that solves the tipping problem.


```
gensurf(readfis('tipper'))
```



This plot was generated by the three rules that accounted for both service and food factors.

Observations Consider some observations about the example so far. You found a piecewise linear relation that solved the problem. It worked, but it was problematic to derive, and when you wrote it down as code, it was not easy to interpret. Conversely, the fuzzy logic system is based on some common sense statements. Also, you were able to add two more rules to the list that influenced the shape of the overall output without needing to undo what had already been done.

Moreover, by using fuzzy logic rules, the maintenance of the structure of the algorithm decouples along fairly clean lines. The notion of an average tip can change from day to

day, city to city, country to country. However, the underlying logic is the same: if the service is good, the tip should be average.

Recalibrating the Method You can recalibrate the method quickly by simply shifting the fuzzy set that defines average without rewriting the fuzzy logic rules.

You can shift lists of piecewise linear functions, but there is a greater likelihood for difficult recalibration.

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

```
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
```

As with all code, the more generality that is introduced, the less precise the algorithm becomes. You can improve clarity by adding more comments, or perhaps rewriting the algorithm in slightly more self-evident ways. But, the piecewise linear methodology is not the optimal way to resolve this issue.

If you remove everything from the algorithm except for three comments, what remain are exactly the fuzzy logic rules you previously wrote down.

- 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

Fuzzy logic uses language that is clear to you and that also has meaning to the computer, which is why it is a successful technique for bridging the gap between people and machines.

By making the equations as simple as possible (linear) you make things simpler for the machine, but more complicated for you. However, the limitation is no longer the computer - it is your mental model of what the computer is doing. Fuzzy logic lets the machine work with your preferences rather than the other way around.

See Also

Related Examples

- “Build Fuzzy Systems at the Command Line” on page 2-38
- “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14

Fuzzy Inference System Modeling

- “Types of Fuzzy Inference Systems” on page 2-2
- “What Is Mamdani-Type Fuzzy Inference?” on page 2-4
- “What Is Sugeno-Type Fuzzy Inference?” on page 2-5
- “Comparison of Sugeno and Mamdani Systems” on page 2-12
- “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14
- “Build Fuzzy Systems at the Command Line” on page 2-38
- “Build Fuzzy Systems Using Custom Functions” on page 2-50
- “Fuzzy Logic Image Processing” on page 2-65

Types of Fuzzy Inference Systems

You can implement two types of fuzzy inference systems in the toolbox:

- Mamdani
- Sugeno

These two types of inference systems vary somewhat in the way outputs are determined.

Mamdani's fuzzy inference method is the most commonly seen fuzzy methodology. Mamdani's method was among the first control systems built using fuzzy set theory. It was proposed in 1975 by Ebrahim Mamdani [1] 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 [2]. Although the inference process described in the next few sections differs somewhat from the methods described in the original paper, the basic idea is much the same.

Mamdani-type inference, as defined for the 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 is 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 type of output is sometimes known as a *singleton* output membership function, and it can be thought of as a pre-defuzzified fuzzy set. It enhances the efficiency of the defuzzification process because it greatly simplifies the computation required by the more general Mamdani method, which finds the centroid of a two-dimensional function. Rather than integrating across the two-dimensional function to find the centroid, you use the weighted average of a few data points. Sugeno-type systems support this type of model. In general, Sugeno-type systems can be used to model any inference system in which the output membership functions are either linear or constant.

For descriptions of these two types of fuzzy inference systems, see [3], [1], and [4].

Fuzzy inference systems have been successfully applied in fields such as automatic control, data classification, decision analysis, expert systems, and computer vision. Because of its multidisciplinary nature, fuzzy inference systems are associated with a number of names, such as fuzzy-rule-based systems, fuzzy expert systems, fuzzy modeling, fuzzy associative memory, fuzzy logic controllers, and simply (and ambiguously) fuzzy systems.

References

- [1] 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.
- [2] 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.
- [3] Jang, J.-S. R. and C.-T. Sun, *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*, Prentice Hall, 1997.
- [4] Sugeno, M., *Industrial applications of fuzzy control*, Elsevier Science Pub. Co., 1985.

See Also

More About

- "Fuzzy Inference Process" on page 1-28
- "What Is Mamdani-Type Fuzzy Inference?" on page 2-4
- "What Is Sugeno-Type Fuzzy Inference?" on page 2-5
- "Comparison of Sugeno and Mamdani Systems" on page 2-12

What Is Mamdani-Type Fuzzy Inference?

Mamdani fuzzy inference is the most commonly seen fuzzy methodology and was among the first control systems built using fuzzy set theory. It was proposed in 1975 by Ebrahim Mamdani [1] 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 [2]. Although the inference process described in the next few sections differs somewhat from the methods described in the original paper, the basic idea is much the same.

Mamdani-type inference, as defined for the 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.

References

- [1] 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.
- [2] 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.

See Also

More About

- "Comparison of Sugeno and Mamdani Systems" on page 2-12
- "What Is Sugeno-Type Fuzzy Inference?" on page 2-5
- "Build Fuzzy Systems Using Fuzzy Logic Designer" on page 2-14
- "Build Fuzzy Systems at the Command Line" on page 2-38
- "Build Fuzzy Systems Using Custom Functions" on page 2-50

What Is Sugeno-Type Fuzzy Inference?

This topic discusses the Sugeno, or Takagi-Sugeno-Kang, method of fuzzy inference. Introduced in 1985 [1], this method is similar to the Mamdani method in many respects. The first two parts of the fuzzy inference process, fuzzifying the inputs and applying the fuzzy operator, are the same. The main difference between Mamdani and Sugeno is that the Sugeno output membership functions are either linear or constant.

A typical rule in a Sugeno fuzzy model has the form:
 If Input 1 is x and Input 2 is y , then Output is $z = ax + by + c$

For a zero-order Sugeno model, the output level z is a constant ($a = b = 0$).

Each rule weights its output level, z_i , by the firing strength of the rule, w_i . For example, for an AND rule with Input 1 = x and Input 2 = y , the firing strength is

$$w_i = \text{AndMethod}(F_1(x), F_2(y))$$

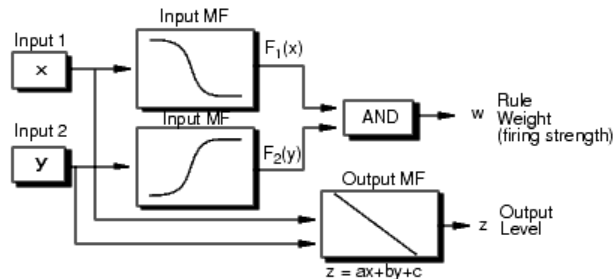
where $F_{1,2}(\cdot)$ are the membership functions for Inputs 1 and 2.

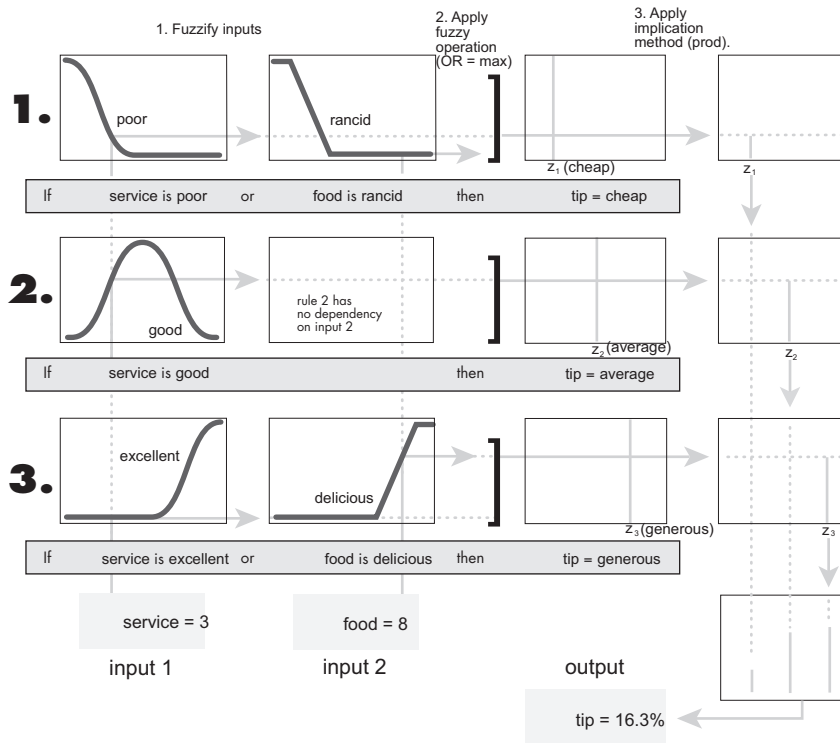
The final output of the system is the weighted average of all rule outputs, computed as

$$\text{Final Output} = \frac{\sum_{i=1}^N w_i z_i}{\sum_{i=1}^N w_i}$$

where N is the number of rules.

A Sugeno rule operates as shown in the following diagram.

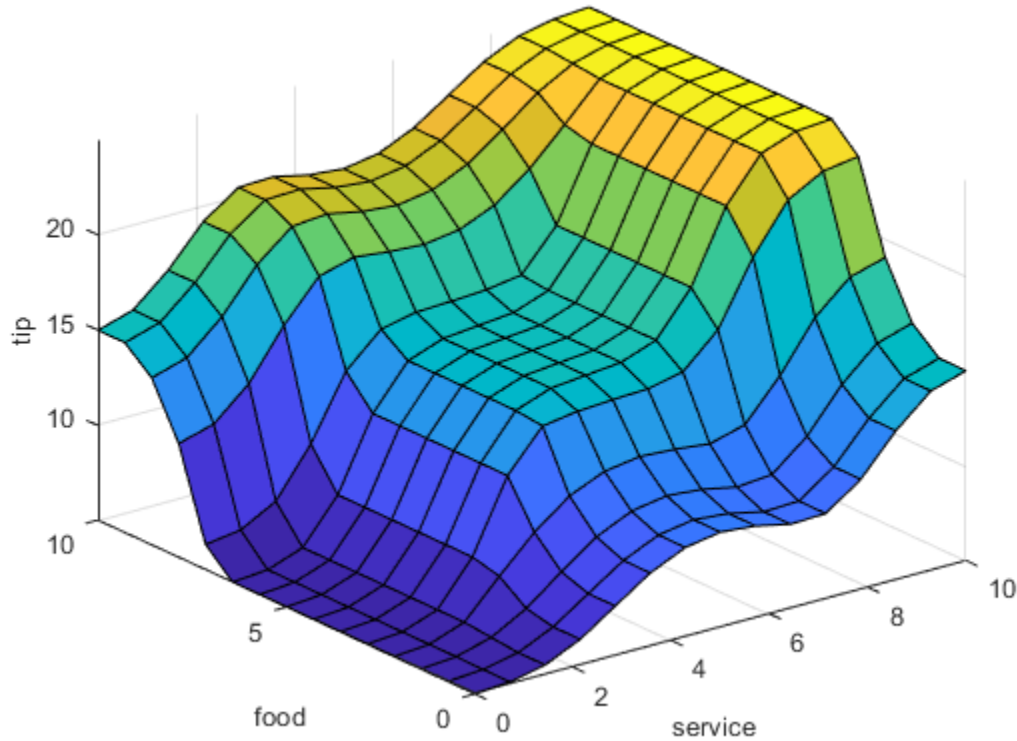




Note Sugeno systems always use product implication and sum aggregation.

The preceding figure shows the fuzzy tipping model developed in “Fuzzy Inference Process” on page 1-28 adapted for use as a Sugeno system. Fortunately, it is frequently the case that singleton output functions are sufficient for the needs of a given problem. As an example, the system `tippersg.fis` is the Sugeno-type representation of the now-familiar tipping model. If you load the system and plot its output surface, you see that it is almost the same as the Mamdani system you have previously seen.

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



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

Because of the linear dependence of each rule on the input variables, the Sugeno method is ideal for acting as an interpolating supervisor of multiple linear controllers that are to be applied, respectively, to different operating conditions of a dynamic nonlinear system. For example, the performance of an aircraft may change dramatically with altitude and Mach number. Linear controllers, though easy to compute and 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 suited to the task of smoothly interpolating the linear gains that would be applied across the input space; it is a natural and efficient gain scheduler. Similarly, a Sugeno system is suited for modeling nonlinear systems by interpolating between multiple linear models.

To see a specific example of a system with linear output membership functions, consider the one-input, one-output system stored in `sugenol.fis`. Load the system and view the properties of its output variable.

```
fis = readfis('sugenol');
fis.Outputs(1)

ans =
    fisvar with properties:
        Name: "output"
        Range: [0 1]
        MembershipFunctions: [1x2 fismf]
```

The output variable has two membership functions. View the properties of the first membership function.

```
fis.Outputs(1).MembershipFunctions(1)

ans =
    fismf with properties:
        Name: "line1"
        Type: "linear"
        Parameters: [-1 -1]
```

View the properties of the second membership function.

```
fis.Outputs(1).MembershipFunctions(2)

ans =
    fismf with properties:
        Name: "line2"
        Type: "linear"
        Parameters: [1 -1]
```

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

$$output = (-1) \times input + (-1)$$

and the membership function `line2` is:

$$output = (1) \times input + (-1)$$

The input membership functions and rules define which of these output functions are expressed and when:

`fis.Rules`

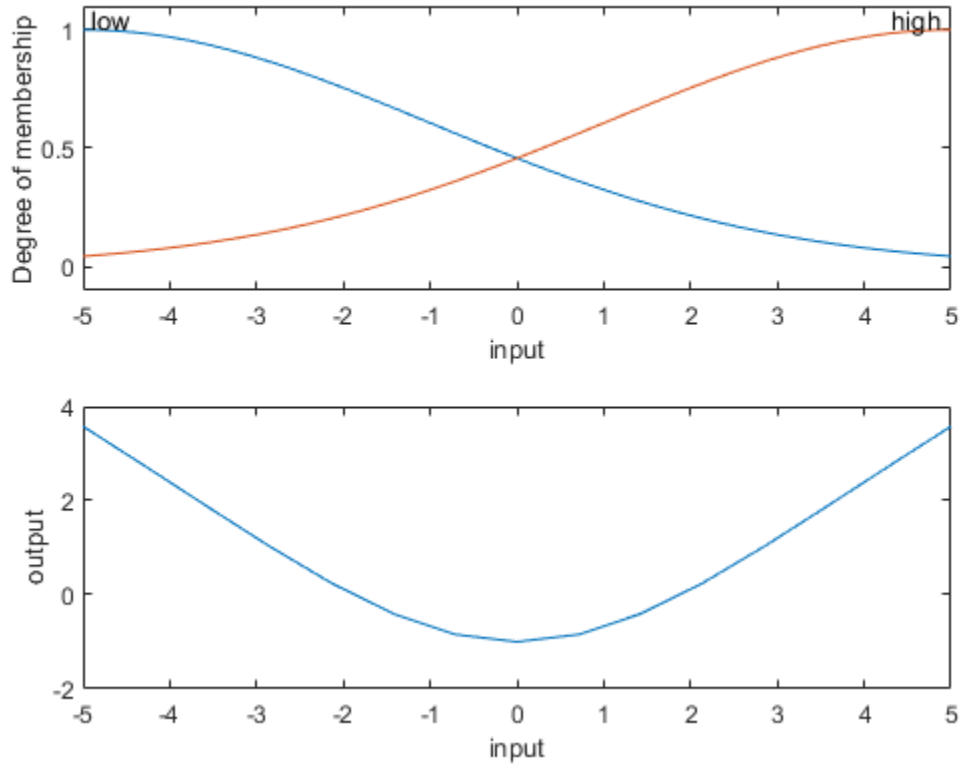
```
ans =
  1x2 fisrule array with properties:

    Description
    Antecedent
    Consequent
    Weight
    Connection

Details:
      Description
-----
  1  "input==low => output=line1 (1)"
  2  "input==high => output=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(fis,'input',1)
subplot(2,1,2)
gensurf(fis)
```



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

References

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

See Also

gensurf | readfis

More About

- “Comparison of Sugeno and Mamdani Systems” on page 2-12
- “What Is Mamdani-Type Fuzzy Inference?” on page 2-4

Comparison of Sugeno and Mamdani Systems

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

You can convert a Mamdani system into a Sugeno system using the `convertToSugeno` function. The resulting Sugeno system has constant output membership functions that correspond to the centroids of the Mamdani output membership functions.

Advantages of the Sugeno Systems

Sugeno systems:

- Are computationally efficient.
- Work well with linear techniques, such as PID control.
- Work well with optimization and adaptive techniques.
- Guarantee continuity of the output surface.
- Well-suited to mathematical analysis.

Advantages of the Mamdani Systems

Mamdani systems:

- Are intuitive.
- Have widespread acceptance.
- Are well-suited to human input.

See Also

`convertToSugeno`

More About

- “What Is Mamdani-Type Fuzzy Inference?” on page 2-4

- “What Is Sugeno-Type Fuzzy Inference?” on page 2-5

Build Fuzzy Systems Using Fuzzy Logic Designer

Fuzzy Logic Toolbox Graphical User Interface Tools

This example shows how to build a fuzzy inference system (FIS) for the tipping example, described in “The Basic Tipping Problem” on page 2-16, using the Fuzzy Logic Toolbox UI tools.

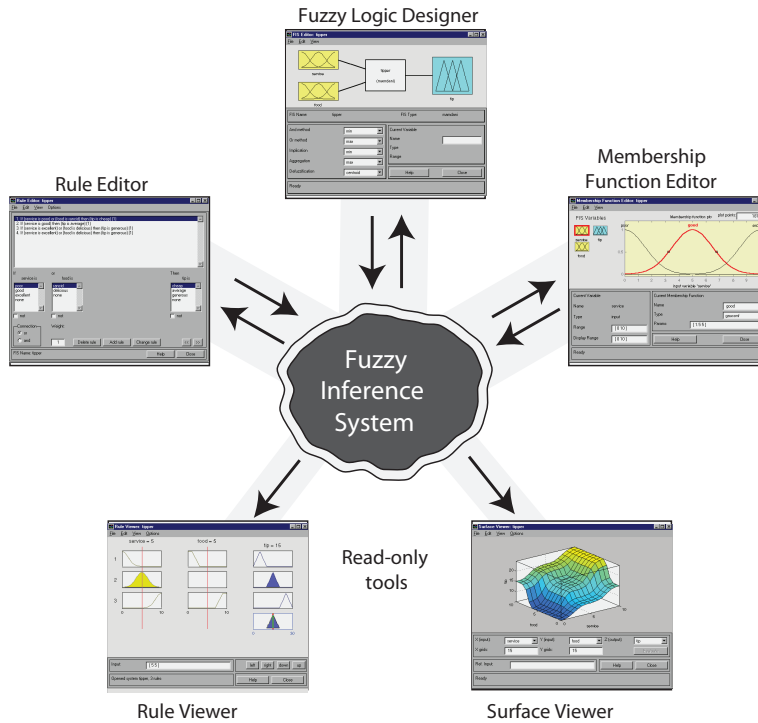
You use the following tools to build, edit, and view fuzzy inference systems:

- **Fuzzy Logic Designer** to handle the high-level issues for the system — How many input and output variables? What are their names?

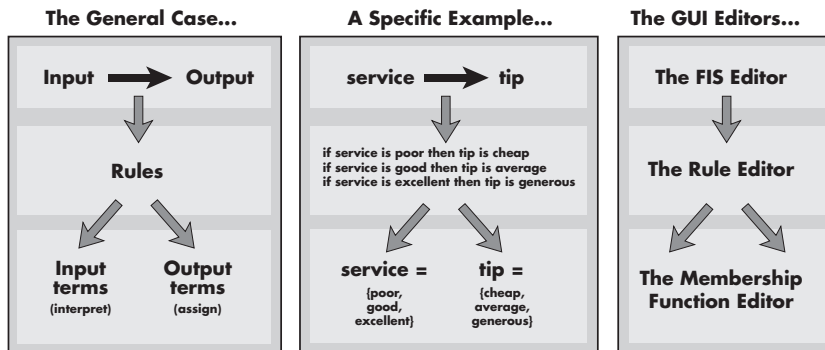
Fuzzy Logic Toolbox software does not limit the number of inputs. However, the number of inputs may be limited by the available memory of your machine. If the number of inputs is too large, or the number of membership functions is too big, then it may also be difficult to analyze the FIS using the other tools.

- **Membership Function Editor** on page 2-22 to define the shapes of all the membership functions associated with each variable
- **Rule Editor** on page 2-30 to edit the list of rules that defines the behavior of the system.
- **Rule Viewer** on page 2-33 to view the fuzzy inference diagram. Use this viewer as a diagnostic to see, for example, which rules are active, or how individual membership function shapes influence the results
- **Surface Viewer** on page 2-35 to view the dependency of one of the outputs on any one or two of the inputs; that is, it generates and plots an output surface map for the system.

These UIs are dynamically linked, in that changes you make to the FIS using one of them, affect what you see on any of the other open UIs. For example, if you change the names of the membership functions in the Membership Function Editor, the changes are reflected in the rules shown in the Rule Editor. You can use the UIs to read and write variables both to the MATLAB workspace and to a file (the read-only viewers can still exchange plots with the workspace and save them to a file). You can have any or all of them open for any given system or have multiple editors open for any number of fuzzy systems.



The following figure shows how the main components of a FIS and the three editors fit together. The two viewers examine the behavior of the entire system.



In addition to these five primary UIs, the toolbox includes the graphical **Neuro-Fuzzy Designer**, which you use to build and analyze Sugeno-type adaptive neuro-fuzzy inference systems.

The Fuzzy Logic Toolbox UIs do not support building a FIS using data. If you want to use data to build a FIS, use one of the following techniques:

- `genfis` to generate a Sugeno-type FIS. Then, select **File > Import** in the **Fuzzy Logic Designer** to import the FIS and perform fuzzy inference, as described in “The Fuzzy Logic Designer” on page 2-17.
- Neuro-adaptive learning techniques to model the FIS, as described in “Neuro-Adaptive Learning and ANFIS” on page 3-2.

If you want to use MATLAB workspace variables, use the command-line interface instead of the **Fuzzy Logic Designer**. For an example, see “Build Fuzzy Systems at the Command Line” on page 2-38.

The Basic Tipping Problem

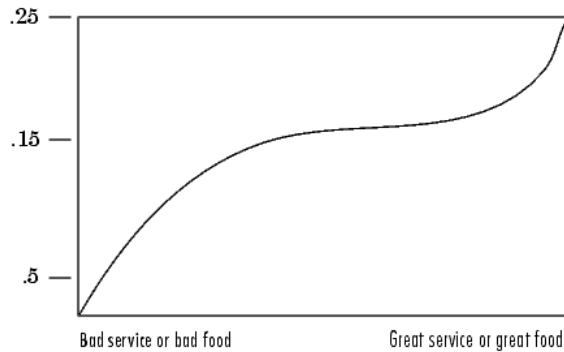
This example creates a Mamdani fuzzy inference system using on a two-input, one-output tipping problem based on tipping practices in the U.S. While the example creates a Mamdani FIS, the methods used apply to creating Sugeno systems as well.

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

The starting point is to write down the three golden rules of tipping:

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

Assume that an average tip is 15%, a generous tip is 25%, and a cheap tip is 5%.



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

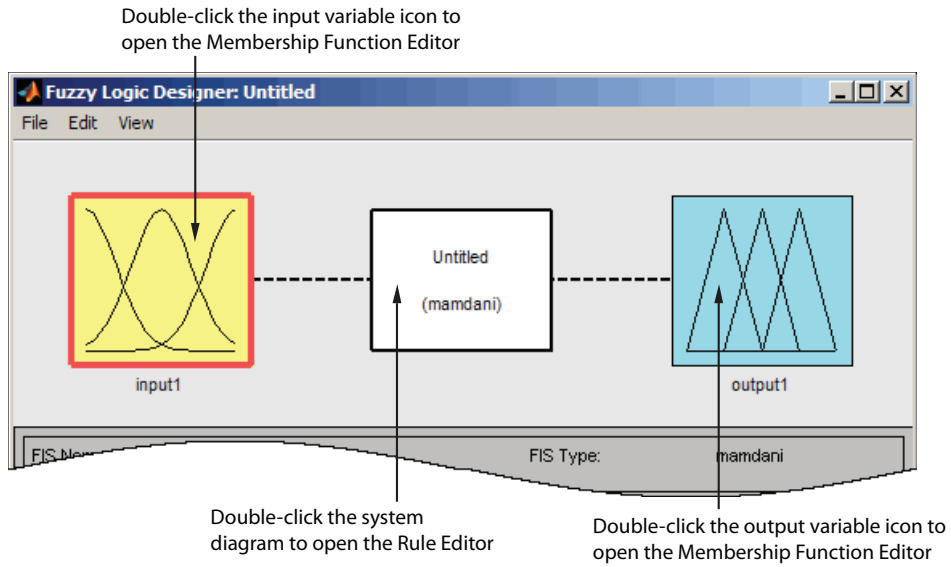
Now that you know the rules and have an idea of what the output should look like, use the UI tools to construct a fuzzy inference system for this decision process.

The Fuzzy Logic Designer

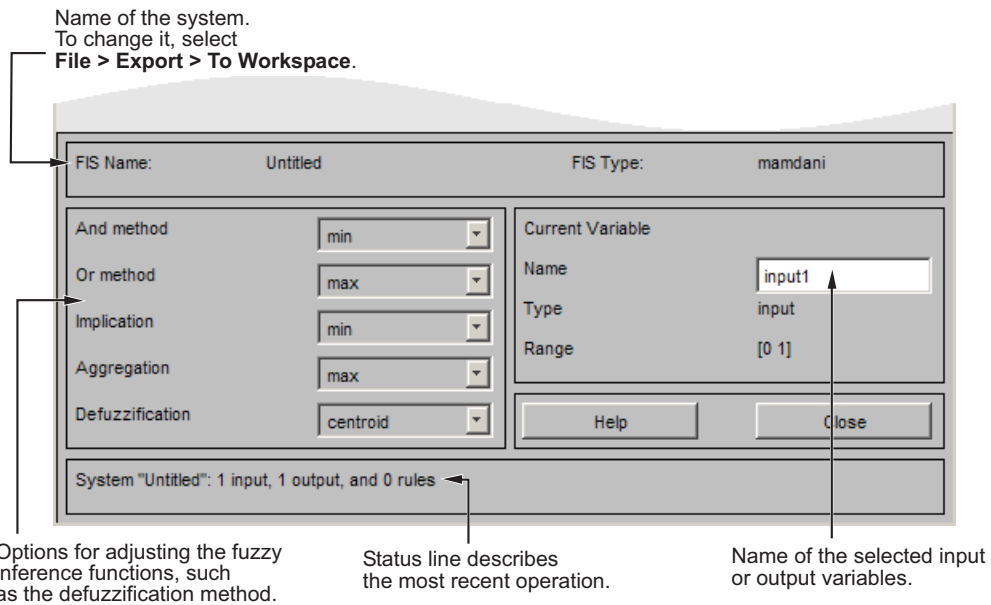
The **Fuzzy Logic Designer** displays information about a fuzzy inference system. To open the **Fuzzy Logic Designer**, type the following command at the MATLAB prompt:

```
fuzzyLogicDesigner
```

The **Fuzzy Logic Designer** opens and displays a diagram of the fuzzy inference system with the names of each input variable on the left, and those of each output variable on the right, as shown in the next figure. The sample membership functions shown in the boxes are just icons and do not depict the actual shapes of the membership functions.



Below the diagram is the name of the system and the type of inference used.



In this example, you use the default Mamdani-type inference. Another type of inference, called Sugeno-type inference, is also available. See “What Is Sugeno-Type Fuzzy Inference?” on page 2-5.

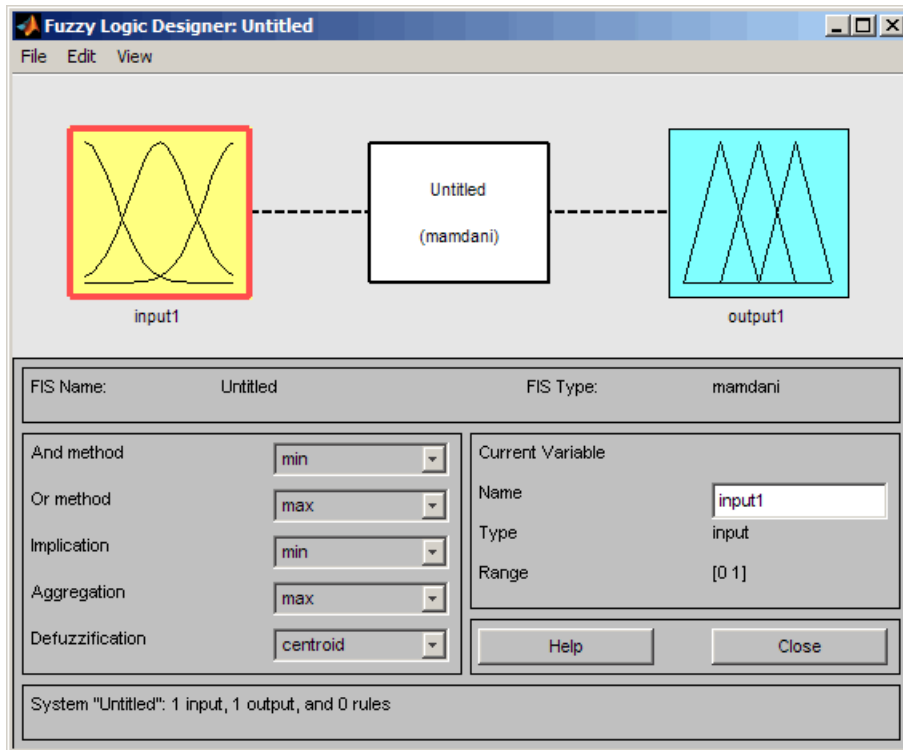
In the **Fuzzy Logic Designer**:

- The drop-down lists let you modify the fuzzy inference functions.
- The **Current Variable** area displays the name of either an input or output variable, its type, and default range.
- A status line at the bottom displays information about the most recent operation.

To build the fuzzy inference system described in “The Basic Tipping Problem” on page 2-16 from scratch, type the following command at the MATLAB prompt:

```
fuzzyLogicDesigner
```

The generic untitled **Fuzzy Logic Designer** opens, with one input **input1**, and one output **output1**.



Tip To open the **Fuzzy Logic Designer** with the prebuilt fuzzy inference system stored in `tipper.fis`, enter

```
fuzzyLogicDesigner('tipper.fis')
```

However, if you load the prebuilt system, you will not build rules or construct membership functions.

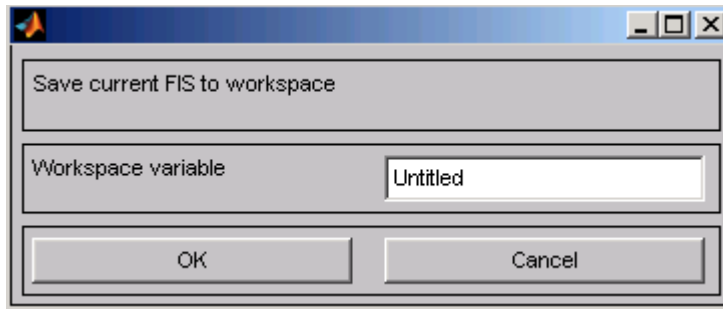
In this example, you construct a two-input, one output system. The two inputs are **service** and **food**. The one output is **tip**.

To add a second input variable and change the variable names to reflect these designations:

- 1 Select **Edit > Add variable > Input**.

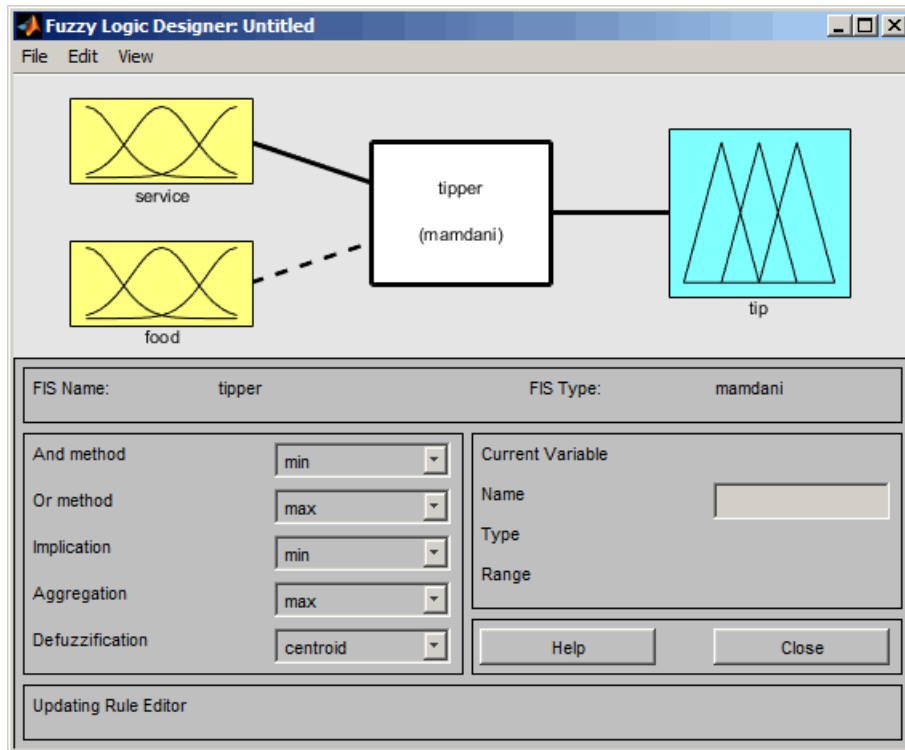
A second yellow box labeled **input2** appears.

- 2 Click the yellow box **input1**. This box is highlighted with a red outline.
- 3 Edit the **Name** field from input1 to **service**, and press **Enter**.
- 4 Click the yellow box **input2**. This box is highlighted with a red outline.
- 5 Edit the **Name** field from input2 to **food**, and press **Enter**.
- 6 Click the blue box **output1**.
- 7 Edit the **Name** field from output1 to **tip**, and press **Enter**.
- 8 Select **File > Export > To Workspace**.



- 9 Enter the **Workspace variable** name **tipper**, and click **OK**.

The diagram is 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 looks something like the following diagram.



Leave the inference options in the lower left in their default positions for now. You have entered all the information you need for this particular UI. Next, define the membership functions associated with each of the variables. To do this, open the Membership Function Editor.

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

- Within the **Fuzzy Logic Designer** window, select **Edit > Membership Functions**.
- Within the **Fuzzy Logic Designer** window, double-click the blue icon called **tip**.
- At the command line, type `mfedit`.

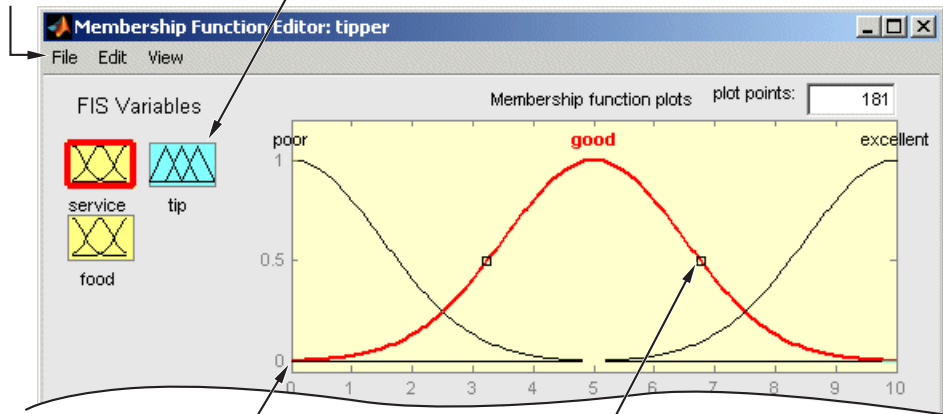
The Membership Function Editor

The Membership Function Editor is the tool that lets you display and edit all of the membership functions associated with all of the input and output variables for the entire

fuzzy inference system. The Membership Function Editor shares some features with the **Fuzzy Logic Designer**, as shown in the next figure. In fact, all of the five basic UI tools have similar menu options, status lines, and **Help** and **Close** buttons.

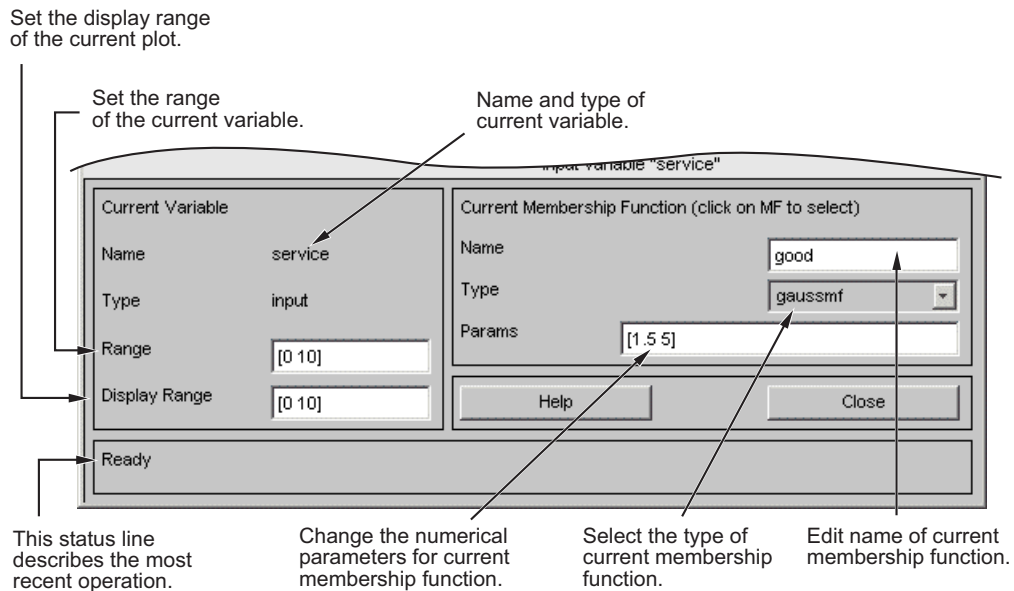
Menu commands for saving, opening, and editing a fuzzy system.

"Variable Palette" area. Click a variable to edit its membership functions.



Graph displays all membership functions for the selected variable.

Click a line to change its attributes, such as name, type, and numerical parameters. Drag the curve to move it or to change its shape.



When you open the Membership Function Editor to work on a fuzzy inference system that does not already exist in the workspace, there are no membership functions associated with the variables that you defined with the **Fuzzy Logic Designer**.

On the upper-left side of the graph area in the Membership Function Editor is a "Variable Palette" that lets you set the membership functions for a given variable.

To set up the membership functions associated with an input or an output variable for the FIS, select a FIS variable in this region by clicking it.

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

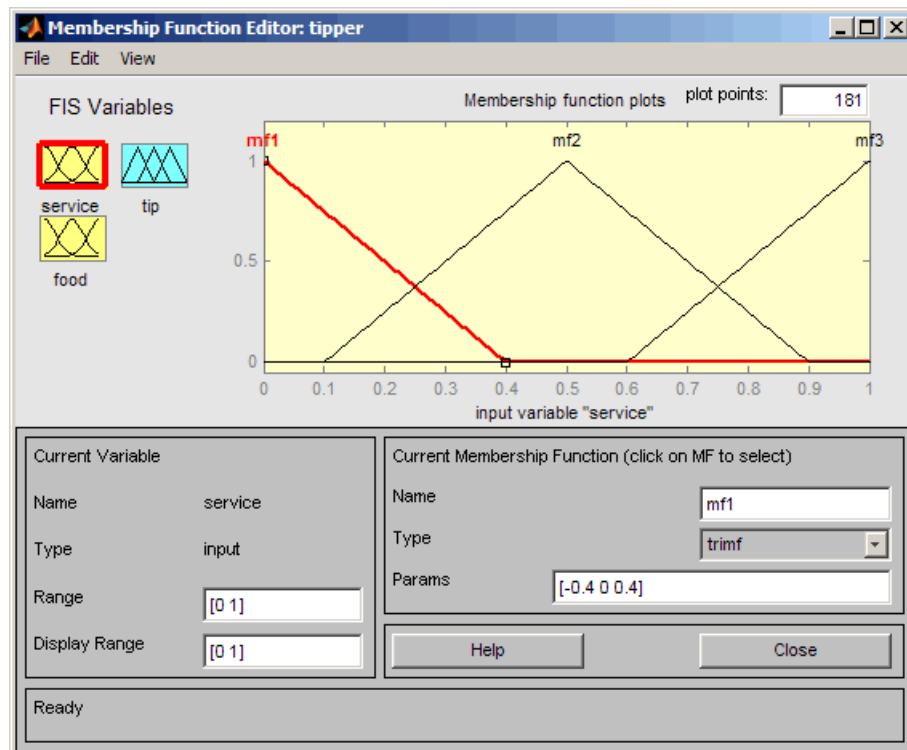
The membership functions from the current variable are displayed in the main graph. These membership functions can be manipulated in two ways. You can first use the mouse to select a particular membership function associated with a given variable quality, (such as poor, for the variable, service), and then drag the membership function from side to side. This action affects the mathematical description of the quality associated with that

membership function for a given variable. The selected membership function can also be tagged for dilation or contraction by clicking on the small square drag points on the membership function, and then dragging the function with the mouse toward the *outside*, for dilation, or toward the *inside*, for contraction. This action changes the parameters associated with that membership function.

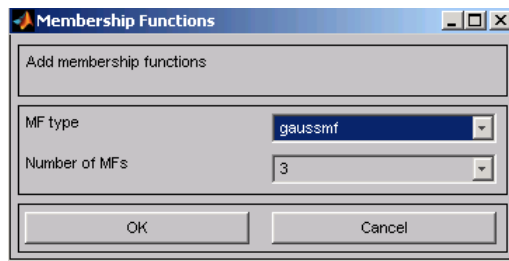
Below the Variable Palette is some information about the type and name of the current variable. There is a text field in this region that lets you change the limits of the current variable's range (universe of discourse) and another that lets you set the limits of the current plot (which has no real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

The process of specifying the membership functions for the two-input tipping example, *tipper*, is as follows:

- 1 Double-click the input variable *service* to open the Membership Function Editor.



- 2 In the Membership Function Editor, enter [0 10] in the **Range** and the **Display Range** fields.
- 3 Create membership functions for the input variable `service`.
 - a Select **Edit > Remove All MFs** to remove the default membership functions for the input variable `service`.
 - b Select **Edit > Add MFs** to open the Membership Functions dialog box.
 - c In the Membership Functions dialog box, select `gaussmf` as the **MF Type**.



- d Verify that 3 is selected as the **Number of MFs**.
 - e Click **OK** to add three Gaussian curves to the input variable `service`.
- 4 Rename the membership functions for the input variable `service`, and specify their parameters.
 - a Click on the curve named `mf1` to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
 - In the **Name** field, enter `poor`.
 - In the **Params** field, enter [1.5 0].

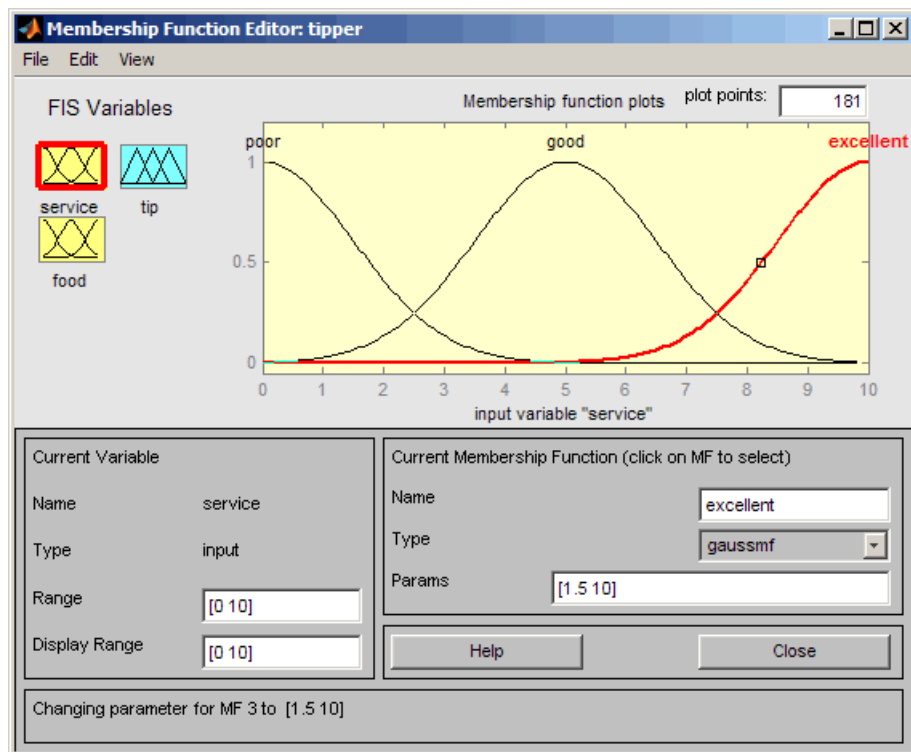
The two inputs of **Params** represent the standard deviation and center for the Gaussian curve.

Tip To adjust the shape of the membership function, type in the desired parameters or use the mouse, as described previously.

- b Click on the curve named `mf2` to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
 - In the **Name** field, enter `good`.

- In the **Params** field, enter [1.5 5].
- c Click on the curve named mf3, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
 - In the **Name** field, enter excellent.
 - In the **Params** field, enter [1.5 10].

The Membership Function Editor window looks similar to the following figure.



- 5 In the **FIS Variables** area, click the input variable **food** to select it.
- 6 Enter [0 10] in the **Range** and the **Display Range** fields.
- 7 Create the membership functions for the input variable **food**.
 - a Select **Edit > Remove All MFs** to remove the default Membership Functions for the input variable **food**.

- b** Select **Edit > Add MFs** to open the Membership Functions dialog box.
 - c** In the Membership Functions dialog box, select `trapmf` as the **MF Type**.
 - d** Select 2 in the **Number of MFs** drop-down list.
 - e** Click **OK** to add two trapezoidal curves to the input variable `food`.
- 8** Rename the membership functions for the input variable `food`, and specify their parameters:
- a** In the **FIS Variables** area, click the input variable `food` to select it.
 - b** Click on the curve named `mf1`, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
 - In the **Name** field, enter `rancid`.
 - In the **Params** field, enter `[0 0 1 3]`.
 - c** Click on the curve named `mf2` to select it, and enter `delicious` in the **Name** field.

Reset the associated parameters if desired.

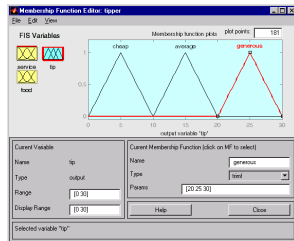
- 9** Click on the output variable `tip` to select it.
- 10** Enter `[0 30]` in the **Range** and the **Display Range** fields to cover the output range.

The inputs ranges from 0 to 10, but the output is a tip between 5% and 25%.

- 11** Rename the default triangular membership functions for the output variable `tip`, and specify their parameters.
- a** Click the curve named `mf1` to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
 - In the **Name** field, enter `cheap`.
 - In the **Params** field, enter `[0 5 10]`.
 - b** Click the curve named `mf2` to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
 - In the **Name** field, enter `average`.
 - In the **Params** field, enter `[10 15 20]`.
 - c** Click the curve named `mf3` to select it, and specify the following:
 - In the **Name** field, enter `generous`.

- In the **Params** field, enter [20 25 30].

The Membership Function Editor looks similar to the following figure.



Now that the variables have been named and the membership functions have appropriate shapes and names, you can enter the rules. To call up the Rule Editor, go to the **Edit** menu and select **Rules**, or type `ruleedit` at the command line.

The Rule Editor

The menu items allow you to save, open, or edit a fuzzy system using any of the five basic GUI tools.

Input or output selection menus.

The rules are entered automatically using the GUI tools.

Link input statements in rules.

This status line describes the most recent operation.

Negate input or output statements in rules.

Create or edit rules with the GUI buttons and choices from the input or output selection menus.

The Help button gives some information about how the Rule Editor works, and the Close button closes the window.

Constructing rules using the graphical Rule Editor interface is fairly self evident. Based on the descriptions of the input and output variables defined with the **Fuzzy Logic Designer**, the Rule Editor allows you to construct the rule statements automatically. You can:

- Create rules by selecting an item in each input and output variable box, selecting one **Connection** item, and clicking **Add Rule**. You can choose **none** as one of the variable qualities to exclude that variable from a given rule and choose **not** under any variable name to negate the associated quality.
- Delete a rule by selecting the rule and clicking **Delete Rule**.
- Edit a rule by changing the selection in the variable box and clicking **Change Rule**.
- Specify weight to a rule by typing in a desired number between 0 and 1 in **Weight**. If you do not specify the weight, it is assumed to be unity (1).

Similar to those in the **Fuzzy Logic Designer** and the Membership Function Editor, the Rule Editor has the menu bar and the status line. The menu items allow you to open, close, save and edit a fuzzy system using the five basic UI tools. From the menu, you can also:

- Set the format for the display by selecting **Options > Format**.
- Set the language by selecting **Options > Language**.

You can access information about the Rule Editor by clicking **Help** and close the UI using **Close**.

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

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

Then, click **Add rule**.

The resulting rule is

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

The numbers in the parentheses represent weights.

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

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

Tip To change a rule, first click on the rule to be changed. Next make the desired changes to that rule, and then click **Change rule**. For example, to change the first rule to *1. If (service not poor) or (food not rancid) then (tip is not cheap) (1)*

Select the **not** check box under each variable, and then click **Change rule**.

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

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

There is not much difference in the display really, but it is slightly more language neutral, because it does not depend on terms like *if* and *then*. If you change the format to indexed, you see an extremely compressed version of the rules.

- 1 1, 1 (1) : 2
- 2 0, 2 (1) : 1
- 3 2, 3 (1) : 2

This is the version of the rules that the machine deals with.

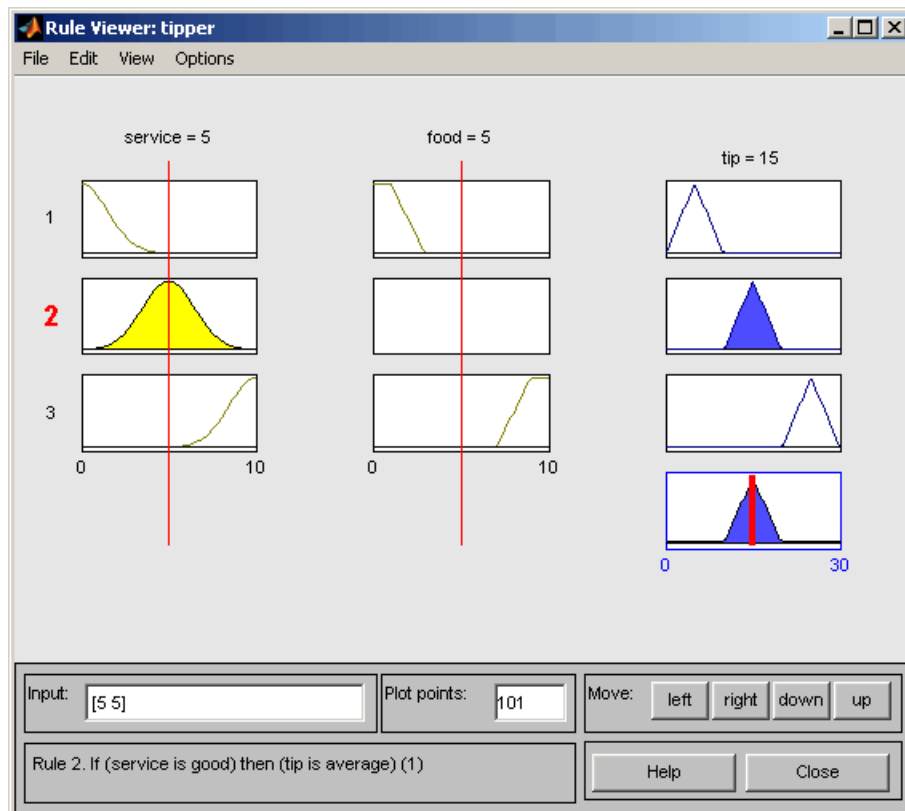
- The first column in this structure corresponds to the input variables.
- The second column corresponds to the output variable.
- The third column displays the weight applied to each rule.
- The fourth column is shorthand that indicates whether this is an OR (2) rule or an AND (1) rule.
- The numbers in the first two columns refer to the index number of the membership function.

A literal interpretation of rule 1 is "If input 1 is MF1 (the first membership function associated with input 1) or if input 2 is MF1, then output 1 should be MF1 (the first membership function associated with output 1) with the weight 1."

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

At this point, the fuzzy inference system has been completely defined, in that the variables, membership functions, and the rules necessary to calculate tips are in place. Now, look at the fuzzy inference diagram presented at the end of the previous section and verify that everything is behaving the way you think it should. You can use the Rule Viewer, the next of the UI tools we'll look at. From the **View** menu, select **Rules**.

The Rule Viewer



The Rule Viewer displays a roadmap of the whole fuzzy inference process. It is based on the fuzzy inference diagram described in the previous section. You see a single figure window with 10 plots nested in it. The three plots across the top of the figure represent the antecedent and consequent of the first rule. Each rule is a row of plots, and each

column is a variable. The rule numbers are displayed on the left of each row. You can click on a rule number to view the rule in the status line.

- The first two columns of plots (the six yellow plots) show the membership functions referenced by the antecedent, or the if-part of each rule.
- The third column of plots (the three blue plots) shows the membership functions referenced by the consequent, or the then-part of each rule.

Notice that under **food**, there is a plot which is blank. This corresponds to the characterization of **none** for the variable **food** in the second rule.

- The fourth plot in the third column of plots represents the aggregate weighted decision for the given inference system.

This decision will depend on the input values for the system. The defuzzified output is displayed as a bold vertical line on this plot.

The variables and their current values are displayed on top of the columns. In the lower left, there is a text field **Input** in which you can enter specific input values. For the two-input system, you will enter an input vector, [9 8], for example, and then press **Enter**. You can also adjust these input values by clicking on any of the three plots for each input. This will move the red index line horizontally, to the point where you have clicked. Alternatively, you can also click and drag this line in order to change the input values. When you release the line, (or after manually specifying the input), a new calculation is performed, and you can see the whole fuzzy inference process take place:

- Where the index line representing service crosses the membership function line "service is poor" in the upper-left plot determines the degree to which rule one is activated.
- A yellow patch of color under the actual membership function curve is used to make the fuzzy membership value visually apparent.

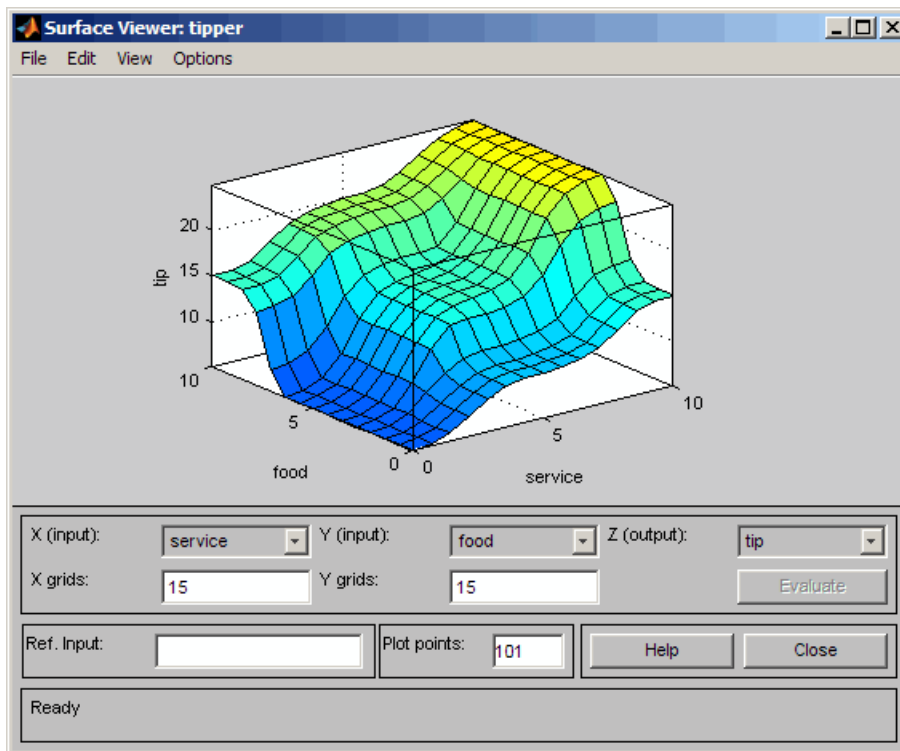
Each of the characterizations of each of the variables is specified with respect to the input index line in this manner. If you follow rule 1 across the top of the diagram, you can see the consequent "tip is cheap" has been truncated to exactly the same degree as the (composite) antecedent — this is the implication process in action. The aggregation occurs down the third column, and the resultant aggregate plot is shown in the single plot appearing 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.

You can shift the plots using **left**, **right**, **down**, and **up**. The menu items allow you to save, open, or edit a fuzzy system using any of the five basic UI tools.

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

The Rule Viewer shows one calculation at a time and in great detail. In this sense, it presents a sort of micro view of the fuzzy inference system. If you want to see the entire output surface of your system — 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 viewer is the last of the five basic Fuzzy Logic Toolbox UI tools. To open the Surface Viewer, select **Surface** from the **View** menu.

The Surface Viewer



Upon opening the Surface Viewer, you see a three-dimensional curve that represents the mapping from food and service quality to tip amount. Because this curve represents a two-input one-output case, you can see the entire mapping in one plot. When we move beyond three dimensions overall, we start to encounter trouble displaying the results.

Accordingly, the Surface Viewer is equipped with drop-down menus **X (input)**, **Y (input)** and **Z (output)** that let you select any two inputs and any one output for plotting. Below these menus are two input fields **X grids** and **Y grids** that let you specify how many x-axis and y-axis grid lines you want to include. This capability allows you to keep the calculation time reasonable for complex problems.

By default, the surface plot updates automatically when you change the input or output variable selections or the number of grid points. To disable automatic plot updates, in the **Options** menu, clear the **Always evaluate** option. When this option is disabled, to update the plot, click **Evaluate**.

If you want to create a smoother plot, use the **Plot points** field to specify the number of points on which the membership functions are evaluated in the input or output range. This field defaults to the minimum number of plot plots, 101. If you specify fewer plot points, the field value automatically resets to 101. When you specify the number of plot points, the surface plot automatically updates.

By clicking on the plot axes and dragging the mouse, you can manipulate the surface so that you can view it from different angles.

The **Ref. Input** field is used in situations when there are more inputs required by the system than the surface is mapping. You can edit this field to explicitly set inputs not specified in the surface plot.

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 because computer monitors cannot display a five-dimensional shape. In such a case, the input is a four-dimensional vector with NaNs holding the place of the varying inputs while numerical values indicates those values that remain fixed.

The menu items allow you to open, close, save and edit a fuzzy system using the five basic UI tools. You can access information about the Surface Viewer by clicking **Help** and close the UI using **Close**.

Importing and Exporting Fuzzy Inference Systems

When you save a fuzzy system to a file, you are saving an ASCII text FIS file representation of that system with the file suffix `.fis`. Do not manually edit the contents of a `.fis` file. Doing so can produce unexpected results when loading the file. When you save your fuzzy system to the MATLAB workspace, you are creating a variable that acts as a MATLAB object for the fuzzy system.

Note If you do not save your FIS to a file, but only save it to the MATLAB workspace, you cannot recover it for use in a new MATLAB session.

See Also

Fuzzy Logic Designer

More About

- “What Is Mamdani-Type Fuzzy Inference?” on page 2-4
- “Build Fuzzy Systems at the Command Line” on page 2-38
- “Simulate Fuzzy Inference Systems in Simulink” on page 5-2

Build Fuzzy Systems at the Command Line

You can construct a fuzzy inference system (FIS) at the MATLAB® command line. This method is an alternative to interactively designing your FIS using Fuzzy Logic Designer.

This example creates a Mamdani fuzzy inference system. While the example creates a Mamdani FIS, the methods used apply to creating Sugeno systems as well.

Tippling Problem at the Command Line

To demonstrate the command-line functionality for creating and viewing fuzzy inference systems, this example uses the tipper FIS.

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

This command returns a `mamfis` object that contains the properties of the fuzzy system. For a Sugeno system, this command returns a `sugfis` object.

You can access the FIS properties using dot notation. For example, view the inputs of the fuzzy system.

```
fis.Inputs
```

```
ans =
```

```
1x2 fisvar array with properties:
```

```
Name  
Range  
MembershipFunctions
```

```
Details:
```

	Name	Range	MembershipFunctions
1	"service"	0 10	[1x3 fismf]
2	"food"	0 10	[1x2 fismf]

To set the properties of your fuzzy system, use dot notation. For example, set the name of the FIS.

```
fis.Name = "gratuity";
```

FIS Object

You represent fuzzy inference systems using `mamfis` and `sugfis` objects. These objects contain all the fuzzy inference system information, including the variable names, membership function definitions, and fuzzy inference methods. Each FIS is itself a hierarchy of objects. The following objects are used within a fuzzy system:

- `fisvar` objects represent both input and output variables.
- `fismf` objects represent membership functions within each input and output variable.
- `fisrule` objects represent fuzzy rules that map inputs to outputs.

View all the information for a FIS by directly listing its properties.

```
fis
```

```
fis =
```

```
    mamfis with properties:
```

```

        Name: "gratuity"
    AndMethod: "min"
    OrMethod: "max"
    ImplicationMethod: "min"
    AggregationMethod: "max"
    DefuzzificationMethod: "centroid"
        Inputs: [1x2 fisvar]
        Outputs: [1x1 fisvar]
        Rules: [1x3 fisrule]
    DisableStructuralChecks: 0
```

You can view the properties of the objects within a FIS object using dot notation. For example, view the `fisvar` object for first input variable.

```
fis.Inputs(1)
```

```
ans =
```

```
    fisvar with properties:
```

```

        Name: "service"
    Range: [0 10]
```

```
MembershipFunctions: [1x3 fismf]
```

Also, view the membership functions for this variable.

```
fis.Inputs(1).MembershipFunctions
```

```
ans =
```

```
1x3 fismf array with properties:
```

```
Name  
Type  
Parameters
```

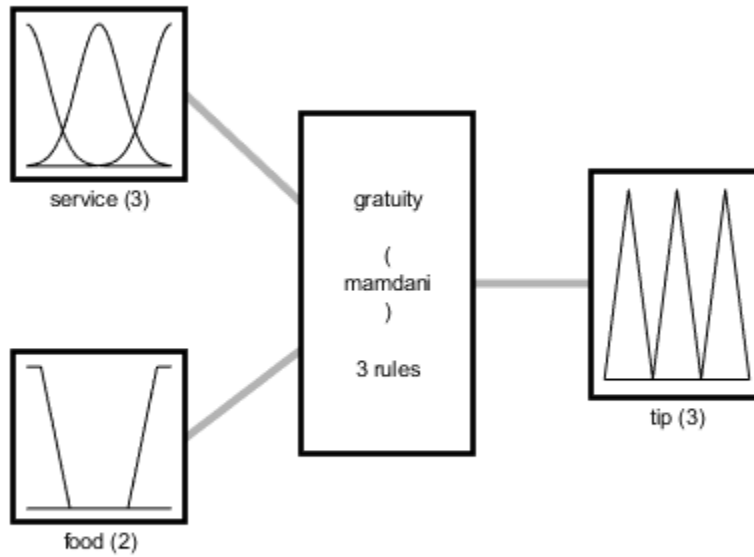
```
Details:
```

	Name	Type	Parameters	
	_____	_____	_____	_____
1	"poor"	"gaussmf"	1.5	0
2	"good"	"gaussmf"	1.5	5
3	"excellent"	"gaussmf"	1.5	10

System Display Functions

To get a high-level view of your fuzzy system from the command line, use the `plotfis`, `plotmf`, and `gensurf` functions. `plotfis` displays the whole system as a block diagram, as shown in the **Fuzzy Logic Designer**.

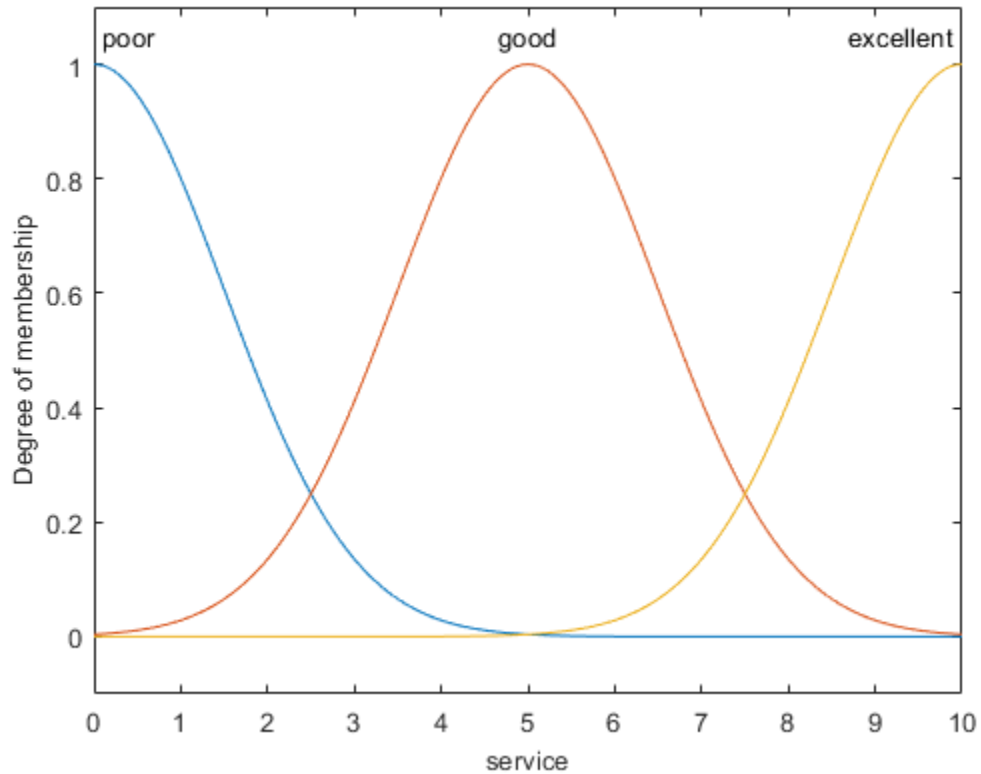
```
plotfis(fis)
```



System gratuity: 2 inputs, 1 outputs, 3 rules

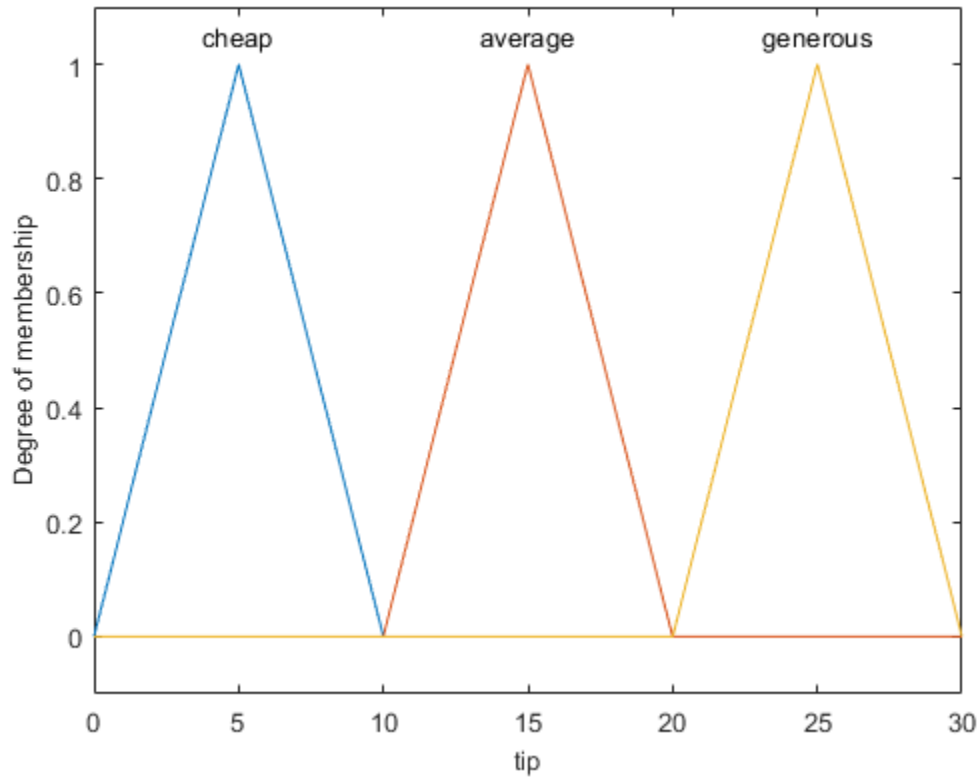
The `plotmf` function plots all the membership functions associated with a given variable. For example, view the membership functions for the first input variable.

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



Similarly, to view the membership functions for the first output, type:

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



`plotmf` does not support viewing the output membership functions for Sugeno systems.

To view the rules of the fuzzy system, type:

```
fis.Rules
```

```
ans =
```

```
1x3 fisrule array with properties:
```

```
Description  
Antecedent  
Consequent
```

Weight
Connection

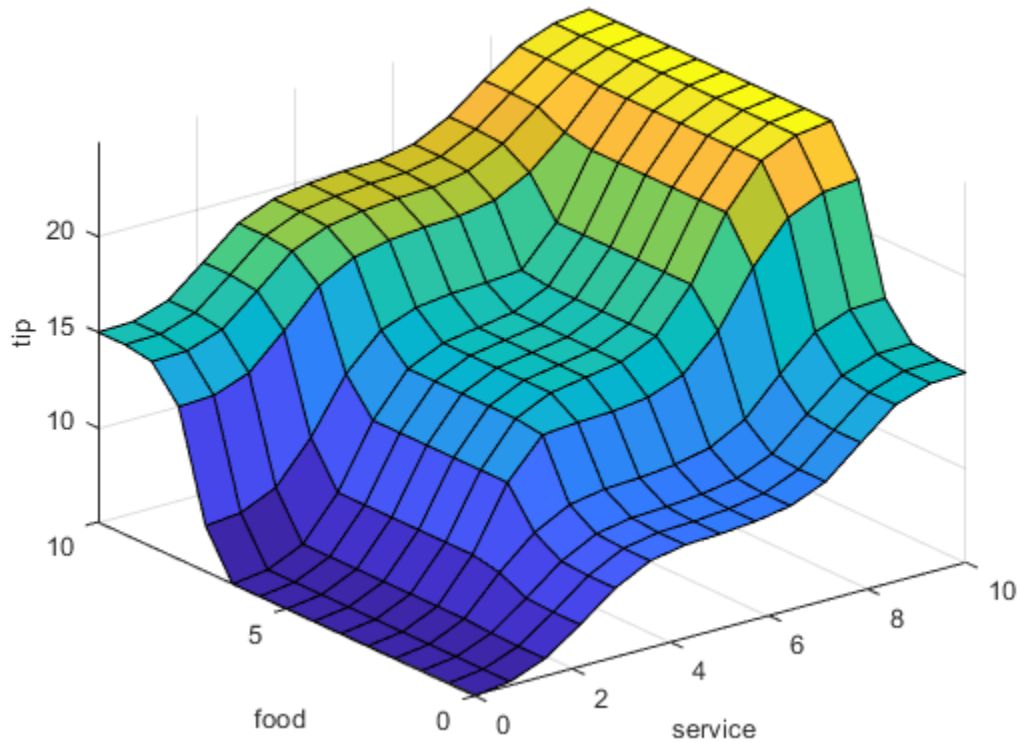
Details:

Description

1	"service==poor food==rancid => tip=cheap (1)"
2	"service==good => tip=average (1)"
3	"service==excellent food==delicious => tip=generous (1)"

The `gensurf` function plots the output of the FIS for any one or two input variables.

`gensurf(fis)`



Build Fuzzy Inference System

As an alternative to using the **Fuzzy Logic Designer** app, you can construct a FIS entirely from the command line.

First, create a Mamdani FIS, specifying its name.

```
fis = mamfis('Name', "tipper");
```

Add the first input variable for the service quality using `addInput`.

```
fis = addInput(fis,[0 10], 'Name', "service");
```

Add membership functions for each of the service quality levels using `addMF`. In this case, use Gaussian membership functions. For more information on Gaussian membership function properties, see `gaussmf`.

```
fis = addMF(fis,"service","gaussmf",[1.5 0],'Name',"poor");
fis = addMF(fis,"service","gaussmf",[1.5 5],'Name',"good");
fis = addMF(fis,"service","gaussmf",[1.5 10],'Name',"excellent");
```

Add the second input variable for the food quality, and add two trapezoidal membership functions. For information on trapezoidal membership functions, see `trapmf`.

```
fis = addInput(fis,[0 10],'Name',"food");
fis = addMF(fis,"food","trapmf",[-2 0 1 3],'Name',"rancid");
fis = addMF(fis,"food","trapmf",[7 9 10 12],'Name',"delicious");
```

Add the output variable for the tip, and add three triangular membership functions. For more information on the triangular membership function, see `trimf`.

```
fis = addOutput(fis,[0 30],'Name',"tip");
fis = addMF(fis,"tip","trimf",[0 5 10],'Name',"cheap");
fis = addMF(fis,"tip","trimf",[10 15 20],'Name',"average");
fis = addMF(fis,"tip","trimf",[20 25 30],'Name',"generous");
```

Specify the following three rules for the FIS as a numeric array:

- 1 If (service is poor) or (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).

Each row of the array contains one rule in the following format.

- Column 1 - Index of membership function for first input
- Column 2 - Index of membership function for second input
- Column 3 - Index of membership function for output
- Column 4 - Rule weight (from 0 to 1)
- Column 5 - Fuzzy operator (1 for AND, 2 for OR)

For the membership function indices, indicate a NOT condition using a negative value. For more information on fuzzy rule specification, see `addRule`.

```
ruleList = [1 1 1 1 2;  
            2 0 2 1 1;  
            3 2 3 1 2];
```

Add the rules to the FIS.

```
fis = addRule(fis,ruleList);
```

Alternatively, you can create the fuzzy inference system using a combination of dot notation and `fisvar`, `fismf`, and `fisrule` objects. This method is not recommended for most applications. However, you can use this approach when your application requires greater flexibility in constructing and modifying your FIS.

Create the fuzzy inference system.

```
fis = mamfis('Name','tipper');
```

Add and configure the first input variable. In this case, create a default `fisvar` object and specify its properties using dot notation.

```
fis.Inputs(1) = fisvar;  
fis.Inputs(1).Name = "service";  
fis.Inputs(1).Range = [0 10];
```

Define the membership functions for the first input variable. For each MF, create a `fismf` object, and set the properties using dot notation.

```
fis.Inputs(1).MembershipFunctions(1) = fismf;  
fis.Inputs(1).MembershipFunctions(1).Name = "poor";  
fis.Inputs(1).MembershipFunctions(1).Type = "gaussmf";  
fis.Inputs(1).MembershipFunctions(1).Parameters = [1.5 0];  
fis.Inputs(1).MembershipFunctions(2) = fismf;  
fis.Inputs(1).MembershipFunctions(2).Name = "good";  
fis.Inputs(1).MembershipFunctions(2).Type = "gaussmf";  
fis.Inputs(1).MembershipFunctions(2).Parameters = [1.5 5];  
fis.Inputs(1).MembershipFunctions(3) = fismf;  
fis.Inputs(1).MembershipFunctions(3).Name = "excellent";  
fis.Inputs(1).MembershipFunctions(3).Type = "gaussmf";  
fis.Inputs(1).MembershipFunctions(3).Parameters = [1.5 10];
```

Add and configure the second input variable. For this variable, specify the name and range when you create the `fisvar` object.

```
fis.Inputs(2) = fisvar([0 10],'Name',"food");
```

Specify the membership functions for the second input. For each MF, specify the name, type, and parameters when you create the `fismf` object.

```
fis.Inputs(2).MembershipFunctions(1) = fismf("trapmf",[-2 0 1 3],...  
                                           'Name',"rancid");  
fis.Inputs(2).MembershipFunctions(2) = fismf("trapmf",[7 9 10 12],...  
                                           'Name',"delicious");
```

Similarly, add and configure the output variable and its membership functions.

```
fis.Outputs(1) = fisvar([0 30],'Name',"tip");
```

In this case, specify the output membership functions using a vector of `fismf` objects.

```
mf1 = fismf("trimf",[0 5 10],'Name',"cheap");  
mf2 = fismf("trimf",[10 15 20],'Name',"average");  
mf3 = fismf("trimf",[20 25 30],'Name',"generous");  
fis.Outputs(1).MembershipFunctions = [mf1 mf2 mf3];
```

Create the rules for the fuzzy system. For each rule create a `fisrule` object. Then, specify the rules using a vector of these objects. When creating a `fisrule` object using numeric values, you must specify the number of inputs variables.

```
rule1 = fisrule([1 1 1 1 2],2);  
rule2 = fisrule([2 0 2 1 1],2);  
rule3 = fisrule([3 2 3 1 2],2);  
rules = [rule1 rule2 rule3];
```

Before adding your rules to your fuzzy system, you must update them using the data in the FIS object. Update the rules and add them the fuzzy system.

```
rules = update(rules,fis);  
fis.Rules = rules;
```

When constructing your fuzzy system, you can also specify custom membership functions and inference functions. For more information, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

Evaluate Fuzzy Inference System

To evaluate the output of a fuzzy system for a given input combination, use the `evalfis` command. For example, evaluate `fis` using input variable values of 1 and 2.

```
evalfis(fis,[1 2])
```

```
ans =  
    5.5586
```

You can also evaluate multiple input combinations using an array where each row represents one input combination.

```
inputs = [3 5;  
          2 7;  
          3 1];  
evalfis(fis,inputs)
```

```
ans =  
    12.2184  
     7.7885  
     8.9547
```

See Also

[evalfis](#) | [gensurf](#) | [mamfis](#) | [plotfis](#) | [plotmf](#) | [sugfis](#)

More About

- “What Is Mamdani-Type Fuzzy Inference?” on page 2-4
- “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14

Build Fuzzy Systems Using Custom Functions

Build Fuzzy Inference Systems Using Custom Functions in Fuzzy Logic Designer

When you build a fuzzy inference system, as described in “Fuzzy Inference Process” on page 1-28, you can replace the built-in membership functions, inference functions, or both with custom functions. In this section, you learn how to build a fuzzy inference system using custom functions in the **Fuzzy Logic Designer** app.

To build a fuzzy inference system using custom functions in the **Fuzzy Logic Designer** app:

- 1 Open **Fuzzy Logic Designer**. At the MATLAB command line, type:

```
fuzzyLogicDesigner
```

- 2 Specify the number of inputs and outputs of the fuzzy system, as described in “The Fuzzy Logic Designer” on page 2-17.
- 3 Create custom membership functions, and replace the built-in membership functions with them, as described in “Specify Custom Membership Functions” on page 2-51.

Membership functions define how each point in the input space is mapped to a membership value between 0 and 1.

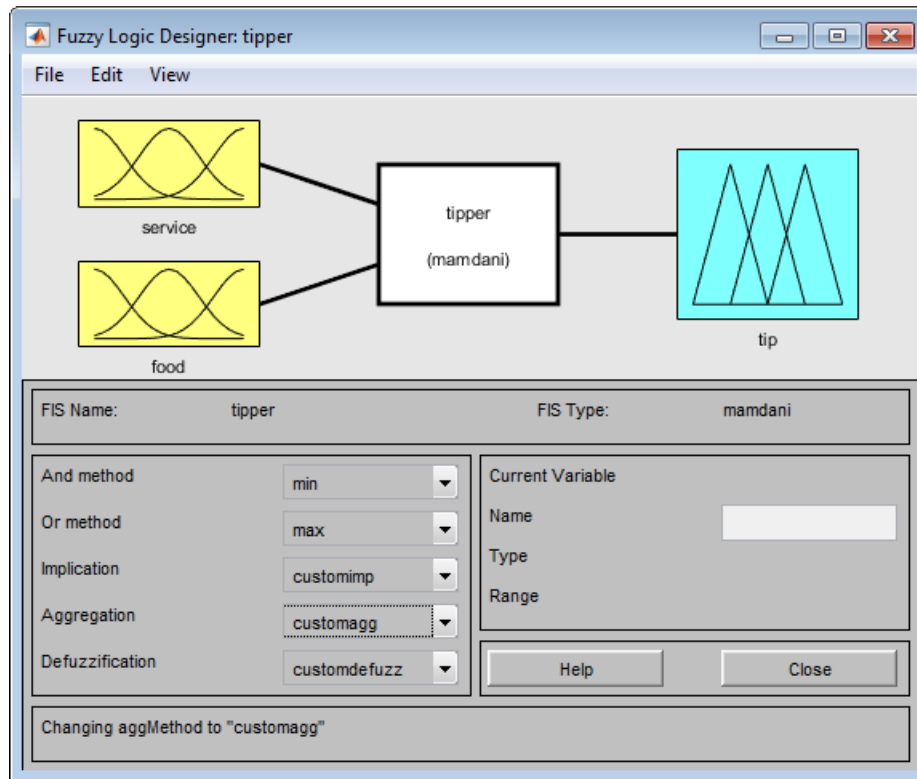
- 4 Create rules using the Rule Editor, as described in “The Rule Editor” on page 2-30.

Rules define the logical relationship between the inputs and the outputs.

- 5 Create custom inference functions, and replace the built-in inference functions with them, as described in “Specify Custom Inference Functions” on page 2-57.

Inference methods include the AND, OR, implication, aggregation, and defuzzification methods. This action generates the output values for the fuzzy system.

The next figure shows the tipping problem example where the built-in **Implication**, **Aggregation** and **Defuzzification** functions are replaced with the custom functions, `customimp`, `customagg`, and `customdefuzz`, respectively.



- 6 Select **View > Surface** to view the output of the fuzzy inference system in the Surface Viewer, as described in “The Surface Viewer” on page 2-35.

Specify Custom Membership Functions

You can create custom membership functions and use them in the fuzzy inference process. The values of these functions must lie between 0 and 1. For more information on the properties of membership functions, see “Membership Functions” on page 1-14.

To create a custom membership function, and replace the built-in membership function:

- 1 Create a MATLAB function, and save it in your current working folder.

To learn how to create MATLAB functions, see “Scripts vs. Functions” (MATLAB).

The following code is an example of a multistep custom membership function, `custmf1`, that depends on eight parameters between 0 and 10.

```
% Function to generate a multi-step custom membership function
% using 8 parameters for the input argument x
function out = custmf1(x,params)

for i = 1:length(x)
    if x(i) < params(1)
        y(i) = params(1);
    elseif x(i) < params(2)
        y(i) = params(2);
    elseif x(i) < params(3)
        y(i) = params(3);
    elseif x(i) < params(4)
        y(i) = params(4);
    elseif x(i) < params(5)
        y(i) = params(5);
    elseif x(i) < params(6)
        y(i) = params(6);
    elseif x(i) < params(7)
        y(i) = params(7);
    elseif x(i) < params(8)
        y(i) = params(8);
    else
        y(i) = 0;
    end
end

out = 0.1*y'; % Scale the output to lie between 0 and 1.
```

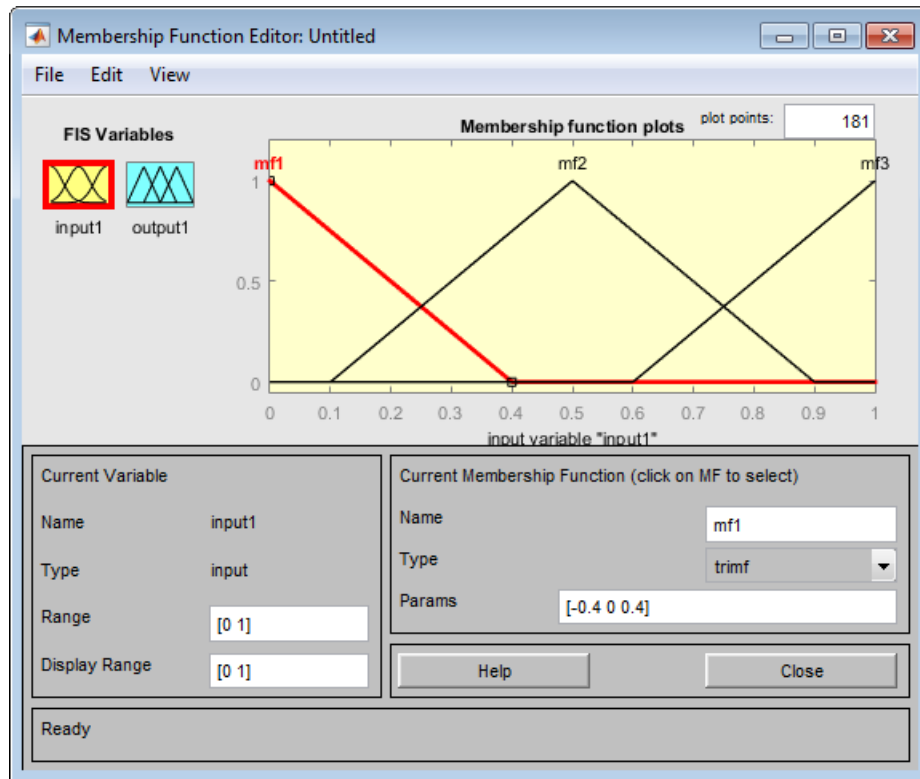
- 2 Open the **Fuzzy Logic Designer** app.

```
fuzzyLogicDesigner
```

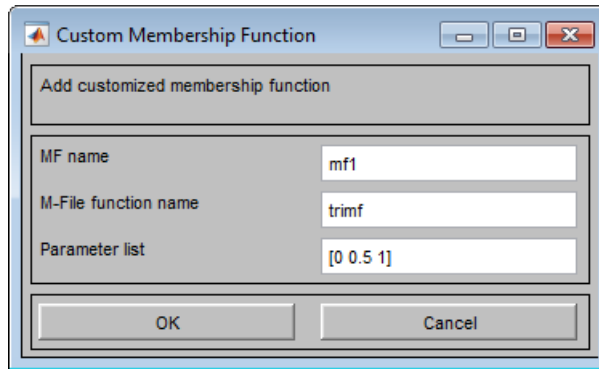
The **Fuzzy Logic Designer** opens with the default FIS name, `Untitled`, and contains one input, `input1`, and one output, `output1`.

- 3 In the **Fuzzy Logic Designer**, select **Edit > Membership Functions** to open the Membership Function Editor.

Three triangular-shaped membership functions for `input1` are displayed by default.



- 4 To replace the default membership function with a custom function in the Membership Function Editor:
 - a Select **Edit > Remove All MFs** to remove the default membership functions for **input1**.
 - b Select **Edit > Add Custom MF** to open the Custom Membership Function dialog box.



- 5 To specify a custom function, in the Custom Membership Function dialog box:
- a In the **MF name** field, specify a name for the custom membership function.

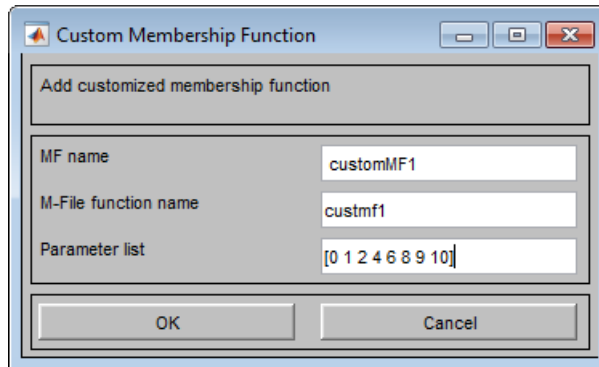
Note When adding additional custom membership functions, specify a different **MF name** for each function.

- b In the **M-file function name** field, specify the name of the custom membership function file.
- c In the **Parameter list**, specify a vector of parameters.

These values determine the shape and position of the membership function, and the function is evaluated using these parameter values.

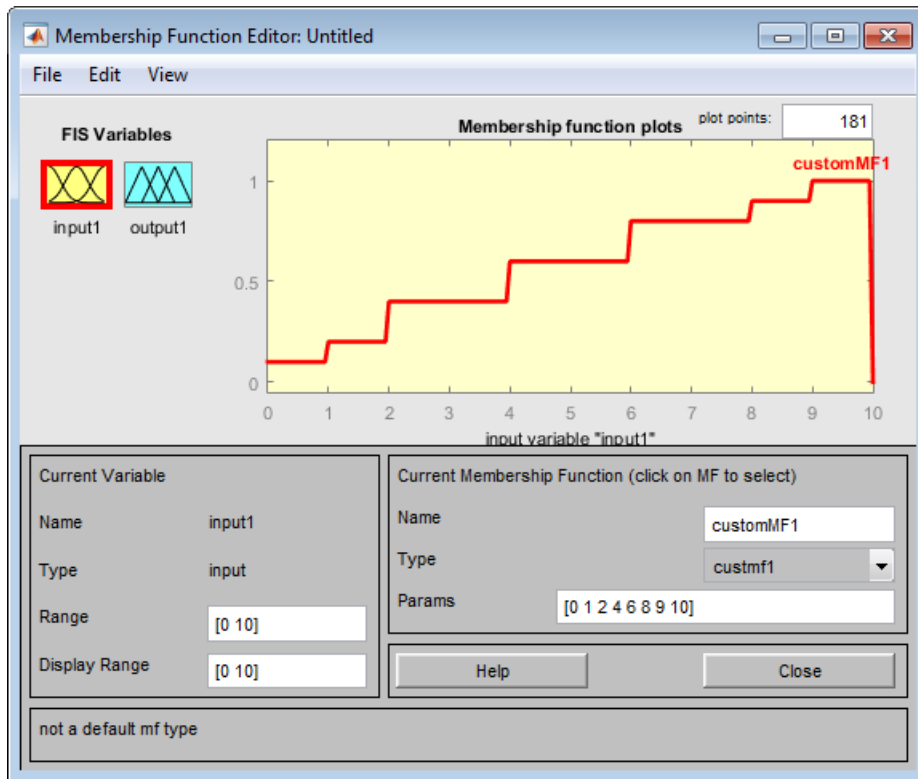
Note The length of the parameter vector must be greater than or equal to the number of parameters in the custom membership function.

Using the `custmf1` example in step 1, the Custom Membership Function dialog box looks similar to the following figure.

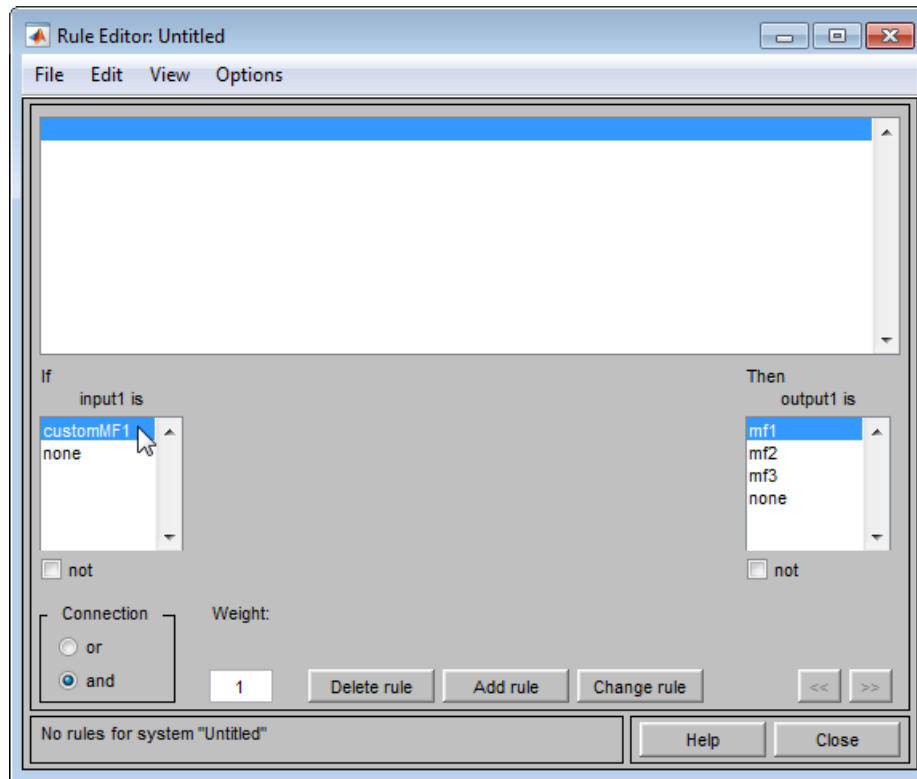


- d Click **OK** to add the custom membership function.
- e Specify both the **Range** and **Display Range** to be [0 10] to match the range of the custom membership function.

The Membership Function Editor displays the custom membership function plot.



This action also adds the custom membership function to the Rule Viewer, and makes it available for creating rules for the fuzzy inference process. To view the custom function in the Rule Viewer, select **Edit > Rules** in either the **Fuzzy Logic Designer** or the Membership Function Editor.



- 6 To add custom membership functions for **output1**, select it in the Membership Function Editor, and repeat steps 4 and 5.

You can also add a custom membership function to a FIS at the MATLAB command line. For example, to add `custmf1` to the first input variable, `input1` of the FIS, `myFIS`, and name it `customMF1`, type the following:

```
myFIS = addMF(myFIS,"input1","custmf1",[0 1 2 4 6 8 9 10],'Name',"customMF1");
```

Specify Custom Inference Functions

You can replace the built-in AND, OR, implication, aggregation, and defuzzification inference methods with custom functions. After you create the custom inference function, save it in your current working folder. To learn how to build fuzzy systems using custom inference functions, see the “Build Fuzzy Inference Systems Using Custom Functions in Fuzzy Logic Designer” on page 2-50 section.

The guidelines for creating and specifying the functions for building fuzzy inference systems are described in the following sections.

- “Create Custom AND and OR Functions” on page 2-58
- “Create Custom Implication Functions” on page 2-59
- “Create Custom Aggregation Functions” on page 2-60
- “Create Custom Defuzzification Functions” on page 2-61
- “Steps for Specifying Custom Inference Functions” on page 2-61

Create Custom AND and OR Functions

The custom AND and OR inference functions must operate column-wise on a matrix, in the same way as the MATLAB functions `max`, `min`, or `prod`.

For a row or column vector x , `min(x)` returns the minimum element.

```
x = [1 2 3 4];  
min(x)  
  
ans =  
    1
```

For a matrix x , `min(x)` returns a row vector containing the minimum element from each column.

```
x = [1 2 3 4;5 6 7 8;9 10 11 12];  
min(x)  
  
ans =  
    1     2     3     4
```

For N-D arrays, `min(x)` operates along the first non-singleton dimension.

The function `min(x,y)` returns an array that is same size as x and y with the minimum elements from x or y . Either of the input arguments can be a scalar. Functions such as `max`, and `prod` operate in a similar manner.

In the toolbox, the AND implication methods perform an element by element matrix operation, similar to the MATLAB function `min(x,y)`.

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

```
ans =
     1     2
     2     2
```

The OR implication methods perform an element by element matrix operation, similar to the MATLAB function `max(x, y)`.

Create Custom Implication Functions

Custom implication functions must operate in the same way as the MATLAB functions `max`, `min`, or `prod`. Your custom implication function must be a T -norm fuzzy intersection operation. For more information, see “Additional Fuzzy Operators” on page 1-21.

An implication function must support either one or two inputs because the software calls the function in two ways:

- To calculate the output fuzzy set values using the firing strength of all the rules and the corresponding output membership functions. In this case, the software calls the implication function using two inputs, similar to the following example:

```
impvals = customimp(w, outputmf)
```

- `w` — Firing strength of multiple rules, specified as an nr -by- ns matrix. Here, nr is the number of rules and ns is the number of samples of the output membership functions.

`w(:, j) = w(:, 1)` for all j . `w(i, 1)` is the firing strength of the i^{th} rule.

- `outputmf` — Output membership function values, specified as an nr -by- ns matrix. Here, nr is the number of rules and ns is the number of samples of the output membership functions.

`outputmf(i, :)` contains the data of the i^{th} output membership function.

- To calculate the output fuzzy value using the firing strength of a single rule and the corresponding output membership function, for a given sample. In this case, the software calls the implication function using one input, similar to the following example:

```
impval = customimp([w outputmf])
```

`w` and `outputmf` are scalar values representing the firing strength of a rule and the corresponding output membership function value, for a given sample.

The following is an example of a bounded product custom implication function with binary mapping $T(a,b) = \max\{0, a+b-1\}$. [1]

```
function y = customimp(x1,x2)

if nargin == 1
    % x1 assumed to be non-empty column vector or matrix.
    minVal = zeros(1,size(x1,2));
    y = ones(1,size(x1,2));

    for i = 1:size(x1,1)
        y = max(minVal,sum([y;x1(i,:)])-1);
    end
else
    % x1 and x2 assumed to be non-empty matrices.
    minVal = zeros(1,size(x1,2));
    y = zeros(size(x1));

    for i = 1:size(x1,1)
        y(i,:) = max(minVal,sum([x1(i,:);x2(i,:)])-1);
    end
end
```

Note Custom implication functions are not supported for Sugeno-type systems.

Create Custom Aggregation Functions

The custom aggregation functions must operate in the same way as the MATLAB functions `max`, `min`, or `prod` and must be of the form `y = customagg(x)`. Your custom implication function must be a T -conorm (S -norm) fuzzy intersection operation. For more information, see “Additional Fuzzy Operators” on page 1-21.

x is an nv -by- nr matrix, which is the list of truncated output functions returned by the implication method for each rule. nv is the number of output variables, and nr is the number of rules. The output of the aggregation method is one fuzzy set for each output variable.

The following is an example of a bounded sum custom aggregation function with binary mapping $S(a,b) = \min\{a+b,1\}$. [1]


```

function y = customagg(x)

maxVal = ones(1,size(x,2));
y = zeros(1,size(x,2));

for i = 1:size(x,1)
    y = min(maxVal,sum([y;x(i,:) ]));
end

```

Note Custom aggregation functions are not supported for Sugeno-type systems.

Create Custom Defuzzification Functions

The custom defuzzification functions must be of the form $y = \text{customdefuzz}(xmf, ymf)$, where (xmf, ymf) is a finite set of membership function values. xmf is the vector of values in the membership function input range. ymf is the value of the membership function at xmf .

The following is an example of a custom defuzzification function:

```

function defuzzfun = customdefuzz(xmf,ymf)

total_area = sum(ymf);
defuzzfun = sum(ymf.*xmf)/total_area;

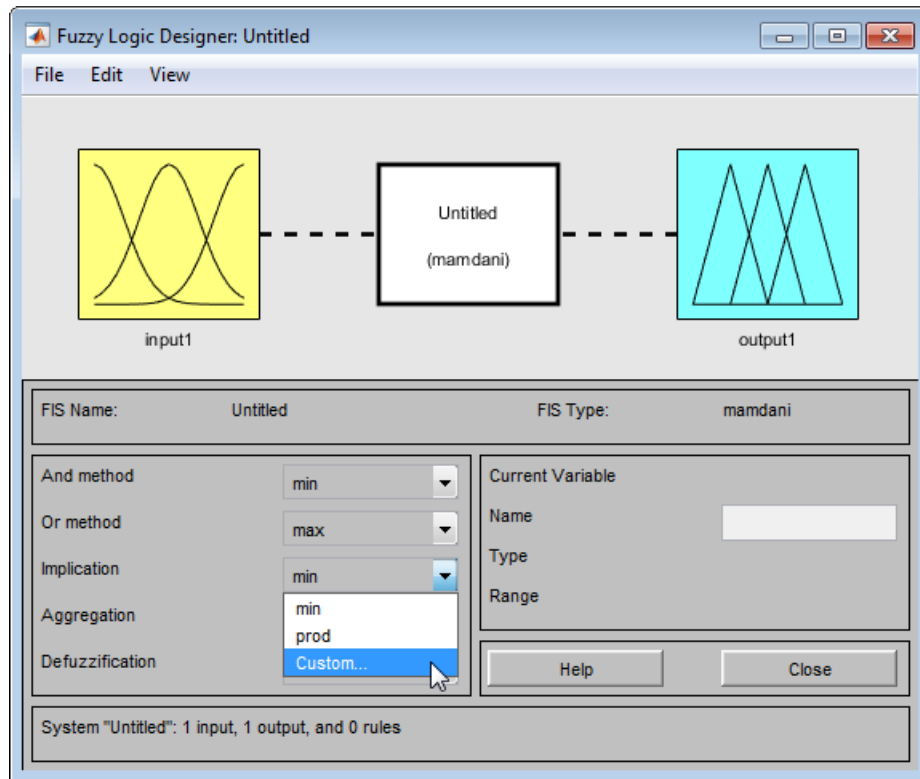
```

Note Custom defuzzification functions are not supported for Sugeno-type systems.

Steps for Specifying Custom Inference Functions

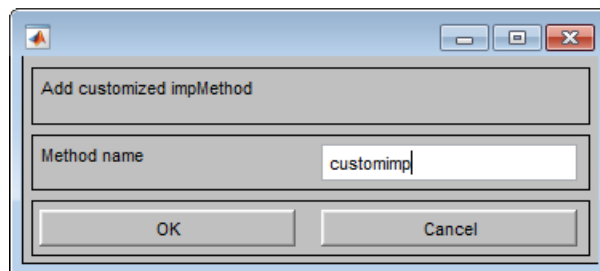
After you create and save a custom inference function, specify the function in the fuzzy inference system using the following steps:

- 1 In the lower-left panel of the **Fuzzy Logic Designer**, select Custom from the drop-down menu corresponding to the inference method for which you want to specify the custom function.



Doing so opens a dialog box where you specify the name of the custom inference function.

- 2 In the **Method name** field, specify the name of the custom inference function, and click **OK**.



The custom function replaces the built-in function when building the fuzzy inference system.

Note In order to specify a custom inference function, you must first add at least one rule to your FIS.

- 3 To specify custom functions for other inference methods, repeat steps 1 and 2.

You can also specify custom inference functions for a FIS at the MATLAB command line. For example, to add a custom:

- Defuzzification method, type

```
myFIS.DefuzzificationMethod = "customdefuzz";
```

where `customdefuzz` is the name of the custom defuzzification function.

- Implication method, type

```
myFIS.ImplicationMethod = "customimp";
```

where `customimp` is the name of the custom implication function.

- Aggregation method, type

```
myFIS.AggregationMethod = "customagg";
```

where `customagg` is the name of the custom aggregation function.

Use Custom Functions in Code Generation

You can use custom functions in fuzzy inference systems for which you generate code. For more information on code generation for fuzzy systems, see “Deploy Fuzzy Inference Systems” on page 6-2.

If you use a nondouble data type for your generated code, you must propagate the data type from the input arguments of your custom function to the output argument. For example, the following custom aggregation function maintains the data type of `x` in `y` using the ones and zeros with the `'like'` argument.

```
function y = customagg(x)
maxVal = ones(1,size(x,2),'like',x);
y = zeros(1,size(x,2),'like',x);
```

```
for i = 1:size(x,1)
    y = min(maxVal,sum([y;x(i,:) ]));
end
```

For more information on writing functions that support C/C++ code generation, see “MATLAB Programming for Code Generation” (MATLAB Coder).

References

- [1] Mizumoto, M. "Pictorial Representations of Fuzzy Connectives, Part II: Cases of Compensatory Operators and Self-Dual Operators." *Fuzzy Sets and Systems*. Vol. 32, Number 1., 1989, pp. 45-79.

See Also

Fuzzy Logic Designer

Related Examples

- “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14
- “Build Fuzzy Systems at the Command Line” on page 2-38

Fuzzy Logic Image Processing

This example shows how to use Fuzzy Logic Toolbox™ software for image processing. Specifically, this example shows how to detect edges in an image.

An edge is a boundary between two uniform regions. You can detect an edge by comparing the intensity of neighboring pixels. However, because uniform regions are not crisply defined, small intensity differences between two neighboring pixels do not always represent an edge. Instead, the intensity difference might represent a shading effect.

The fuzzy logic approach for image processing allows you to use membership functions to define the degree to which a pixel belongs to an edge or a uniform region.

Import RGB Image and Convert to Grayscale

Import the image into MATLAB.

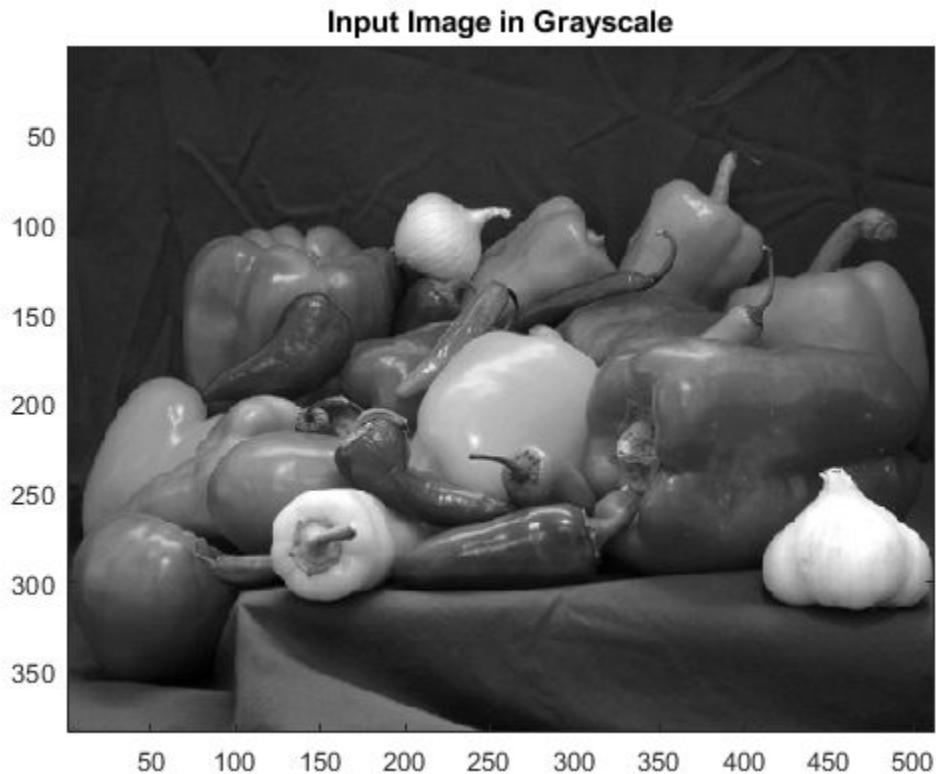
```
Irgb = imread('peppers.png');
```

Irgb is a 384 x 512 x 3 uint8 array. The three channels of Irgb (third array dimension) represent the red, green, and blue intensities of the image.

Convert Irgb to grayscale so that you can work with a 2-D array instead of a 3-D array. Use the standard NTSC conversion formula to calculate the effective luminance of each pixel.

```
Igray = 0.2989*Irgb(:,:,1)+0.5870*Irgb(:,:,2)+0.1140*Irgb(:,:,3);
```

```
figure  
image(Igray, 'CDataMapping', 'scaled');  
colormap('gray')  
title('Input Image in Grayscale')
```



Alternatively, you can use the `rgb2gray` function in the Image Processing Toolbox™ software to convert `Irgb` to grayscale.

Convert Image to Double-Precision Data

The Fuzzy Logic Toolbox software operates on double-precision numbers only. So, convert `Igray`, a `uint8` array, to a double array.

```
I = double(Igray);
```

Because `uint8` values are in the $[0 \ 2^8-1]$ range, all elements of `I` are in that range too. Scale `I` so that its elements are in the $[0 \ 1]$ range.

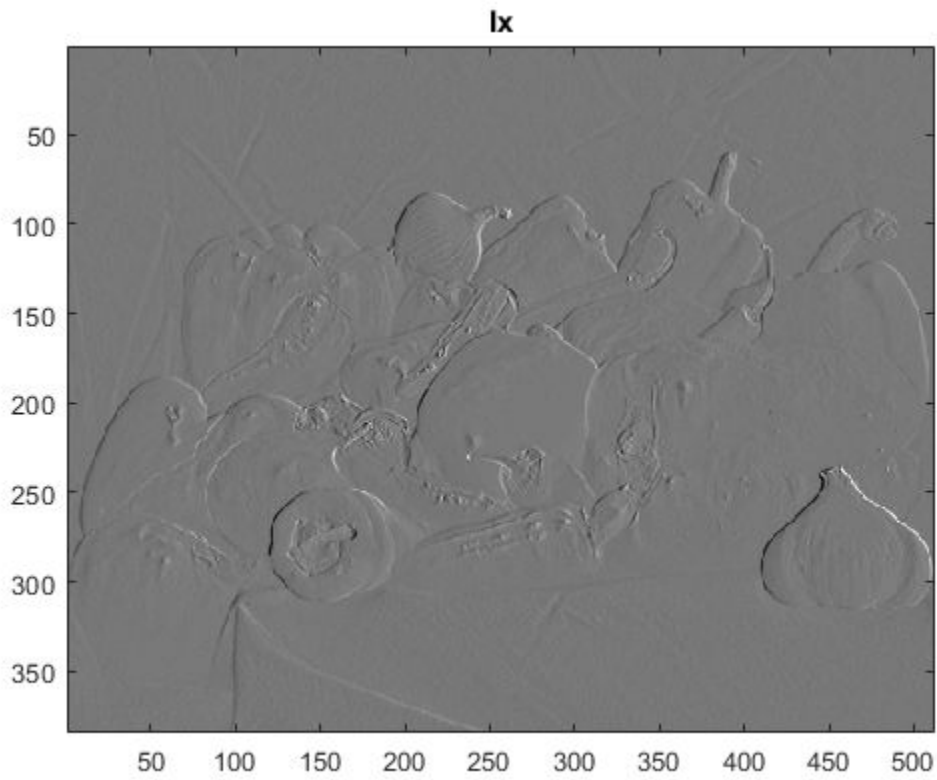
```
classType = class(Igray);  
scalingFactor = double(intmax(classType));  
I = I/scalingFactor;
```

Alternatively, you can use the `im2double` function in the Image Processing Toolbox software to convert `Igray` to a scaled, double-precision image.

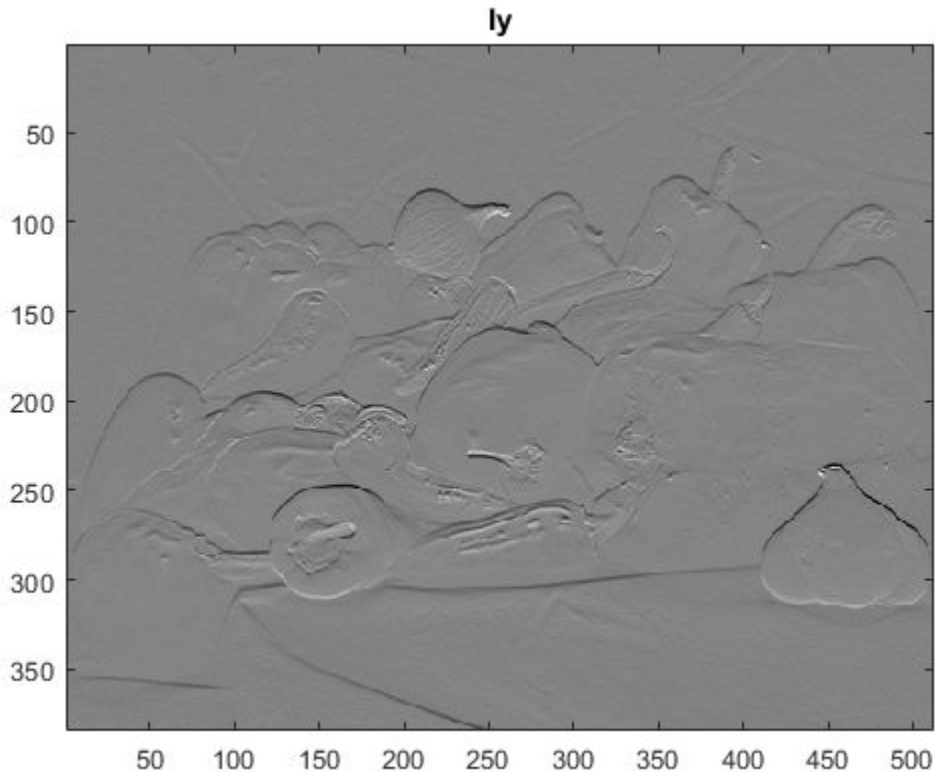
Obtain Image Gradient

The fuzzy logic edge-detection algorithm for this example relies on the image gradient to locate breaks in uniform regions. Calculate the image gradient along the x-axis and y-axis.

```
Gx = [-1 1];  
Gy = Gx';  
Ix = conv2(I,Gx,'same');  
Iy = conv2(I,Gy,'same');  
  
figure  
image(Ix,'CDataMapping','scaled')  
colormap('gray')  
title('Ix')
```



```
figure
image(Iy,'CDataMapping','scaled')
colormap('gray')
title('Iy')
```

G_x and G_y are simple gradient filters. You convolve I with G_x , using the `conv2` function, to obtain a matrix containing the x-axis gradients of I . The gradient values are in the $[-1 \ 1]$ range. Similarly, you convolve I with G_y to obtain the y-axis gradients of I . You can use other filters to obtain the image gradients, such as the Sobel operator or the Prewitt operator. For information about how you can filter an image using convolution, see “Convolution” (Image Processing Toolbox).

Alternatively, if you have the Image Processing Toolbox software, you can use the `imfilter`, `imgradientxy`, or `imgradient` functions to obtain the image gradients.

Define Fuzzy Inference System (FIS) for Edge Detection

Create a fuzzy inference system (FIS) for edge detection, `edgeFIS`.

```
edgeFIS = mamfis('Name', 'edgeDetection');
```

Specify the image gradients, I_x and I_y , as the inputs of `edgeFIS`.

```
edgeFIS = addInput(edgeFIS, [-1 1], 'Name', 'Ix');  
edgeFIS = addInput(edgeFIS, [-1 1], 'Name', 'Iy');
```

Specify a zero-mean Gaussian membership function for each input. If the gradient value for a pixel is 0, then it belongs to the zero membership function with a degree of 1.

```
sx = 0.1;  
sy = 0.1;  
edgeFIS = addMF(edgeFIS, 'Ix', 'gaussmf', [sx 0], 'Name', 'zero');  
edgeFIS = addMF(edgeFIS, 'Iy', 'gaussmf', [sy 0], 'Name', 'zero');
```

sx and sy specify the standard deviation for the zero membership function for the I_x and I_y inputs. You can change the values of sx and sy to adjust the edge detector performance. Increasing the values makes the algorithm less sensitive to the edges in the image and decreases the intensity of the detected edges.

Specify the intensity of the edge-detected image as an output of `edgeFIS`.

```
edgeFIS = addOutput(edgeFIS, [0 1], 'Name', 'Iout');
```

Specify the triangular membership functions, `white` and `black`, for `Iout`.

```
wa = 0.1;  
wb = 1;  
wc = 1;  
ba = 0;  
bb = 0;  
bc = 0.7;  
edgeFIS = addMF(edgeFIS, 'Iout', 'trimf', [wa wb wc], 'Name', 'white');  
edgeFIS = addMF(edgeFIS, 'Iout', 'trimf', [ba bb bc], 'Name', 'black');
```

As you can with sx and sy , you can change the values of wa , wb , wc , ba , bb , and bc to adjust the edge detector performance. The triplets specify the start, peak, and end of the triangles of the membership functions. These parameters influence the intensity of the detected edges.

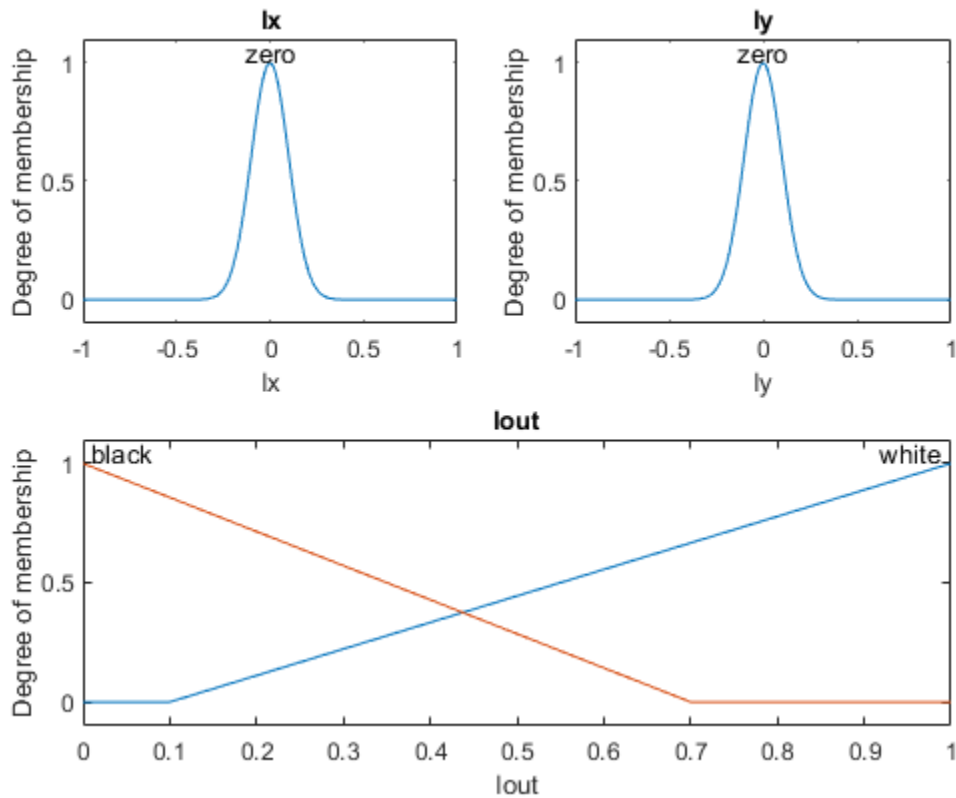
Plot the membership functions of the inputs/outputs of `edgeFIS`.

```
figure  
subplot(2,2,1)
```

```

plotmf(edgeFIS, 'input',1)
title('Ix')
subplot(2,2,2)
plotmf(edgeFIS, 'input',2)
title('Iy')
subplot(2,2,[3 4])
plotmf(edgeFIS, 'output',1)
title('Iout')

```



Specify FIS Rules

Add rules to make a pixel white if it belongs to a uniform region. Otherwise, make the pixel black.

```
r1 = "If Ix is zero and Iy is zero then Iout is white";  
r2 = "If Ix is not zero or Iy is not zero then Iout is black";  
edgeFIS = addRule(edgeFIS,[r1 r2]);  
edgeFIS.Rules
```

```
ans =  
  1x2 fisrule array with properties:
```

```
  Description  
  Antecedent  
  Consequent  
  Weight  
  Connection
```

```
Details:
```

```
      Description
```

```
  1  "Ix==zero & Iy==zero => Iout=white (1)"  
  2  "Ix~=zero | Iy~=zero => Iout=black (1)"
```

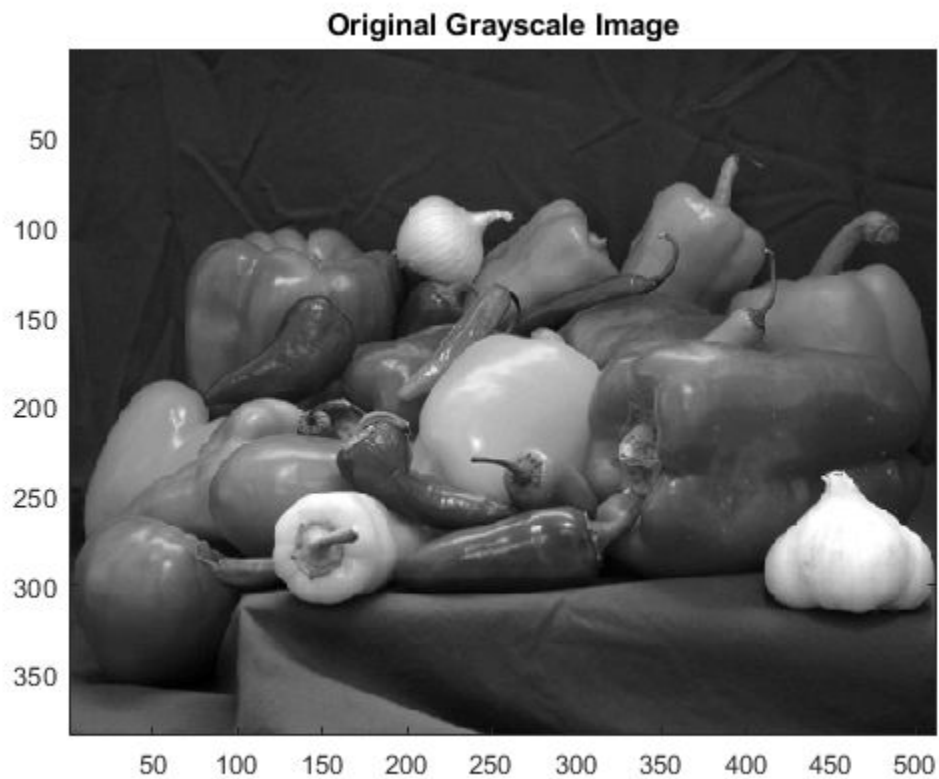
Evaluate FIS

Evaluate the output of the edge detector for each row of pixels in *I* using corresponding rows of *Ix* and *Iy* as inputs.

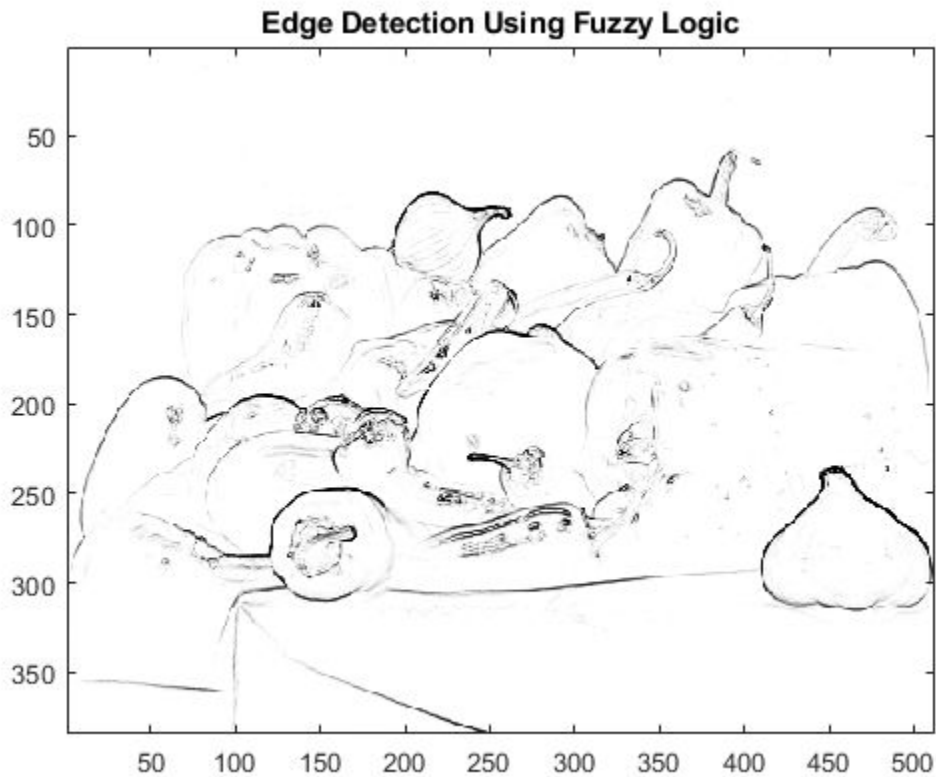
```
Ieval = zeros(size(I));  
for ii = 1:size(I,1)  
    Ieval(ii,:) = evalfis(edgeFIS,[(Ix(ii,:));(Iy(ii,:));]);  
end
```

Plot Results

```
figure  
image(I,'CDataMapping','scaled')  
colormap('gray')  
title('Original Grayscale Image')
```



```
figure
image(Ieval,'CDataMapping','scaled')
colormap('gray')
title('Edge Detection Using Fuzzy Logic')
```



Summary

You detected the edges in an image using a FIS, comparing the gradient of every pixel in the x and y directions. If the gradient for a pixel is not zero, then the pixel belongs to an edge (black). You defined the gradient as zero using Gaussian membership functions for your FIS inputs.

See Also

`evalfis`

More About

- “Build Fuzzy Systems at the Command Line” on page 2-38

Adaptive Neuro-Fuzzy Modeling

- “Neuro-Adaptive Learning and ANFIS” on page 3-2
- “Comparison of anfis and Neuro-Fuzzy Designer Functionality” on page 3-7
- “Train Adaptive Neuro-Fuzzy Inference Systems” on page 3-13
- “Test Data Against Trained System” on page 3-18
- “Save Training Error Data to MATLAB Workspace” on page 3-35
- “Predict Chaotic Time-Series” on page 3-43
- “Chaotic Time-Series Prediction” on page 3-51
- “Modeling Inverse Kinematics in a Robotic Arm” on page 3-61
- “Adaptive Noise Cancellation Using ANFIS” on page 3-72
- “Nonlinear System Identification” on page 3-82
- “Gas Mileage Prediction” on page 3-97

Neuro-Adaptive Learning and ANFIS

When to Use Neuro-Adaptive Learning

The basic structure of Mamdani fuzzy inference system is a model that maps input characteristics to input membership functions, input membership functions to rules, rules to a set of output characteristics, output characteristics to output membership functions, and the output membership functions to a single-valued output or a decision associated with the output. Such a system uses fixed membership functions that are chosen arbitrarily and a rule structure that is essentially predetermined by the user's interpretation of the characteristics of the variables in the model.

`anfis` and the **Neuro-Fuzzy Designer** apply fuzzy inference techniques to data modeling. As you have seen from the other fuzzy inference GUIs, the shape of the membership functions depends on parameters, and changing these parameters change the shape of the membership function. Instead of just looking at the data to choose the membership function parameters, you choose membership function parameters automatically using these Fuzzy Logic Toolbox applications.

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

In some modeling situations, you cannot discern what the membership functions should look like simply from looking at data. Rather than choosing the parameters associated with a given membership function arbitrarily, these parameters could be chosen so as to tailor the membership functions to the input/output data in order to account for these types of variations in the data values. In such cases, you can use the Fuzzy Logic Toolbox *neuro-adaptive* learning techniques incorporated in the `anfis` command.

Model Learning and Inference Through ANFIS

The neuro-adaptive learning method works similarly to that of neural networks. Neuro-adaptive learning techniques provide a method for the fuzzy modeling procedure to *learn* information about a data set. Fuzzy Logic Toolbox software computes the membership function parameters that best allow the associated fuzzy inference system to track the given input/output data. The Fuzzy Logic Toolbox function that accomplishes this membership function parameter adjustment is called `anfis`. The `anfis` function can be

accessed either from the command line or through the **Neuro-Fuzzy Designer**. Because the functionality of the command line function `anfis` and the **Neuro-Fuzzy Designer** is similar, they are used somewhat interchangeably in this discussion, except when specifically describing the **Neuro-Fuzzy Designer** app.

What Is ANFIS?

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

FIS Structure and Parameter Adjustment

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

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

Know Your Data

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

In general, this type of modeling works well if the training data presented to `anfis` for training (estimating) membership function parameters is fully representative of the features of the data that the trained FIS is intended to model. In some cases however,

data is collected using noisy measurements, and the training data cannot be representative of all the features of the data that will be presented to the model. In such situations, model validation is helpful.

Model Validation Using Testing and Checking Data Sets

Model validation is the process by which the input vectors from input/output data sets on which the FIS was not trained, are presented to the trained FIS model, to see how well the FIS model predicts the corresponding data set output values.

One problem with model validation for models constructed using adaptive techniques is selecting a data set that is both representative of the data the trained model is intended to emulate, yet sufficiently distinct from the training data set so as not to render the validation process trivial.

If you have collected a large amount of data, hopefully this data contains all the necessary representative features, so the process of selecting a data set for checking or testing purposes is made easier. However, if you expect to be presenting noisy measurements to your model, it is possible the training data set does not include all of the representative features you want to model.

The testing data set lets you check the generalization capability of the resulting fuzzy inference system. The idea behind using a checking data set for model validation is that after a certain point in the training, the model begins overfitting the training data set. In principle, the model error for the checking data set tends to decrease as the training takes place up to the point that overfitting begins, and then the model error for the checking data suddenly increases. Overfitting is accounted for by testing the FIS trained on the training data against the checking data, and choosing the membership function parameters to be those associated with the minimum checking error if these errors indicate model overfitting.

Usually, these training and checking data sets are collected based on observations of the target system and are then stored in separate files.

In the first example, two similar data sets are used for checking and training, but the checking data set is corrupted by a small amount of noise. This example illustrates the use of the **Neuro-Fuzzy Designer** with checking data to reduce the effect of model overfitting. In the second example, a training data set that is presented to `anfis` is sufficiently different than the applied checking data set. By examining the checking error sequence over the training period, it is clear that the checking data set is not good for model validation purposes. This example illustrates the use of the **Neuro-Fuzzy Designer** to compare data sets.

References

- [1] 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.
- [2] 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.
- [3] 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.
- [4] Jang, J.-S. R. and C.-T. Sun, "Neuro-fuzzy modeling and control," *Proceedings of the IEEE*, March 1995.
- [5] Jang, J.-S. R. and C.-T. Sun, *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*, Prentice Hall, 1997.
- [6] Wang, L.-X., *Adaptive fuzzy systems and control: design and stability analysis*, Prentice Hall, 1994.
- [7] Widrow, B. and D. Stearns, *Adaptive Signal Processing*, Prentice Hall, 1985.

See Also

Apps

Neuro-Fuzzy Designer

Functions

`anfis`

More About

- "Comparison of `anfis` and Neuro-Fuzzy Designer Functionality" on page 3-7
- "Train Adaptive Neuro-Fuzzy Inference Systems" on page 3-13

- “Test Data Against Trained System” on page 3-18
- “Save Training Error Data to MATLAB Workspace” on page 3-35
- “Predict Chaotic Time-Series” on page 3-43

Comparison of anfis and Neuro-Fuzzy Designer Functionality

This topic discusses the arguments and range components of the command-line function `anfis` and the analogous functionality of the **Neuro-Fuzzy Designer**.

You can configure the training options for the `anfis` command using an `anfisOptions` option set.

```
opt = anfisOptions;  
[fismat1, trnError, ss, fismat2, chkError] = anfis(trnData, opt);
```

where `trnData` is the training data. To use default training options, omit `opt`. You can modify the default option set using dot notation. For more information on the available options and their default values, see the `anfisOptions` reference page. All output arguments other than `fismat1`, the tuned fuzzy system, are optional.

When you open the **Neuro-Fuzzy Designer**, only the training data set must exist before implementing `anfis`. In addition, the step-size is fixed when the adaptive neuro-fuzzy system is trained using this app.

Training Data

The training data, `trnData`, is a required argument to `anfis`, and to the **Neuro-Fuzzy Designer**. Each row of `trnData` is a desired input/output pair of the target system you want to model. Each row starts with an input vector and is followed by an output value. Therefore, the number of rows of `trnData` is equal to the number of training data pairs. Since there is only one output, the number of columns of `trnData` is equal to the number of inputs plus one.

Input FIS Object

You can define the FIS object to tune using the `opt.InitialFIS` training option. You can create this object using:

- The **Fuzzy Logic Designer**
- The Membership Function Editor
- The Rule Editor from the **Neuro-Fuzzy Designer** (which allows an FIS object to be loaded from a file or the MATLAB workspace)

- The command-line function, `genfis` (for which you only specify numbers and types of membership functions)

The FIS object contains both the model structure (which specifies such items as the number of rules in the FIS and the number of membership functions for each input) and the parameters (which specify the shapes of membership functions).

There are two *methods* that `anfis` learning employs for updating membership function parameters, which you can select using the `opt.OptimizationMethod` training option:

- Backpropagation for all parameters (a steepest descent method)
- A hybrid method consisting of backpropagation for the parameters associated with the input membership functions, and least squares estimation for the parameters associated with the output membership functions

As a result, the training error decreases, at least locally, throughout the learning process. Therefore, the more the initial membership functions resemble the optimal ones, the easier it is for the model parameter training to converge. Human expertise about the target system to be modeled can aid in setting up these initial membership function parameters in the FIS object.

The `genfis` function, when used with grid partitioning, produces an FIS object based on a fixed number of membership functions. This object can cause an excessive number of rules when the number of inputs is moderately large; that is, more than four or five. Fuzzy Logic Toolbox software offers a method that provides for some dimension reduction in the fuzzy inference system: you can generate an FIS object using the clustering algorithm discussed in “Subtractive Clustering” on page 4-3. To use the clustering algorithm, you must select the **Sub. Clustering** option in the **Generate FIS** portion of the **Neuro-Fuzzy Designer** before the FIS is generated. This subtractive clustering method partitions the data into clusters, and generates an FIS with the minimum number of rules required to distinguish the fuzzy qualities associated with each clusters.

Training Options

The **Neuro-Fuzzy Designer** allows you to choose your desired error tolerance and number of training epochs.

For `anfis` command, you can specify training the training termination condition and gradient descent step size. To specify the following options, first create a default `anfisOptions` option set, `opt`. You can then modify the options using dot notation.

- `opt.EpochNumber` — Number of training epochs (default = 10)
- `opt.ErrorGoal` — Training error goal (default = 0)
- `opt.InitialStepSize` — Initial step-size (default = 0.01)
- `opt.StepSizeDecreaseRate` — Step-size decrease rate (default = 0.9)
- `opt.StepSizeIncreaseRate` — Step-size increase rate (default = 1.1)

If you do not modify an option in `opt`, the default value is used. The training process stops if the designated epoch number is reached or the error goal is achieved, whichever comes first.

Usually, the step-size profile is a curve that increases initially, reaches some maximum, and then decreases for the remainder of the training. You achieve this ideal step-size profile by adjusting the initial step-size and the increase and decrease rates (`opt.InitialStepSize`, `opt.StepSizeDecreaseRate`, `opt.StepSizeIncreaseRate`). The default values are configured to cover a wide range of learning tasks. For any specific application, you can modify these step-size options to optimize the training. However, there are no user-specified step-size options for training the adaptive neuro-fuzzy inference system generated using the **Neuro-Fuzzy Designer**.

Display Options

Display options apply only to the command-line function `anfis`. You can specify what training progress information to display in the MATLAB Command Window. As with the training options, you specify the display options using the `anfisOptions` option set, `opt`. For each display option, if you specify a value of 1 (the default value), the corresponding data is displayed. Specifying a value of 0 suppresses the display:

- `opt.DisplayANFISInformation` — Display ANFIS information at the start of training
- `opt.DisplayErrorValues` — Display the training error at each epoch
- `opt.DisplayStepSize` — Display the step-size each time it changes.
- `opt.DisplayFinalResults` — Display the final training error and validation error

Method

Both the **Neuro-Fuzzy Designer** and the command-line `anfis` apply either a backpropagation form of the steepest descent method for membership function parameter estimation, or a hybrid combination of backpropagation and the least-squares

methods. The choices for this argument are `hybrid` or `backpropagation`. To specify the training method for the `anfis` function, use the `opt.OptimizationMethod` training option as either 1 (hybrid) or 0 (backpropagation).

Output FIS Object for Training Data

`fismat1` is the output FIS object corresponding to the minimum training error. This FIS object is the one that you use to represent the fuzzy system when there is no checking data used for model cross-validation. `fismat1` corresponds to the FIS object that the **Neuro-Fuzzy Designer** saves when the checking data option is not used. For more information on cross-validation using checking data, see “Checking Data” on page 3-11.

Training Error

The training error is the difference between the training data output value, and the output of the fuzzy inference system corresponding to the same training data input value (the one associated with that training data output value). The training error `trnError` records the root mean squared error (RMSE) of the training data set at each epoch. `fismat1` is the snapshot of the FIS object when the training error measure is at its minimum.

The **Neuro-Fuzzy Designer** plots the training error versus epochs curve as the system is trained.

Step-Size

You cannot control the step-size options with the **Neuro-Fuzzy Designer**. Using the command-line `anfis`, the step-size array `ss` records the step-size during the training. Plotting `ss` gives the step-size profile, which serves as a reference for adjusting the initial step-size (`opt.InitialStepSize`) and the corresponding decrease and increase rates. The step-size for the command-line function `anfis` is updated according to the following guidelines:

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

The default value for the initial step-size is 0.01; the default values for `opt.StepSizeIncreaseRate` and `opt.StepSizeDecreaserate` are 1.1 and 0.9, respectively.

Checking Data

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

The checking data is important for learning tasks for which the input number is large or the data itself is noisy. A fuzzy inference system should track a given input/output data set well. Because the model structure used for `anfis` is fixed with a large number of parameters, there is a tendency for the model to overfit the data on which it is trained, especially for many training epochs. If overfitting does occur, the fuzzy inference system may not respond well to other independent data sets, especially if they are noisy. A validation or checking data set can be useful for these situations. This data set is used to cross-validate the fuzzy inference model. This cross-validation requires applying the checking data to the model and then seeing how well the model responds to this data.

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

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

- Is similar enough to the training data that the checking data error decreases as the training begins.
- Increases at some point in the training after the data overfitting occurs.

For information on using checking data, see “Checking Data Does Not Validate Model” on page 3-29.

Output FIS Object for Checking Data

The output of the command-line `anfis`, `fismat2`, is the output FIS object with the minimum checking error. This FIS object is the one that you should use for further calculation if checking data is used for cross-validation. `fismat2` is only returned if you specify validation data using `opt.ValidationData`.

`fismat2` corresponds to the FIS object that the **Neuro-Fuzzy Designer** saves when the checking data option is used.

Checking Error

The checking error is the difference between the checking data output value, and the output of the fuzzy inference system corresponding to the same checking data input value, which is the one associated with that checking data output value. The checking error `chkError` records the RMSE for the checking data at each epoch. `fismat2` is the snapshot of the FIS object when the checking error is at its minimum. `chkError` is only returned if you specify validation data using `opt.ValidationData`.

The **Neuro-Fuzzy Designer** plots the checking error versus epochs curve as the system is trained.

See Also

Apps

Neuro-Fuzzy Designer

Functions

`anfis`

More About

- “Neuro-Adaptive Learning and ANFIS” on page 3-2
- “Train Adaptive Neuro-Fuzzy Inference Systems” on page 3-13
- “Test Data Against Trained System” on page 3-18
- “Save Training Error Data to MATLAB Workspace” on page 3-35
- “Predict Chaotic Time-Series” on page 3-43

Train Adaptive Neuro-Fuzzy Inference Systems

This example shows how to create, train, and test Sugeno-type fuzzy systems using the **Neuro-Fuzzy Designer**.

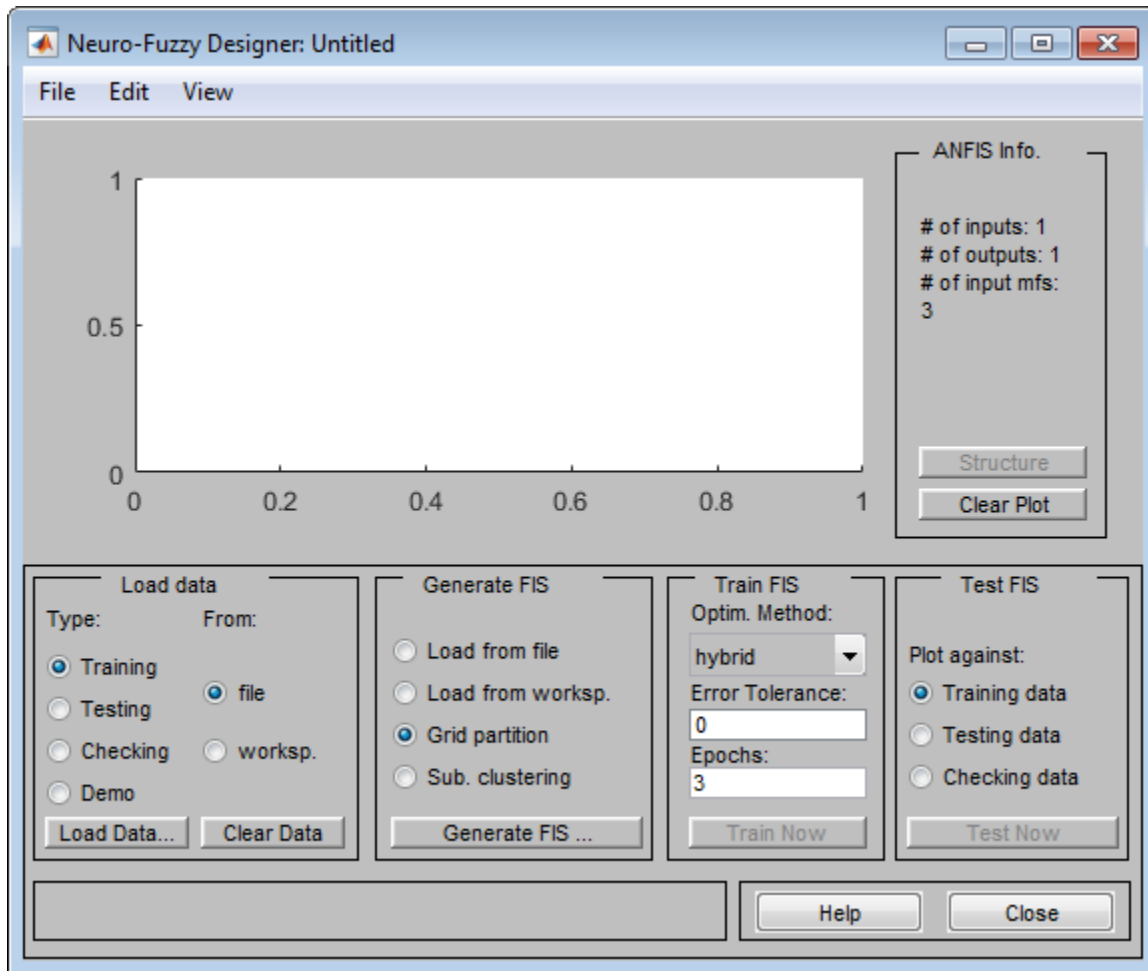
To start the app, type the following command at the MATLAB prompt:

```
neuroFuzzyDesigner
```

The **Neuro-Fuzzy Designer** includes four distinct areas to support a typical workflow. The app lets you perform the following tasks:

- 1 “Loading, Plotting, and Clearing the Data” on page 3-14
- 2 “Generating or Loading the Initial FIS Structure” on page 3-15
- 3 “Training the FIS” on page 3-16
- 4 “Validating the Trained FIS” on page 3-16

Access the online help topics by clicking **Help** in the **Neuro-Fuzzy Designer**.



Loading, Plotting, and Clearing the Data

To train an FIS, you must begin by loading a **Training** data set that contains the desired input/output data of the system to be modeled. Any data set you load must be an array with the data arranged as column vectors, and the output data in the last column.

You can also load **Testing** and **Checking** data in the designer. For more information on testing and checking data sets, see “Model Validation Using Testing and Checking Data Sets” on page 3-4.

To load a data set using the **Load data** portion of the designer:

- 1 Specify the data **Type**.
- 2 Select the data from a **file** or the MATLAB **worksp**.
- 3 Click **Load Data**.

After you load the data, it displays in the plot. The training, testing and checking data are annotated in blue as *circles*, *diamonds*, and *pluses* respectively.

To clear a specific data set from the designer:

- 1 In the **Load data** area, select the data **Type**.
- 2 Click **Clear Data**.

This action also removes the corresponding data from the plot.

Generating or Loading the Initial FIS Structure

Before you start the FIS training, you must specify an initial FIS model structure. To specify the model structure, perform one of the following tasks:

- Load a previously saved Sugeno-type FIS structure from a file or the MATLAB workspace.
- Generate the initial FIS model by choosing one of the following partitioning techniques:
 - **Grid partition**— Generates a single-output Sugeno-type FIS by using grid partitioning on the data.
 - **Sub. clustering** — Generates an initial model for ANFIS training by first applying subtractive clustering on the data.

To view a graphical representation of the initial FIS model structure, click **Structure**.

Training the FIS

After loading the training data and generating the initial FIS structure, you can start training the FIS.

Tip If you want to save the training error generated during ANFIS training to the MATLAB workspace, see “Save Training Error Data to MATLAB Workspace” on page 3-35.

The following steps show you how to train the FIS.

- 1 In **Optim. Method**, choose **hybrid** or **backpropaga** as the optimization method.

The optimization methods train the membership function parameters to emulate the training data.

Note The **hybrid** optimization method is a combination of least-squares and backpropagation gradient descent method.

- 2 Enter the number of training **Epochs** and the training **Error Tolerance** to set the stopping criteria for training.

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

- 3 Click **Train Now** to train the FIS.

This action adjusts the membership function parameters and displays the error plots.

Examine the error plots to determine overfitting during the training. If you notice the checking error increasing over iterations, it indicates model overfitting. For examples on model overfitting, see “Checking Data Helps Model Validation” on page 3-18 and “Checking Data Does Not Validate Model” on page 3-29.

Validating the Trained FIS

After the FIS is trained, validate the model using a **Testing** or **Checking** data that differs from the one you used to train the FIS. To validate the trained FIS:

- 1 Select the validation data set and click **Load Data**.

2 Click Test Now.

This action plots the test data against the FIS output (shown in red) in the plot.

For more information on the use of testing data and checking data for model validation, see “Model Validation Using Testing and Checking Data Sets” on page 3-4.

See Also

Neuro-Fuzzy Designer

More About

- “Neuro-Adaptive Learning and ANFIS” on page 3-2
- “Comparison of anfis and Neuro-Fuzzy Designer Functionality” on page 3-7
- “Test Data Against Trained System” on page 3-18
- “Save Training Error Data to MATLAB Workspace” on page 3-35

Test Data Against Trained System

Checking Data Helps Model Validation

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

- 1 “Loading Data” on page 3-18
- 2 “Initializing and Generating Your FIS” on page 3-22
- 3 “Viewing Your FIS Structure” on page 3-24
- 4 “ANFIS Training” on page 3-26
- 5 “Testing Your Data Against the Trained FIS” on page 3-28

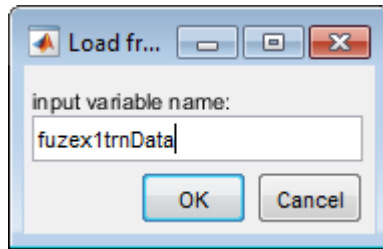
Loading Data

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

To load the data sets from the workspace into the **Neuro-Fuzzy Designer**:

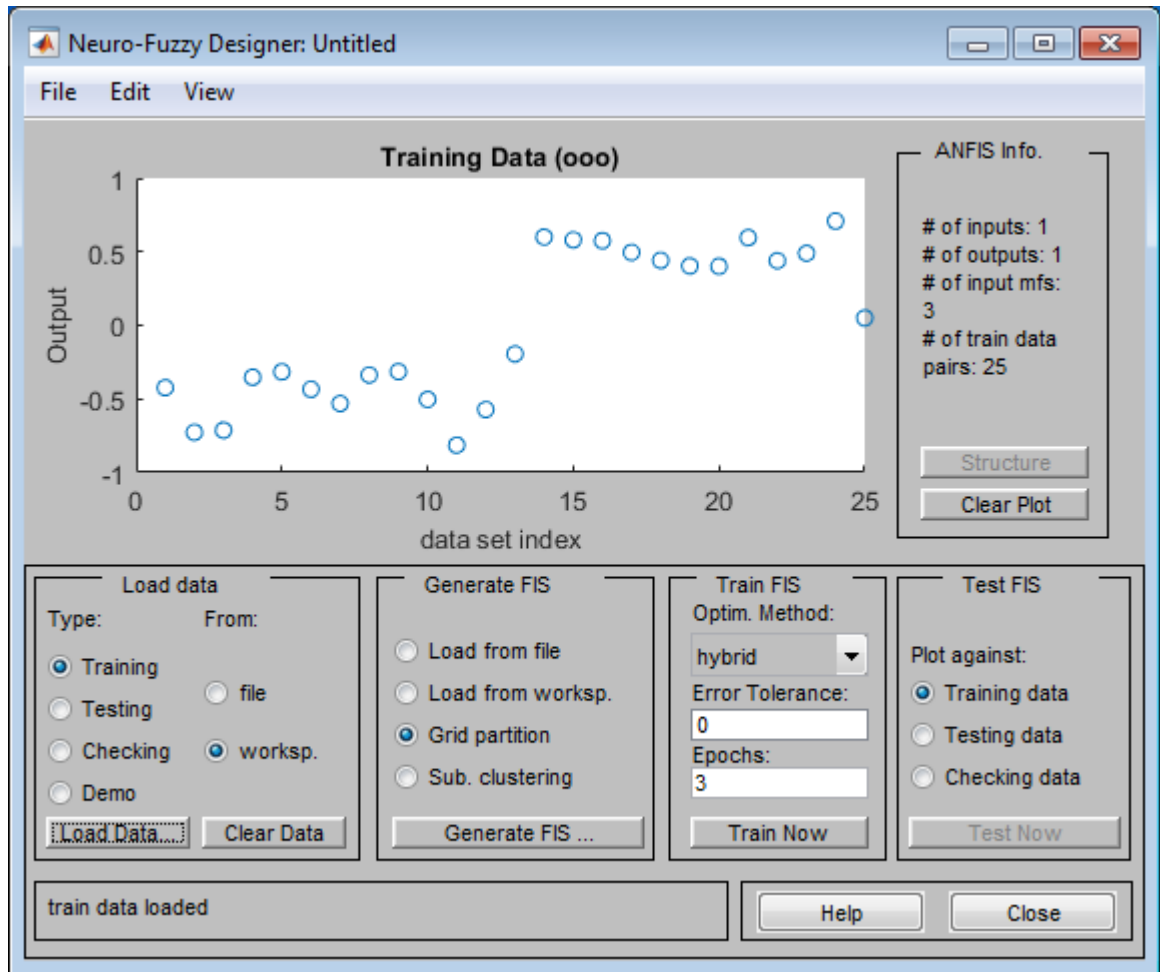
- 1 Type the following commands at the MATLAB command line to load the data sets from the folder `fuzzydemos` into the MATLAB workspace:

```
load fuzex1trnData.dat
load fuzex2trnData.dat
load fuzex1chkData.dat
load fuzex2chkData.dat
```
- 2 Open the **Neuro-Fuzzy Designer** by typing `neuroFuzzyDesigner` in the MATLAB command line.
- 3 To load the training data set from the workspace:
 - a In the **Load data** portion of the designer, select the following options:
 - **Type: Training**
 - **From: worksp**
 - b Click **Load Data** to open the Load from workspace dialog box.



- c Type `fuzex1trnData` as shown in the following figure, and click **OK**.

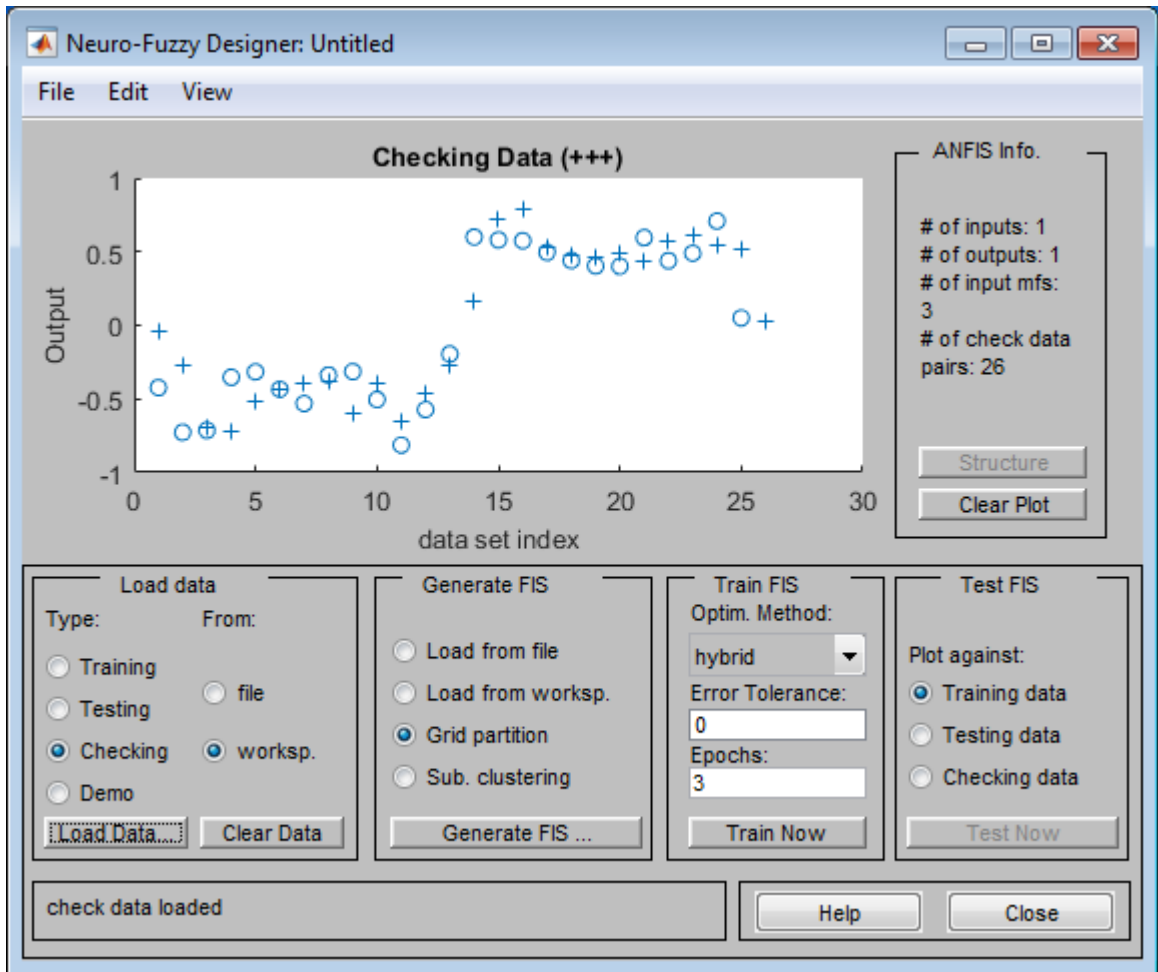
The training data set is used to train a fuzzy system by adjusting the membership function parameters that best model this data, and appears in the plot in the center of the app as a set of *circles*.



The horizontal axis is marked **data set index**. This index indicates the row from which that input data value was obtained (whether or not the input is a vector or a scalar).

- 4 To load the checking data set from the workspace:
 - a In the **Load data** section, select **Checking** in the **Type** column.
 - b Click **Load Data** to open the Load from workspace dialog box.
 - c Type `fuzex1chkData` as the variable name and click **OK**.

The checking data appears in the plot as *pluses* superimposed on the training data.



The next step is to specify an initial fuzzy inference system for anfis to train.

Initializing and Generating Your FIS

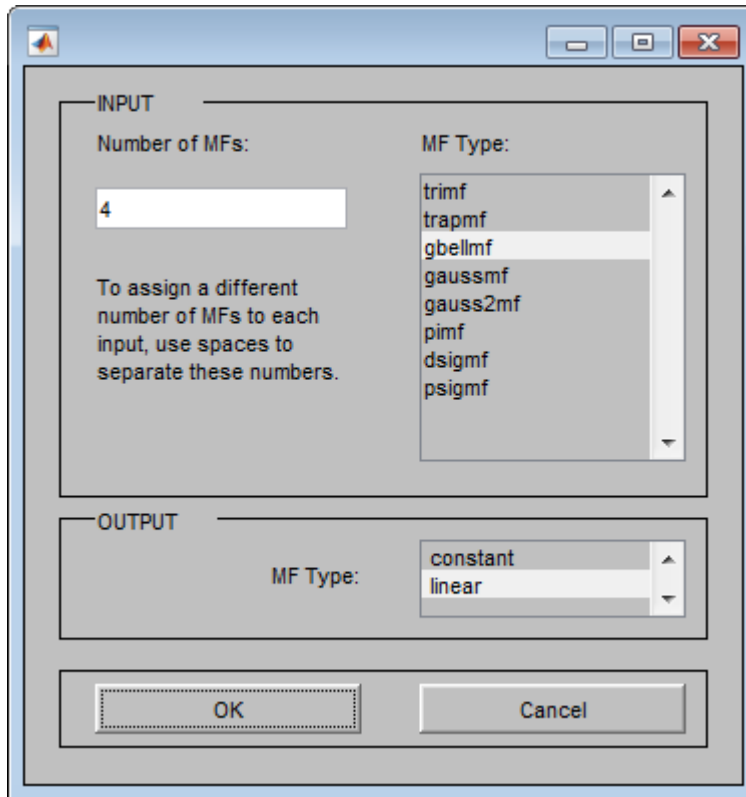
You can either initialize the FIS parameters to your own preference, or if you do not have any preference for how you want the initial membership functions to be parameterized, you can let `anfis` initialize the parameters for you, as described in the following sections:

- “Automatic FIS Structure Generation” on page 3-22
- “Specifying Your Own Membership Functions for ANFIS” on page 3-23

Automatic FIS Structure Generation

To initialize your FIS using `anfis`:

- 1 Choose **Grid partition**, the default partitioning method. The two partition methods, grid partitioning and subtractive clustering, are described in `genfis`.
- 2 Click **Generate FIS**. Clicking this button displays a menu from which you can choose the number of membership functions, **MFs**, and the type of input and output membership functions. There are only two choices for the output membership function: `constant` and `linear`. This limitation of output membership function choices is because `anfis` only operates on Sugeno-type systems.
- 3 Fill in the entries as shown in the following figure, and click **OK**.



You can also implement this FIS generation from the command line using `genfis`.

Specifying Your Own Membership Functions for ANFIS

You can choose your own preferred membership functions with specific parameters to be used by `anfis` as an initial FIS for training.

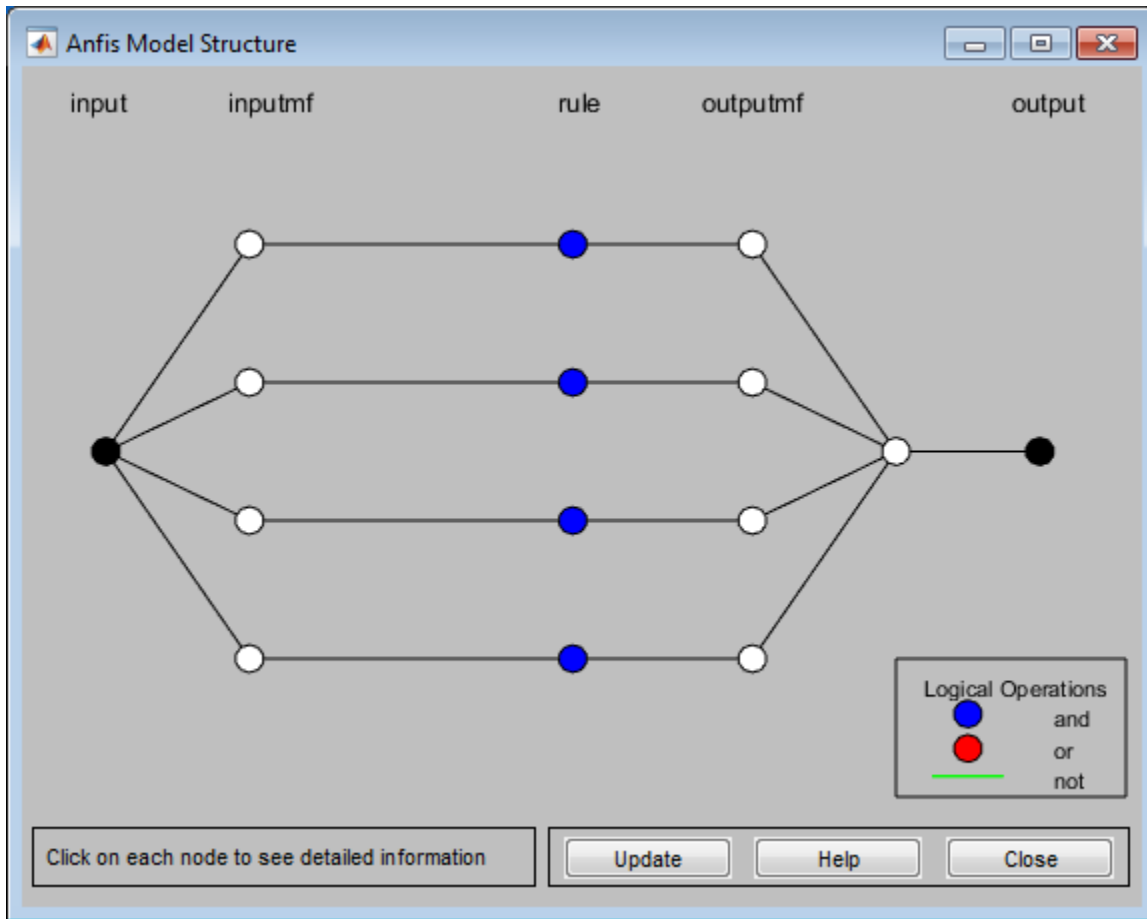
To define your own FIS structure and parameters:

- 1 Open the **Membership functions** menu item from the **Edit** menu.
- 2 Add your desired membership functions (the custom membership option will be disabled for `anfis`). The output membership functions must either be all constant or all linear. For carrying out this and the following step, see “The Fuzzy Logic Designer” on page 2-17 and “The Membership Function Editor” on page 2-22.

- 3 Select the **Rules** menu item in the **Edit** menu, and use the Rule Editor to generate the rules (see “The Rule Editor” on page 2-30).
- 4 Select the **FIS Properties** menu item from the **Edit** menu. Name your FIS, and save it to either the workspace or to file.
- 5 Click the **Close** button to return to the **Neuro-Fuzzy Designer** to train the FIS.
- 6 To load an existing FIS for ANFIS initialization, in the **Generate FIS** portion of the designer, click **Load from worksp** or **Load from file**. You load your FIS from a file if you have saved an FIS previously that you would like to use. Otherwise you load your FIS from the workspace.

Viewing Your FIS Structure

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



The branches in this graph are color coded. Color coding of branches characterize the rules and indicate whether or not *and*, *not*, or *or* are used in the rules. The input is represented by the left-most node and the output by the right-most node. The node represents a normalization factor for the rules. Clicking on the nodes indicates information about the structure.

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

ANFIS Training

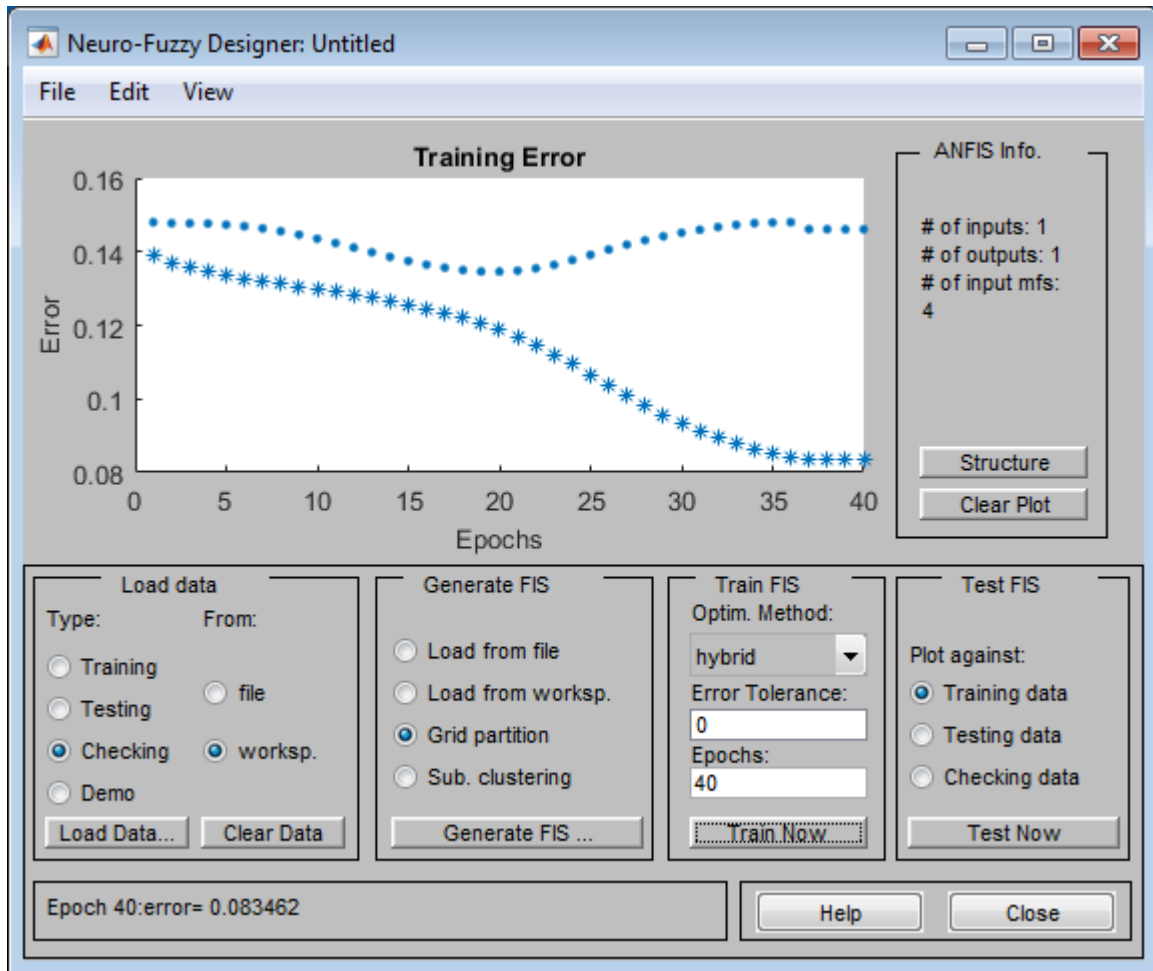
The two `anfis` parameter optimization method options available for FIS training are `hybrid` (the default, mixed least squares and backpropagation) and `backpropa` (backpropagation). **Error Tolerance** is used to create a training stopping criterion, which is related to the error size. The training will stop after the training data error remains within this tolerance. This is best left set to 0 if you are unsure how your training error may behave.

Note If you want to save the training error data generated during ANFIS training to the MATLAB workspace, you must train the FIS at the command line. For an example, “Save Training Error Data to MATLAB Workspace” on page 3-35.

To start the training:

- 1** Leave the optimization method at `hybrid`.
- 2** Set the number of training **Epochs** to 40 (the default value is 3).
- 3** Select **Train Now**.

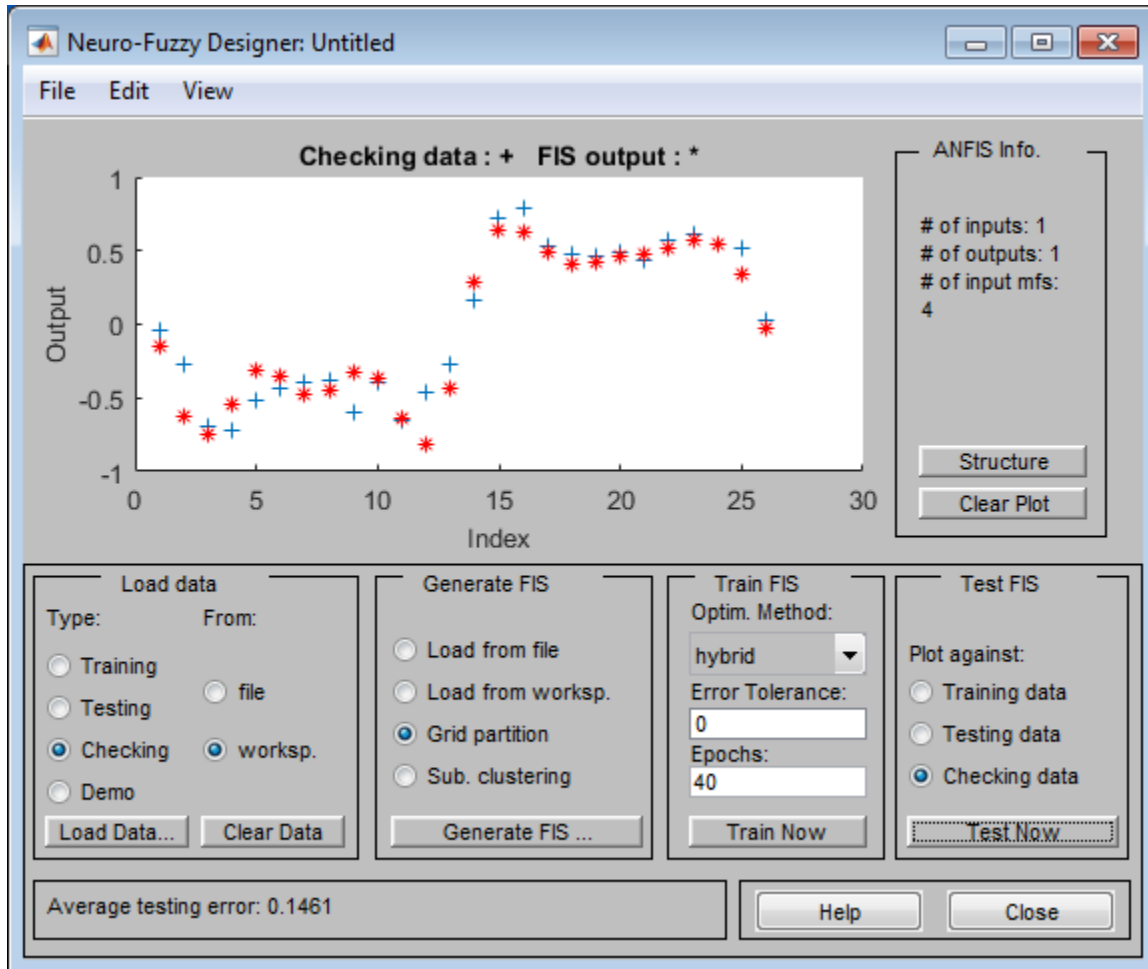
The following window appears on your screen.



The plot shows the checking error as ♦ ♦ on the top . The training error appears as * * on the bottom. The checking error decreases up to a certain point in the training, and then it increases. This increase represents the point of model overfitting. anfis chooses the model parameters associated with the minimum checking error (just prior to this jump point). This example shows why the checking data option of anfis is useful.

Testing Your Data Against the Trained FIS

To test your FIS against the checking data, select **Checking data** in the **Test FIS** portion of the **Neuro-Fuzzy Designer**, and click **Test Now**. When you test the checking data against the FIS, it looks satisfactory.



Loading More Data with `anfis`

If you load data into `anfis` after clearing previously loaded data, you must make sure that the newly loaded data sets have the same number of inputs as the previously loaded ones did. Otherwise, you must start a new **Neuro-Fuzzy Designer** session from the command line.

Checking Data Option and Clearing Data

If you do not want to use the checking data option of `anfis`, then do not load any checking data before you train the FIS. If you decide to retrain your FIS with no checking data, you can unload the checking data in one of two ways:

- Select the **Checking** option button in the **Load data** portion of the **Neuro-Fuzzy Designer**, and then click **Clear Data** to unload the checking data.
- Close the **Neuro-Fuzzy Designer**, and go to the MATLAB command line, and retype `neuroFuzzyDesigner`. In this case you must reload the training data.

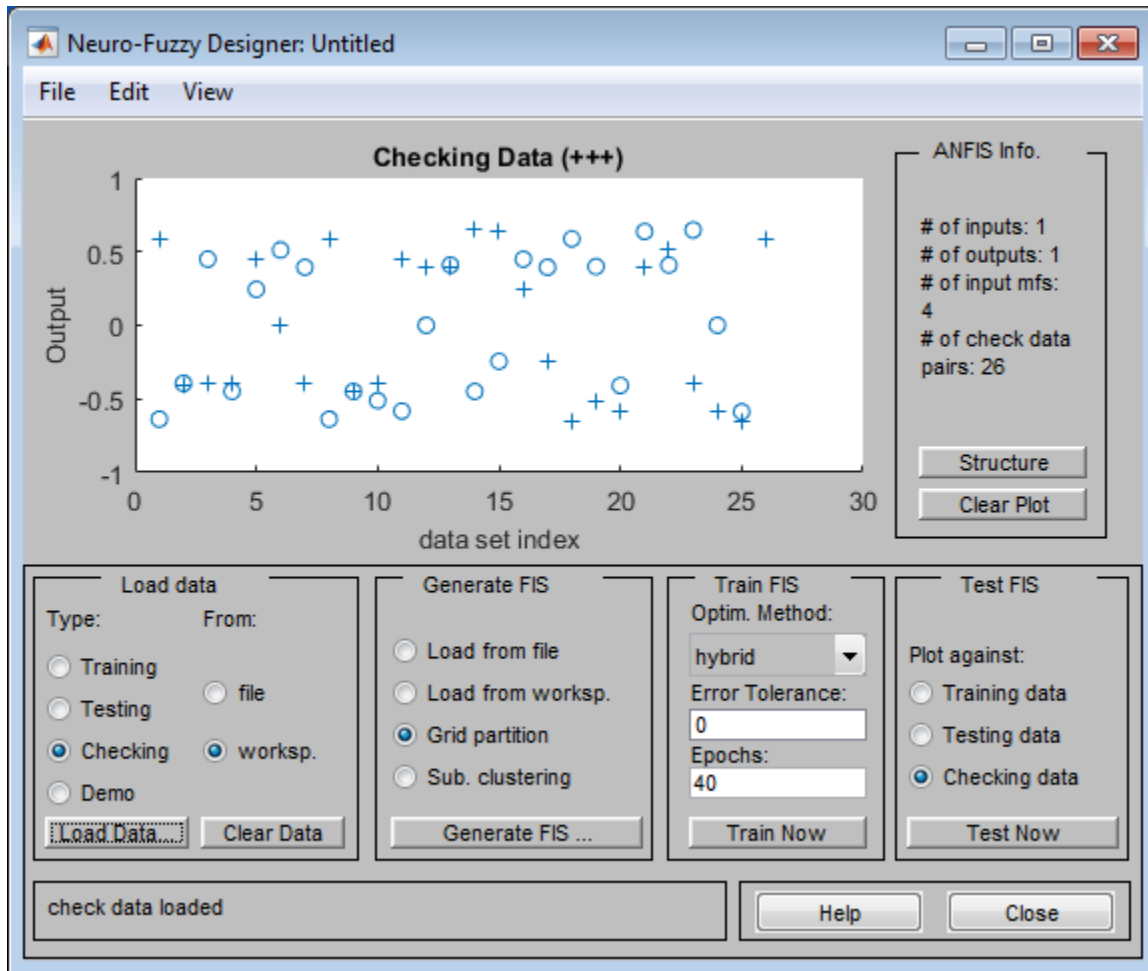
After clearing the data, you must regenerate your FIS. After the FIS is generated, you can use your first training experience to decide on the number of training epochs you want for the second round of training.

Checking Data Does Not Validate Model

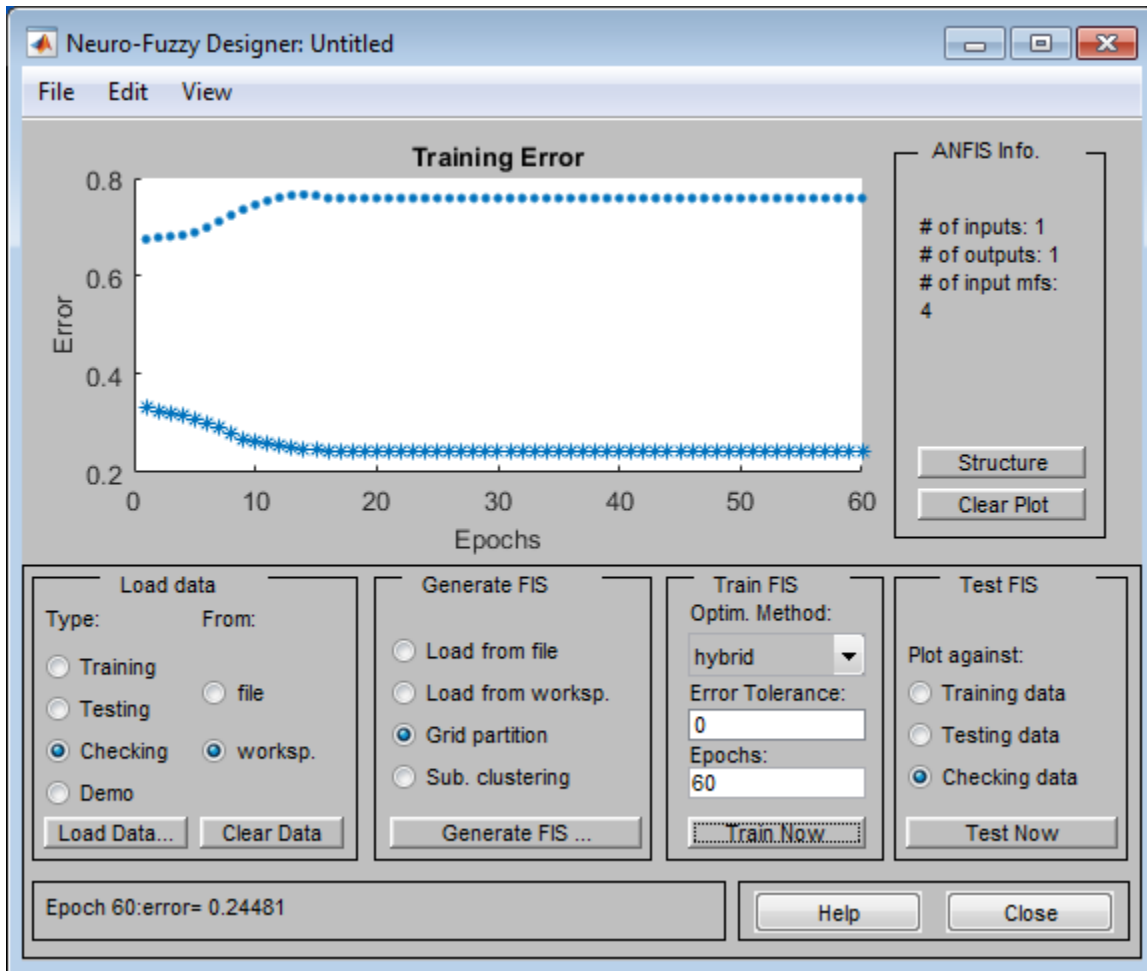
This example examines what happens when the training and checking data sets are sufficiently different. To see how the **Neuro-Fuzzy Designer** can be used to learn something about data sets and how they differ:

- 1 Clear the **Neuro-Fuzzy Designer**:
 - Clear both the training and checking data.
 - (optional) Click the **Clear Plot** button on the right.
- 2 Load `fuzex2trnData` and `fuzex2chkData` (respectively, the training data and checking data) from the MATLAB workspace just as you did in the previous example.

You should see a plot similar to the one in the following figure. The training data appears as *circles* superimposed with the checking data, appearing as *pluses*.



Train the FIS for this system exactly as you did in the previous example, except now choose **60 Epochs** before training. You should get the following plot, showing the checking error as ♦ ♦ on top and the training error as * * on the bottom.



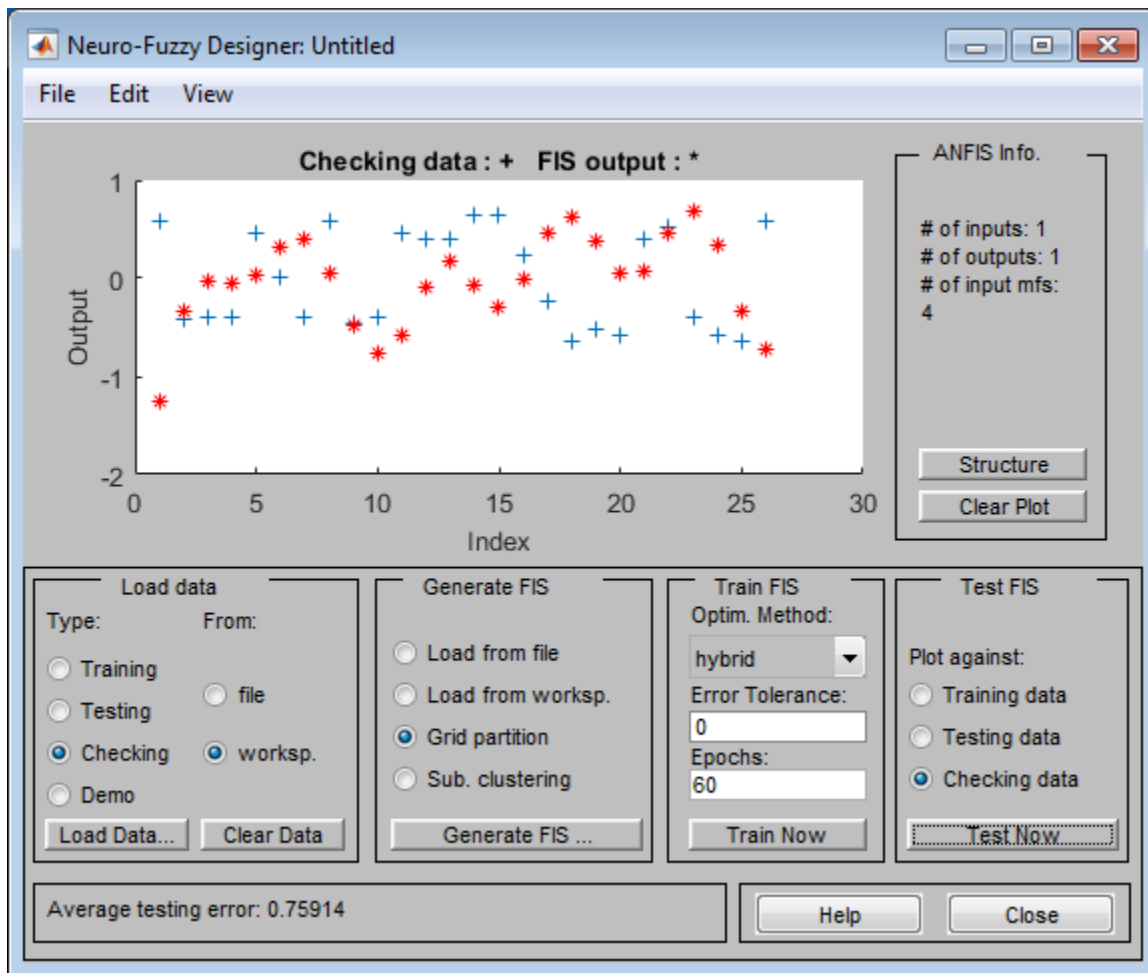
In this case, the checking error is quite large. It appears that the minimum checking error occurs within the first epoch. Using the checking data option with `anfis` automatically sets the FIS parameters to be those associated with the minimum checking error. Clearly this set of membership functions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

This example illustrates the problem discussed earlier wherein the checking data set presented to `anfis` for training was sufficiently different from the training data set. As a result, the trained FIS did not capture the features of this data set very well. It is important to know the features of your data set well when you select your training and

checking data. When you do not know the features of your data, you can analyze the checking error plots to see whether or not the checking data performed sufficiently well with the trained model.

In this example, the checking error is sufficiently large to indicate that either you need to select more data for training or modify your membership function choices (both the number of membership functions and the type). Otherwise, the system can be retrained without the checking data, if you think the training data sufficiently captures the features you are trying to represent.

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



See Also

Neuro-Fuzzy Designer

More About

- “Neuro-Adaptive Learning and ANFIS” on page 3-2

- “Comparison of anfis and Neuro-Fuzzy Designer Functionality” on page 3-7
- “Train Adaptive Neuro-Fuzzy Inference Systems” on page 3-13
- “Save Training Error Data to MATLAB Workspace” on page 3-35

Save Training Error Data to MATLAB Workspace

When using **Neuro-Fuzzy Designer**, you can export your initial FIS structure to the MATLAB workspace and then save the ANFIS training error values in the workspace.

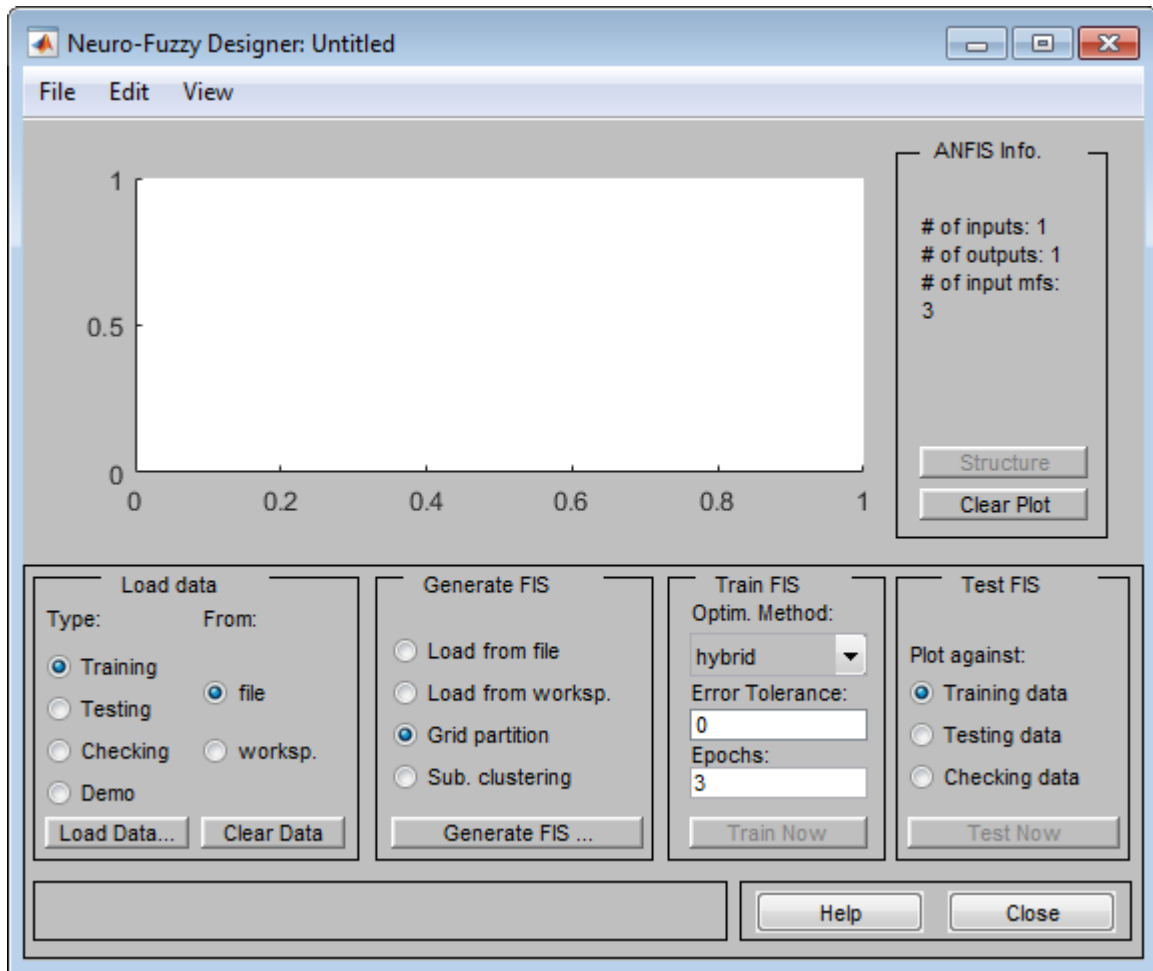
The following example shows how to save the training error generated during ANFIS training to the MATLAB workspace:

- 1 Load the training and checking data in the MATLAB workspace by typing the following commands at the MATLAB prompt:

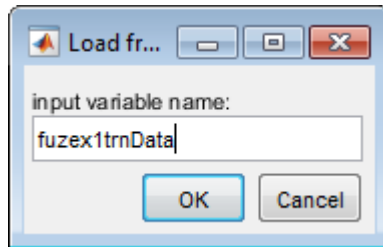
```
load fuzex1trnData.dat  
load fuzex1chkData.dat
```

- 2 Open the **Neuro-Fuzzy Designer** by typing the following command:

```
neuroFuzzyDesigner
```

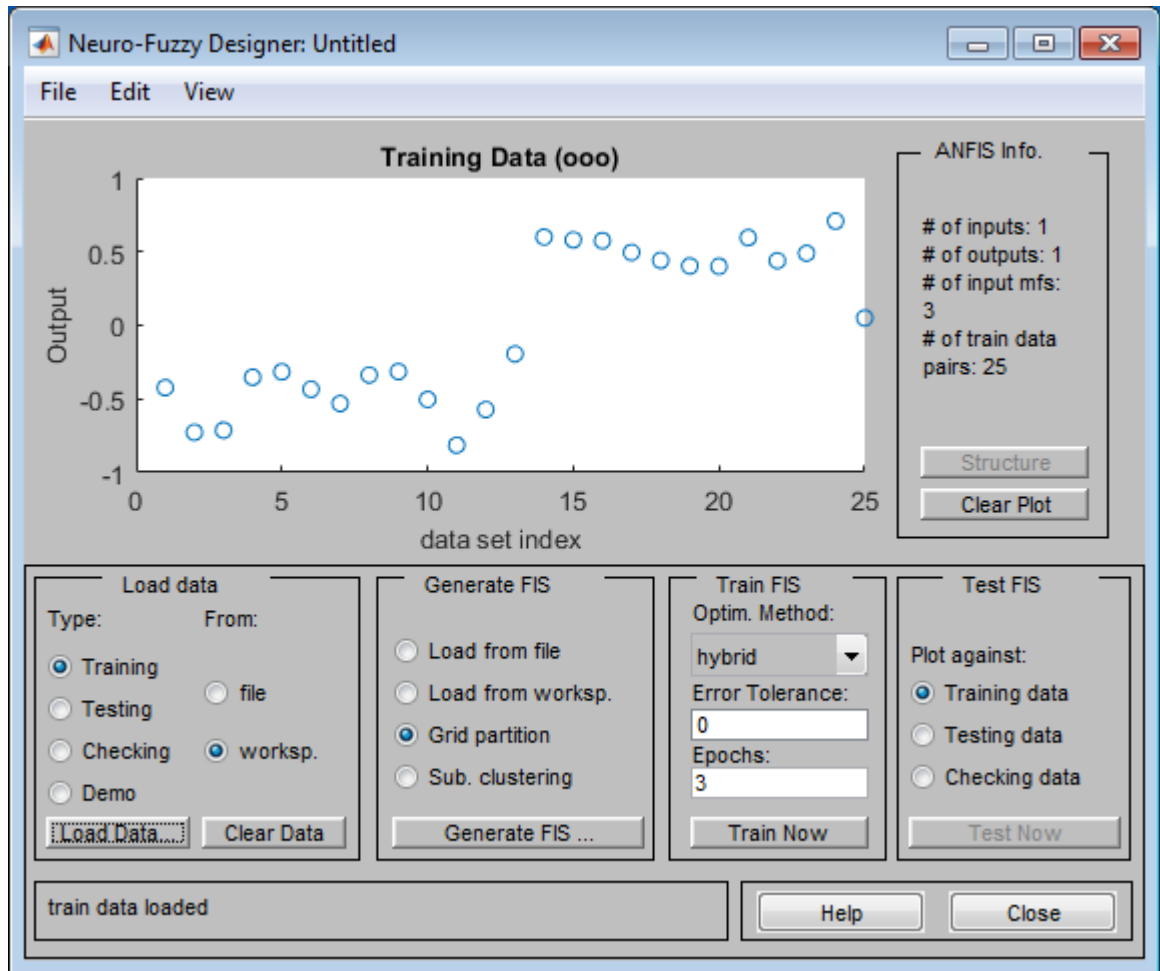


- 3 Load the training data from the MATLAB workspace into the **Neuro-Fuzzy Designer**:
 - a In the **Load data** panel of the **Neuro-Fuzzy Designer**, verify that **Training** is selected in the **Type** column.
 - b Select **worksp** in the **From** column.
 - c Click **Load Data** to open the Load from workspace dialog box.



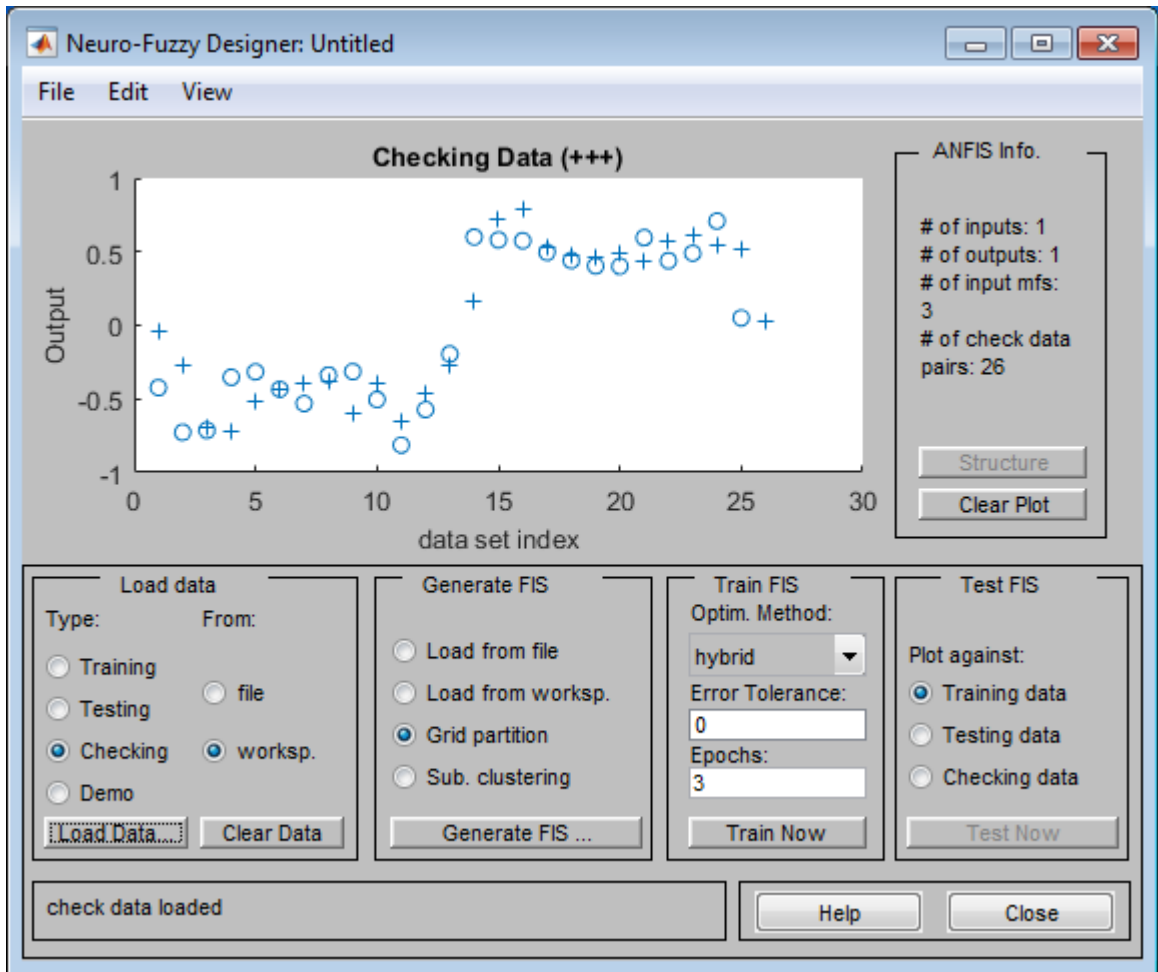
- d** Type `fuzex1trnData`, and click **OK**.

The **Neuro-Fuzzy Designer** displays the training data in the plot as a set of circles (○).



- 4 Load the checking data from the MATLAB workspace into the **Neuro-Fuzzy Designer**:
 - a In the **Load data** panel of the **Neuro-Fuzzy Designer**, select **Checking** in the **Type** column.
 - b Click **Load Data** to open the Load from workspace dialog box.
 - c Type `fuzex1chkData` as the variable name, and click **OK**.

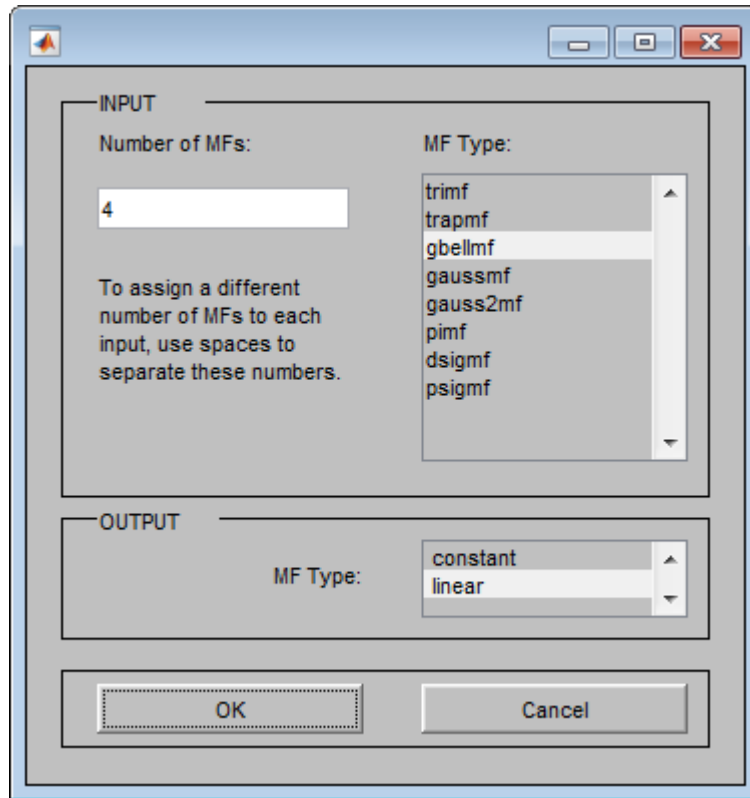
The **Neuro-Fuzzy Designer** displays the checking data as plus signs (+) superimposed on the training data.



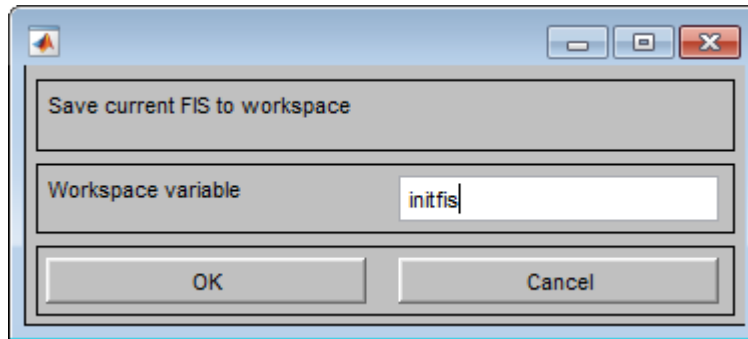
- 5 Generate an initial FIS:
 - a In the **Generate FIS** panel, verify that **Grid partition** option is selected.
 - b Click **Generate FIS**.

This action opens a dialog box where you specify the structure of the FIS.

- c In the dialog box, specify the following:
 - Enter 4 in the **Number of MFs** field.
 - Select **gbellmf** as the **Membership Type** for the input.
 - Select **linear** as the **Membership Type** for the output.



- d Click **OK** to generate the FIS and close the dialog box.
- 6 Export the initial FIS to the MATLAB workspace:
 - a In the **Neuro-Fuzzy Designer**, select **File > Export > To Workspace**.
This action opens a dialog box where you specify the MATLAB variable name.
 - b In the dialog box, in the **Workspace variable** text box, enter **initfis**.



- c Click **OK** to close the dialog box.

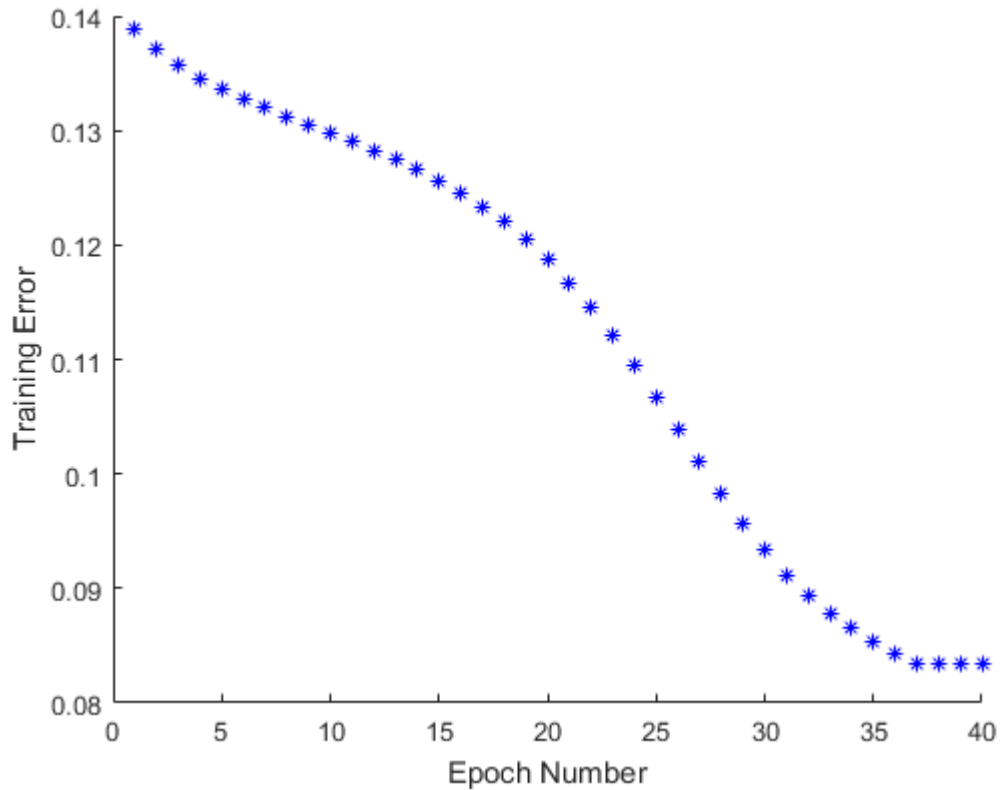
A variable named `initfis` now appears in the MATLAB workspace.

- 7 Train the FIS for 40 epochs by typing the following command at the MATLAB prompt:

```
figure
hold on
fismat = initfis;
opt = anfisOptions('EpochNumber',2,'ValidationData',fuzex1chkData);
for ct = 1:40
    opt.InitialFIS = fismat;
    [fismat,error] = anfis(fuzex1trnData,opt);
    plot(ct,error(1),'b*');
end
```

To improve accuracy when you train the FIS, the code uses the results of the current iteration returned by the `anfis` command as the initial conditions for the next iteration. The output argument `error` contains the root mean squared errors representing the training data error. For more information, see the `anfis` reference page.

The plot of the training error versus the number of epochs appears in the next figure.



See Also

Neuro-Fuzzy Designer

More About

- “Neuro-Adaptive Learning and ANFIS” on page 3-2
- “Comparison of anfis and Neuro-Fuzzy Designer Functionality” on page 3-7
- “Train Adaptive Neuro-Fuzzy Inference Systems” on page 3-13
- “Test Data Against Trained System” on page 3-18

Predict Chaotic Time-Series

This example shows how to use the command-line features of `anfis` on a chaotic time-series prediction example.

Generating a FIS using the **Neuro-Fuzzy Designer** is simple.

However, when implementing the checking data validation feature of `anfis`, you must confirm that the checking data error does what it is supposed to. Otherwise, you must retrain the FIS.

This example uses `anfis` to predict a time series generated by the following Mackey-Glass (MG) time-delay differential equation.

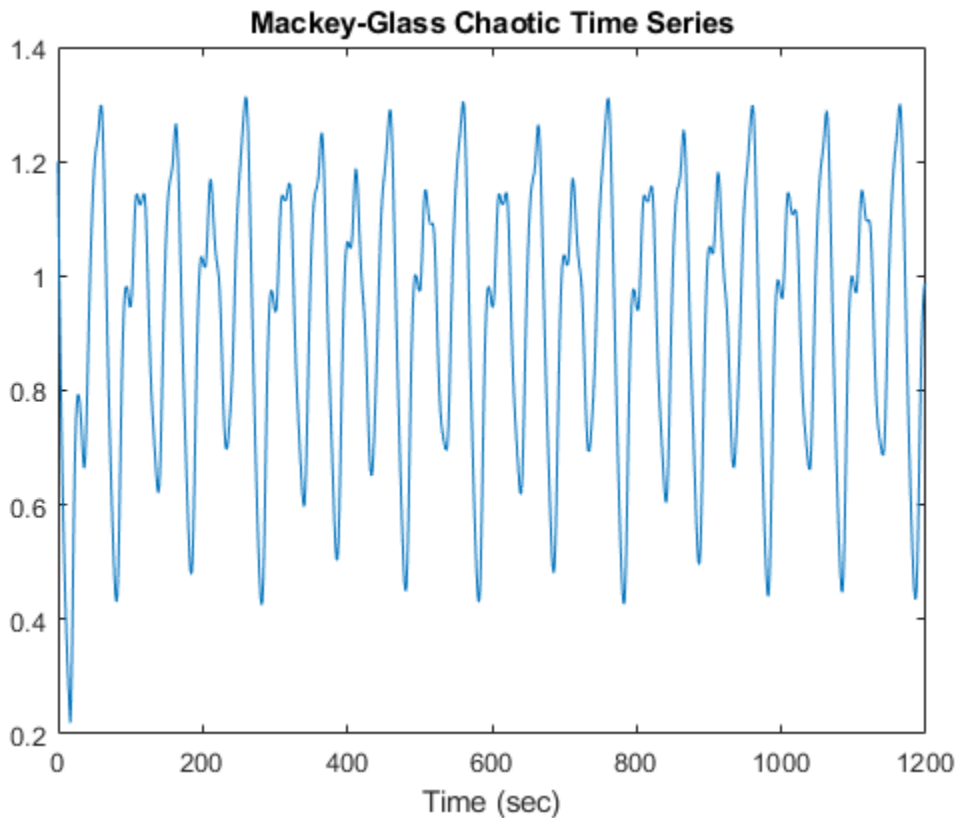
$$\dot{x}(t) = \frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)} = 0.1x(t)$$

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

To obtain the time series value at integer points, the fourth-order Runge-Kutta method was used to find the numerical solution to the previous MG equation. It was assumed that $x(0) = 1.2$, $\tau = 17$, and $x(t) = 0$ for $t < 0$. The result was saved in the file `mgdata.dat`.

Plot the MG time series.

```
load mgdata.dat
time = mgdata(:,1);
x = mgdata(:, 2);
figure(1)
plot(time,x)
title('Mackey-Glass Chaotic Time Series')
xlabel('Time (sec)')
```



In time-series prediction, you must use known values of the time series up to the point in time, t , to predict the value at some point in the future, $t + P$. The standard method for this type of prediction is to create a mapping from D sample data points, sampled every Δ units in time ($x(t - (D - 1)\Delta), \dots, x(t - \Delta), x(t)$) to a predicted future value $x = (t + P)$. Following the conventional settings for predicting the MG time series, set $D = 4$ and $\Delta = P = 6$. For each t , the input training data for anfis is a four-column vector of the following form.

$$w(t) = [x(t - 19), x(t - 12), x(t - 6), x(t)]$$

The output training data corresponds to the trajectory prediction.

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

For each t , ranging in values from 118 to 1117, the training input/output data is a structure whose first component is the four-dimensional input w , and whose second component is the output s . There are 1000 input/output data values. You use the first 500 data values for the `anfis` training, while the others are used as checking data for validating the identified fuzzy model. This division of data values results in two 500-point data model, `trnData` and `chkData`.

```
for t = 118:1117
    Data(t-117,:) = [x(t-18) x(t-12) x(t-6) x(t) x(t+6)];
end
trnData = Data(1:500,:);
chkData = Data(501:end,:);
```

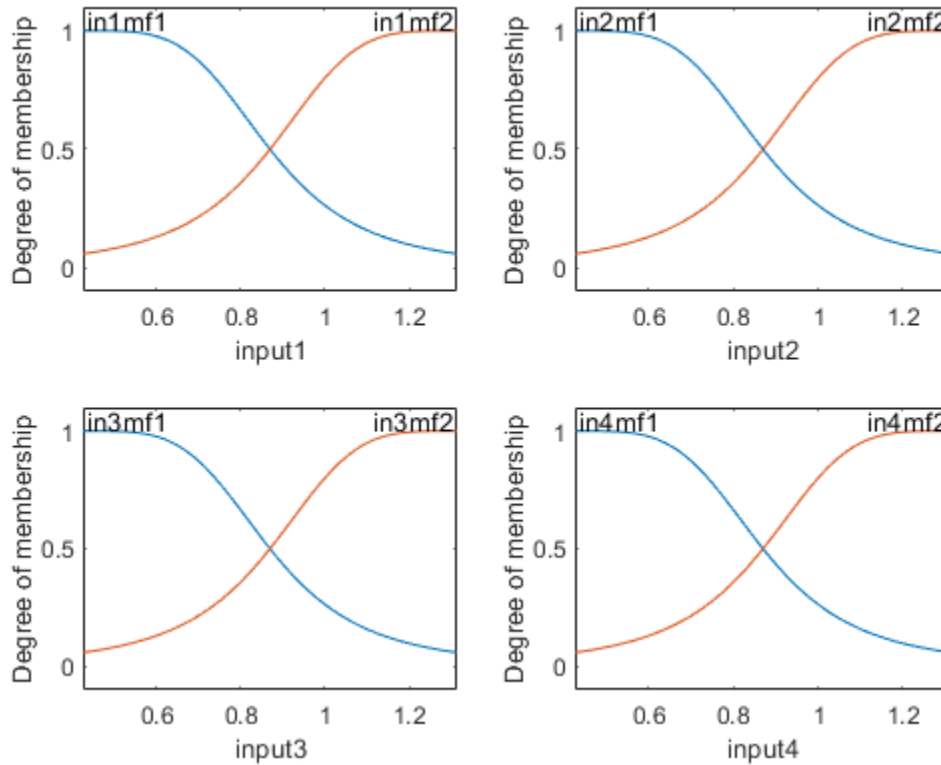
To start the training, specify an initial FIS object using the `genfis` command. By default, `genfis` creates the FIS object using grid partitioning.

```
fismat = genfis(trnData(:,1:end-1),trnData(:,end));
```

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

The `genfis` command generates initial membership functions that are equally spaced and cover the whole input space.

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



To configure training options, create an `anfisOptions` option set, specifying the initial FIS and validation data.

```
opt = anfisOptions('InitialFIS',fismat,'ValidationData',chkData);
```

Suppress the display of training information to the Command Window.

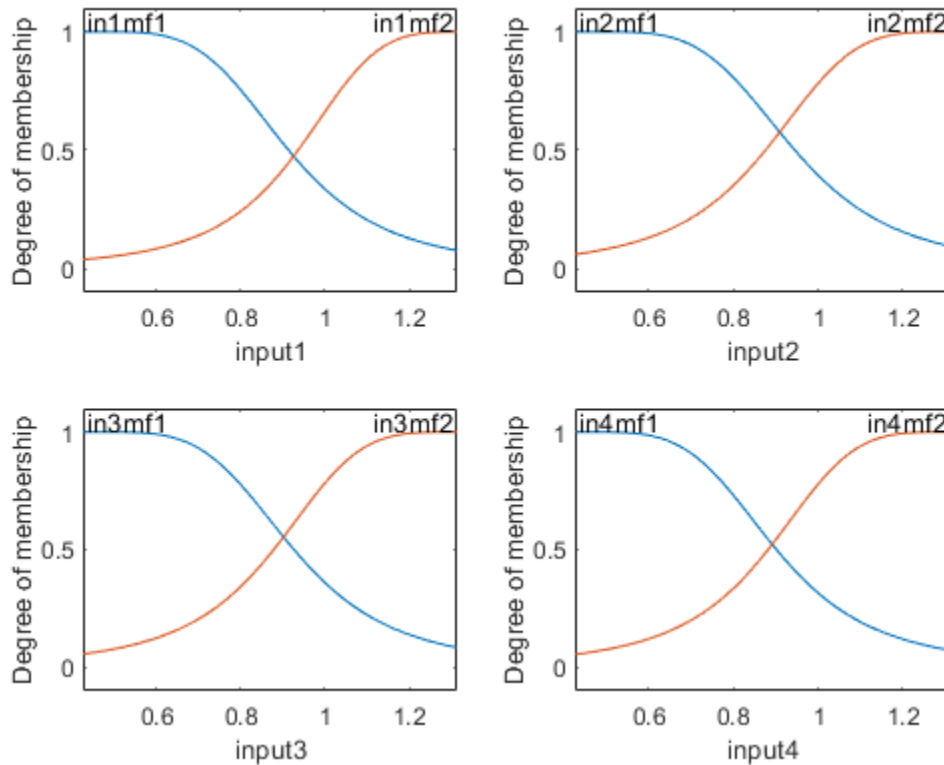
```
opt.DisplayANFISInformation = 0;  
opt.DisplayErrorValues = 0;  
opt.DisplayStepSize = 0;  
opt.DisplayFinalResults = 0;
```

Train the FIS using the specified training data and options.

```
[fismat1,error1,ss,fismat2,error2] = anfis(trnData,opt);
```

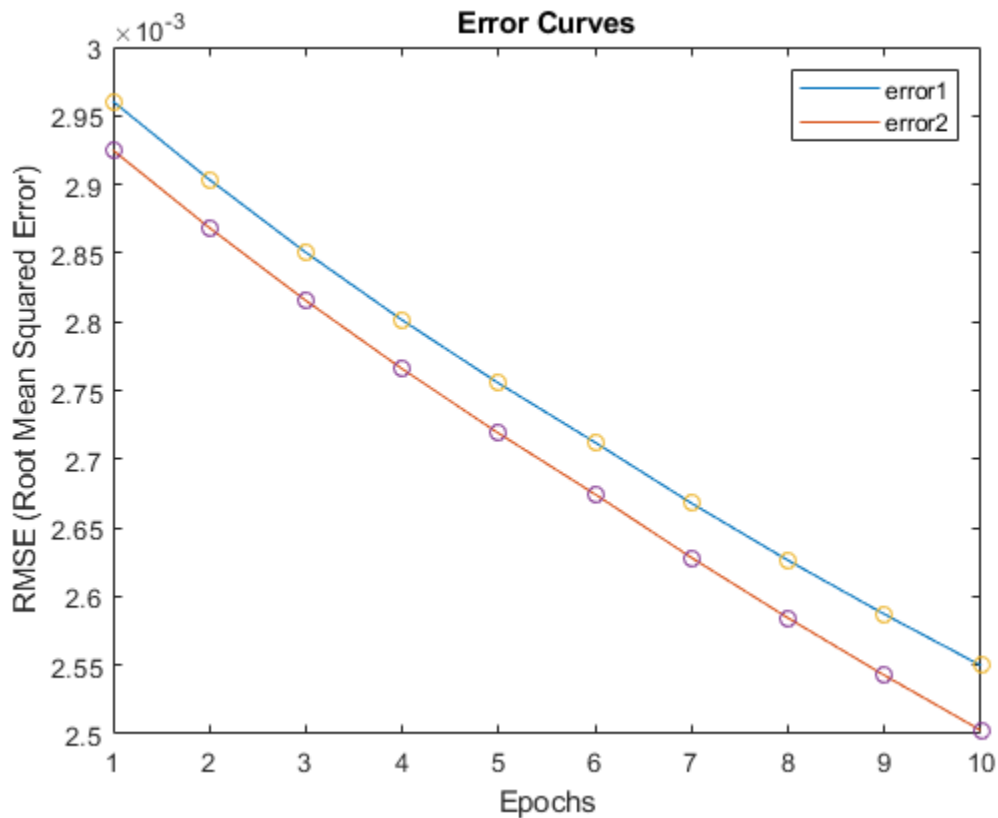
Because the `ValidationData` option is set, the final FIS you choose is the one associated with the minimum checking error. This result is stored in `fismat2`. Plots these new membership functions.

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



Plot the error signals.

```
figure(4)
plot([error1 error2])
hold on
plot([error1 error2], 'o')
legend('error1', 'error2')
xlabel('Epochs')
ylabel('RMSE (Root Mean Squared Error)')
title('Error Curves')
```

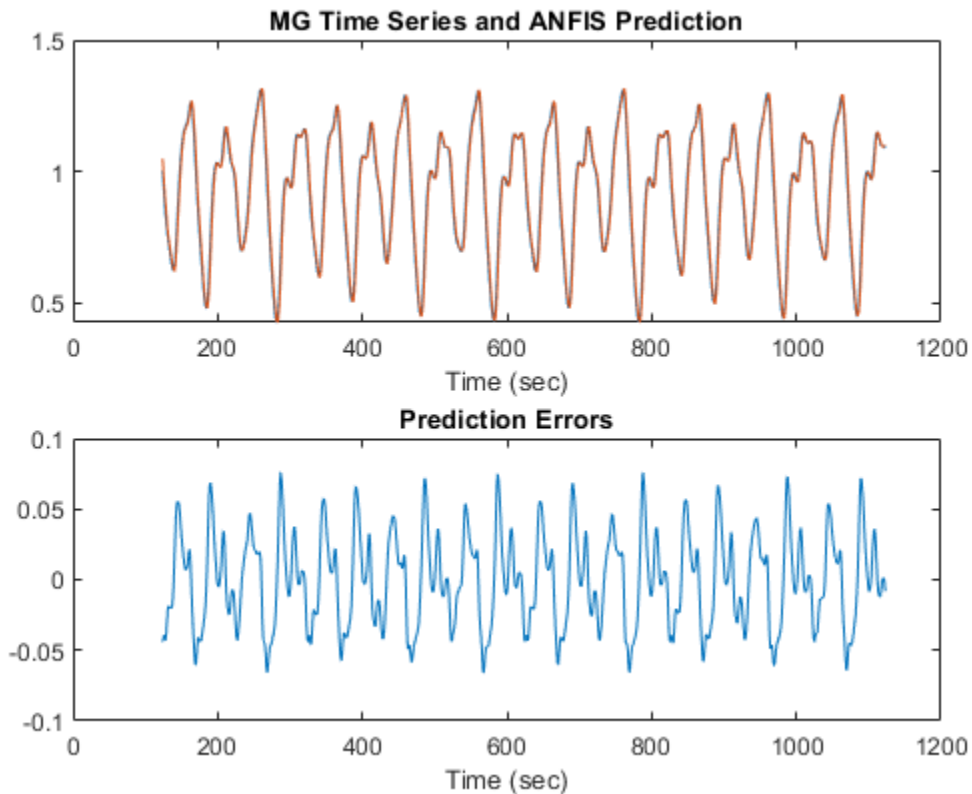


In addition to these error plots, you can plot the FIS output versus the training or checking data. To compare the original MG time series and the fuzzy prediction side by side, try:


```

figure(5)
anfis_output = evalfis(fismat2,[trnData(:,1:4); chkData(:,1:4)]);
index = 125:1124;
subplot(2,1,1)
plot(time(index),[x(index) anfis_output])
xlabel('Time (sec)')
title('MG Time Series and ANFIS Prediction')
subplot(2,1,2)
plot(time(index),x(index) - anfis_output)
xlabel('Time (sec)')
title('Prediction Errors')

```



The difference between the original MG time series and the values estimated using `anfis` is small. Therefore, you can only see one curve in the first plot. The prediction error

appears in the second plot with a much finer scale. You trained for only 10 epochs. If you apply more extensive training, you get better performance.

See Also

`anfis` | `evalfis` | `genfis`

More About

- “Neuro-Adaptive Learning and ANFIS” on page 3-2
- “Comparison of `anfis` and Neuro-Fuzzy Designer Functionality” on page 3-7

Chaotic Time-Series Prediction

This example shows how to do chaotic time series prediction using ANFIS.

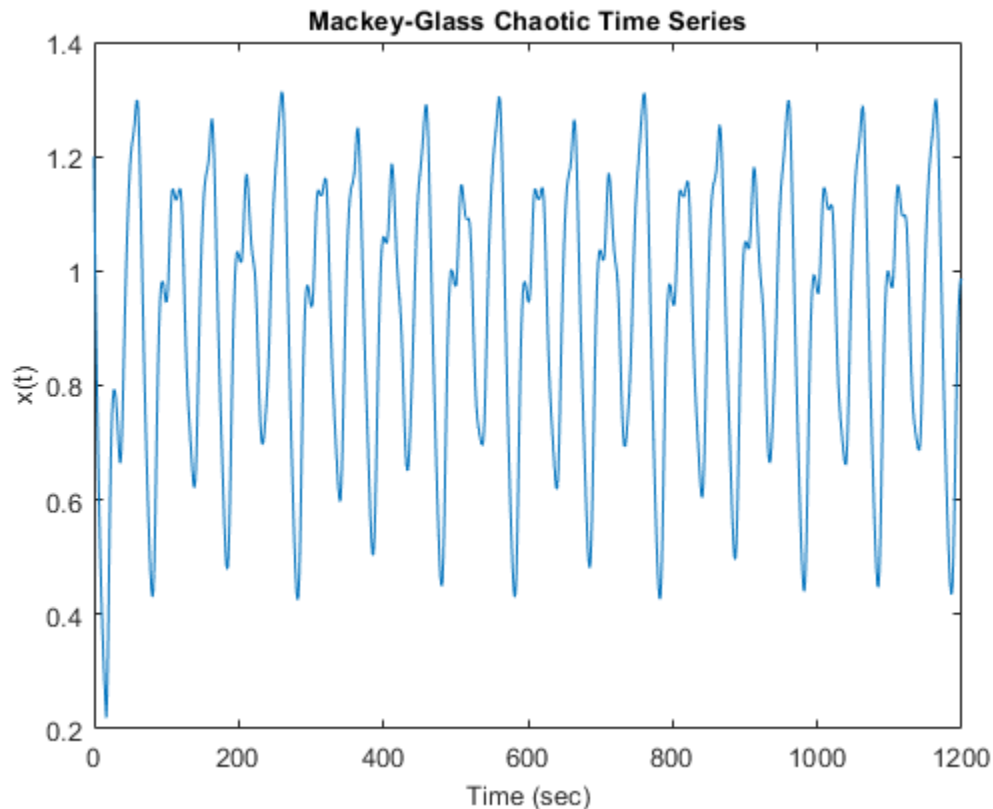
Time Series Data

The data is generated from the Mackey-Glass time-delay differential equation which is defined by

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

When $x(0) = 1.2$ and $\tau = 17$, we have a non-periodic and non-convergent time series that is very sensitive to initial conditions. (We assume $x(t) = 0$ when $t < 0$.)

```
% Load time series data
load mgdata.dat
a = mgdata;
time = a(:, 1);
x_t = a(:, 2);
plot(time, x_t);
xlabel('Time (sec)', 'fontsize', 10); ylabel('x(t)', 'fontsize', 10);
title('Mackey-Glass Chaotic Time Series', 'fontsize', 10);
```



Preprocessing the Data

Now we want to build an ANFIS that can predict $x(t+6)$ from the past values of this time series, that is, $x(t-18)$, $x(t-12)$, $x(t-6)$, and $x(t)$. Therefore the training data format is

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

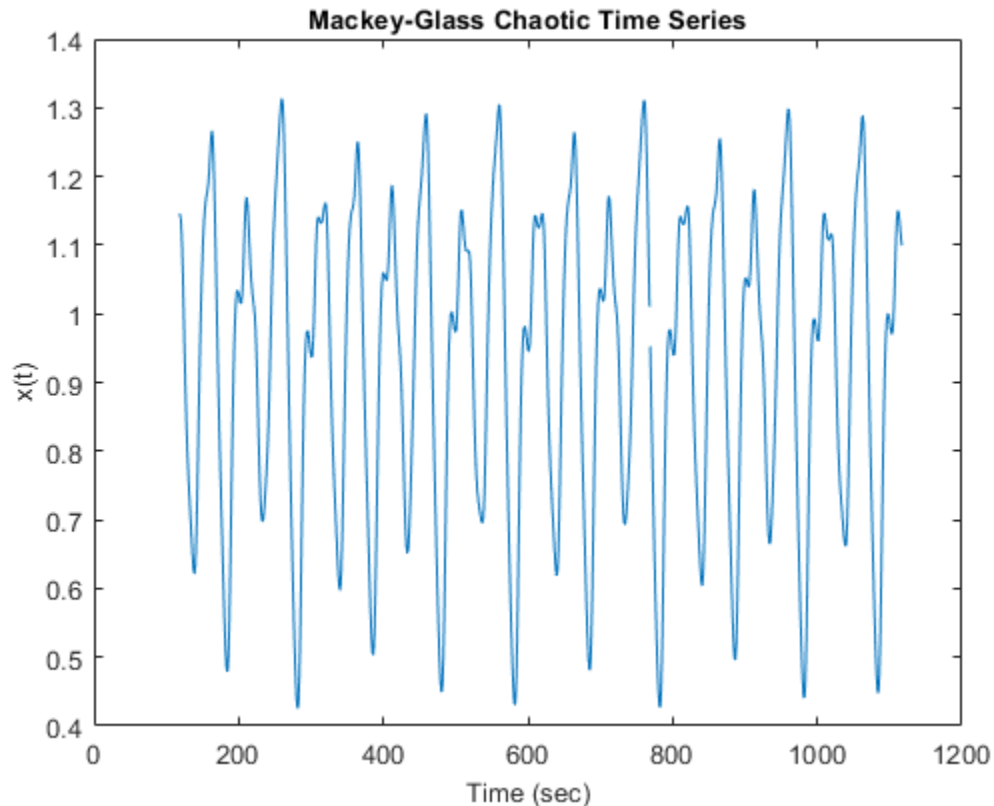
From $t = 118$ to 1117, we collect 1000 data pairs of the above format. The first 500 are used for training while the others are used for checking. The plot shows the segment of the time series where data pairs were extracted from. The first 100 data points are ignored to avoid the transient portion of the data.

```
trn_data = zeros(500, 5);  
chk_data = zeros(500, 5);
```

```
% Prepare training data
trn_data(:, 1) = x_t(101:600);
trn_data(:, 2) = x_t(107:606);
trn_data(:, 3) = x_t(113:612);
trn_data(:, 4) = x_t(119:618);
trn_data(:, 5) = x_t(125:624);

% Prepare checking data
chk_data(:, 1) = x_t(601:1100);
chk_data(:, 2) = x_t(607:1106);
chk_data(:, 3) = x_t(613:1112);
chk_data(:, 4) = x_t(619:1118);
chk_data(:, 5) = x_t(625:1124);

index = 119:1118; % ts starts with t = 0
plot(time(index), x_t(index));
xlabel('Time (sec)', 'fontsize', 10); ylabel('x(t)', 'fontsize', 10);
title('Mackey-Glass Chaotic Time Series', 'fontsize', 10);
```



Building the ANFIS Model

We use GENFIS to generate an initial FIS matrix from training data. The command is quite simple since default values for MF number (2) and MF type (`gbellmf`) are used:

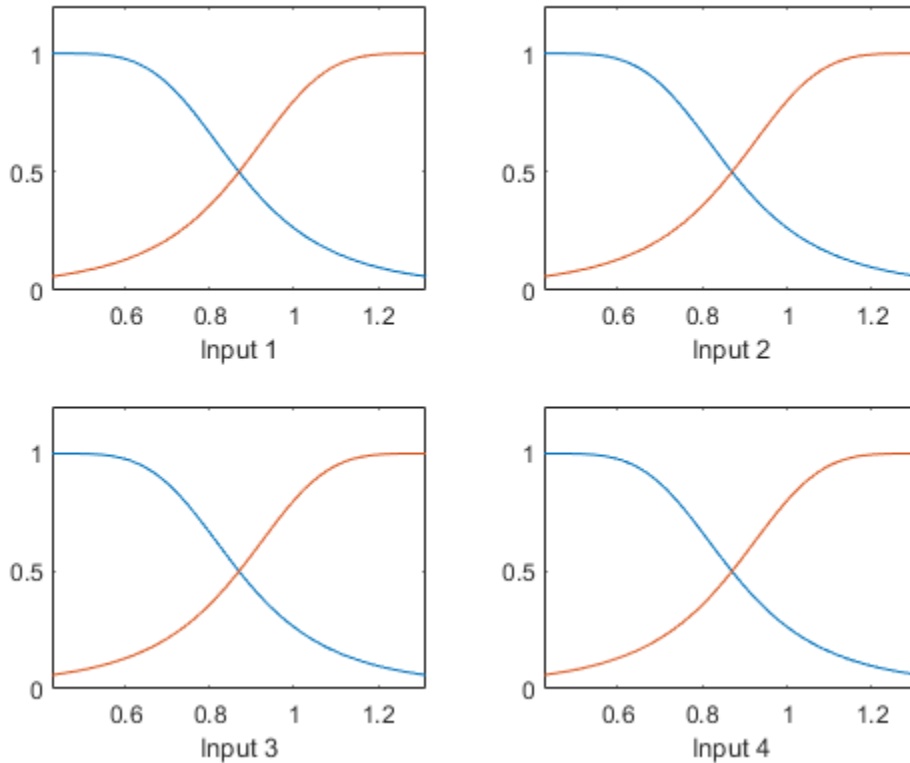
```
% Generate an initial FIS
fismat = genfis(trn_data(:,1:end-1),trn_data(:,end), ...
    genfisOptions('GridPartition'));

% The initial MFs for training are shown in the plots.
for input_index = 1:4
    subplot(2,2,input_index)
    [x,y] = plotmf(fismat,'input',input_index);
    plot(x,y)
```

```

axis([-inf inf 0 1.2]);
xlabel(['Input ' int2str(input_index)], 'fontsize', 10);
end

```



There are $2^4 = 16$ rules in the generated FIS matrix and the number of fitting parameters is 108, including 24 nonlinear parameters and 80 linear parameters. This is a proper balance between number of fitting parameters and number of training data (500). The ANFIS command uses training data to train the initial FIS as shown below.

```

% Create ANFIS training options
options = anfisOptions;
options.InitialFIS = fismat;
options.ValidationData = chk_data;

```

```
% Train FIS
[trn_fismat,trn_error,~,~,chk_error] = anfis(trn_data,options);

ANFIS info:
  Number of nodes: 55
  Number of linear parameters: 80
  Number of nonlinear parameters: 24
  Total number of parameters: 104
  Number of training data pairs: 500
  Number of checking data pairs: 500
  Number of fuzzy rules: 16

Start training ANFIS ...

   1      0.00295217      0.00292906
   2      0.00289582      0.00287246
   3      0.00284349      0.00281933
   4      0.00279482      0.00276936
   5      0.00274946      0.00272223
Step size increases to 0.011000 after epoch 5.
   6      0.00270705      0.00267764
   7      0.0026634      0.00263118
   8      0.00262247      0.00258705
   9      0.00258382      0.00254494
Step size increases to 0.012100 after epoch 9.
  10      0.00254705      0.00250455

Designated epoch number reached --> ANFIS training completed at epoch 10.

Minimal training RMSE = 0.002547
Minimal checking RMSE = 0.00250455
```

After ten epochs of training, the final MFs are shown in the plots. Note that these MFs after training do not change drastically. Obviously most of the fitting is done by the linear parameters while the nonlinear parameters are mostly for fine-tuning for further improvement.

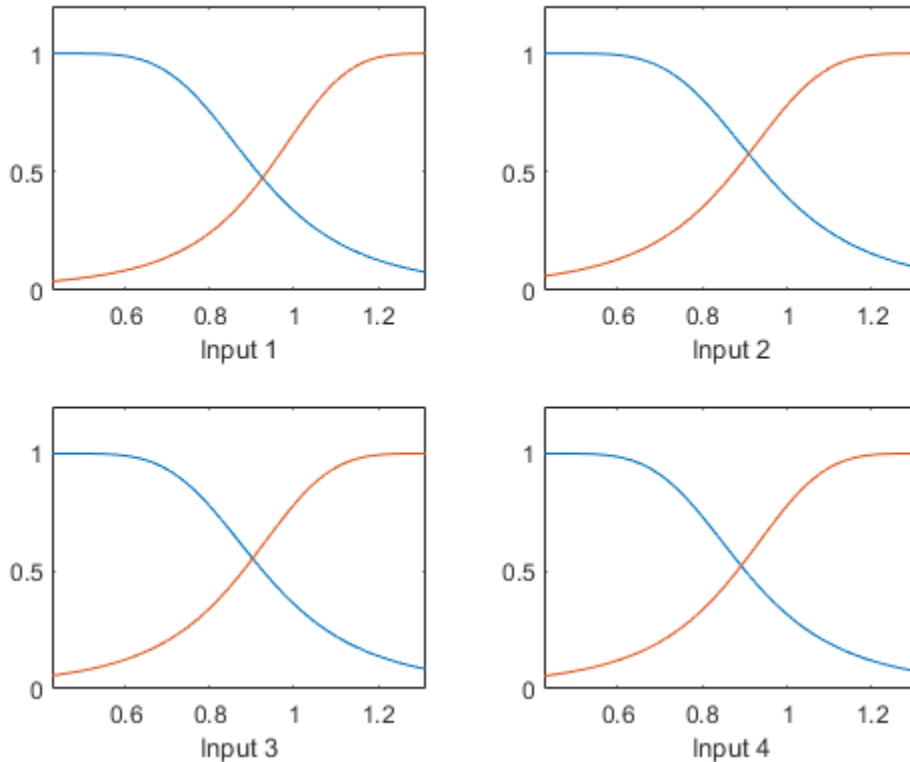
```
% Plot final MF's on x, y, z, u
for input_index=1:4
  subplot(2,2,input_index)
  [x,y]=plotmf(trn_fismat,'input',input_index);
  plot(x,y)
  axis([-inf inf 0 1.2]);
```



```

xlabel(['Input ' int2str(input_index)], 'fontsize', 10);
end

```



Error Curves

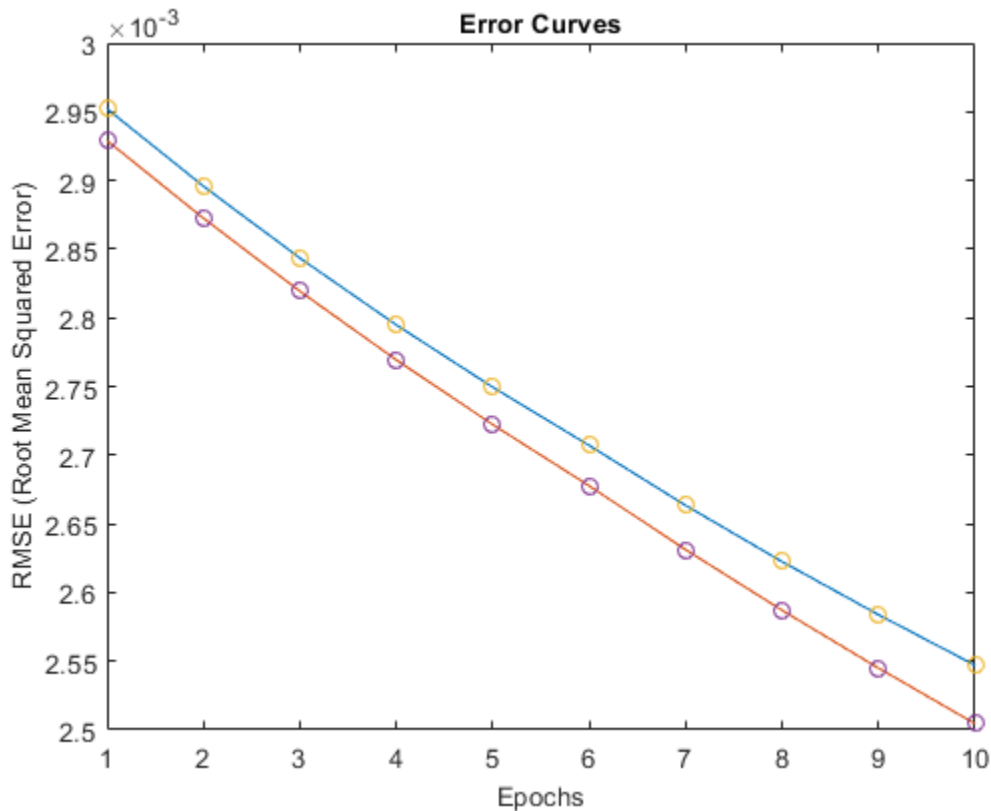
This plot displays error curves for both training and checking data. Note that the training error is higher than the checking error. This phenomenon is not uncommon in ANFIS learning or nonlinear regression in general; it could indicate that the training process is not close to finished yet.

```

% Error curves plot
close all;
epoch_n = 10;
plot([trn_error chk_error]);

```

```
hold on; plot([trn_error chk_error], 'o'); hold off;  
xlabel('Epochs','fontsize',10);  
ylabel('RMSE (Root Mean Squared Error)','fontsize',10);  
title('Error Curves','fontsize',10);
```



Comparison

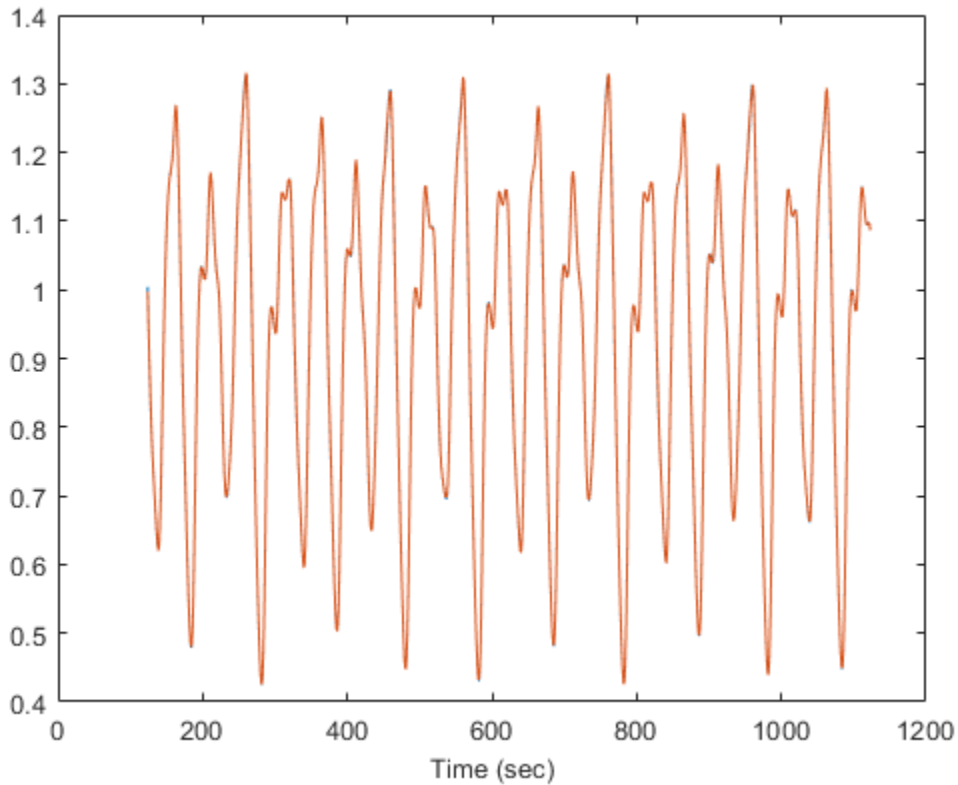
This plot shows the original time series and the one predicted by ANFIS. The difference is so tiny that it is impossible to tell one from another by eye inspection. That is why you probably see only the ANFIS prediction curve. The prediction errors must be viewed on another scale.

```
% Comparison between original and predicted time series  
input = [trn_data(:, 1:4); chk_data(:, 1:4)];
```

```

anfis_output = evalfis(trn_fismat,input);
index = 125:1124;
plot(time(index), [x_t(index) anfis_output]);
xlabel('Time (sec)', 'fontsize', 10);

```



Prediction Errors of ANFIS

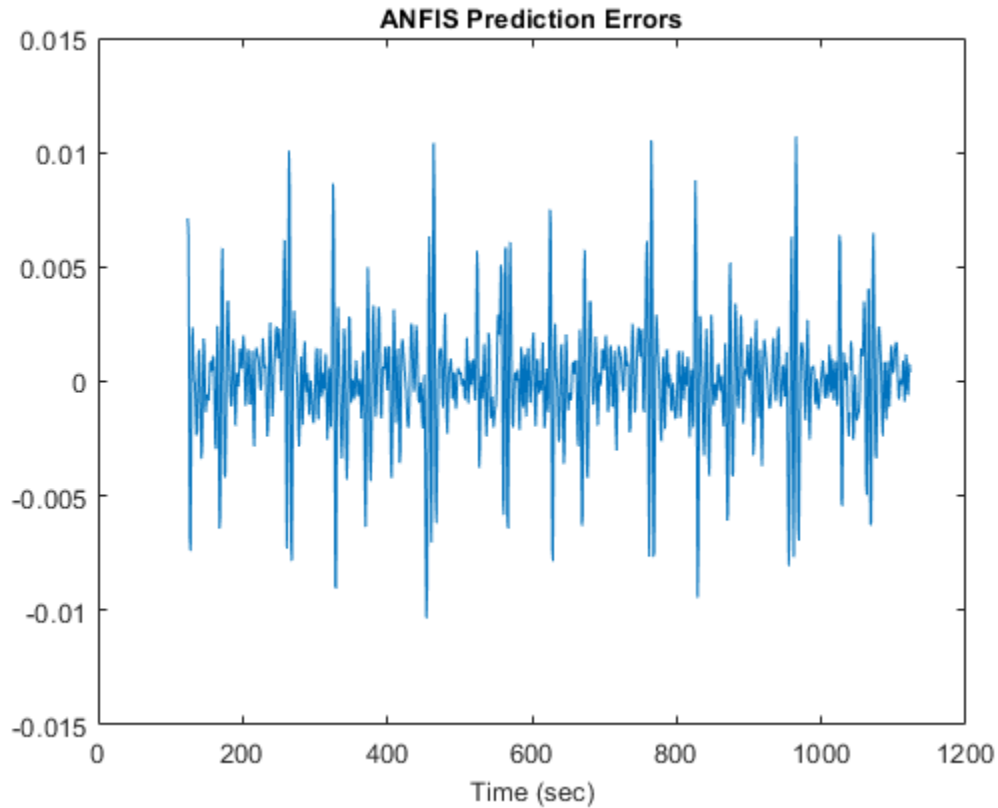
Prediction error of ANFIS is shown here. Note that the scale is about a hundredth of the scale of the previous plot. Remember that we have only 10 epochs of training in this case; better performance is expected if we have extensive training.

```

% Prediction error
diff = x_t(index)-anfis_output;
plot(time(index), diff);

```

```
xlabel('Time (sec)', 'fontsize', 10);  
title('ANFIS Prediction Errors', 'fontsize', 10);
```



See Also

`anfis` | `evalfis` | `genfis`

More About

- “Neuro-Adaptive Learning and ANFIS” on page 3-2
- “Comparison of `anfis` and Neuro-Fuzzy Designer Functionality” on page 3-7

Modeling Inverse Kinematics in a Robotic Arm

This example shows how to use a fuzzy system to model the inverse kinematics in a two-joint robotic arm.

What Is Inverse Kinematics?

Kinematics is the science of motion. In a two-joint robotic arm, given the angles of the joints, the kinematics equations give the location of the tip of the arm. Inverse kinematics refers to the reverse process. Given a desired location for the tip of the robotic arm, what should the angles of the joints be so as to locate the tip of the arm at the desired location. There is usually more than one solution and can at times be a difficult problem to solve.

This is a typical problem in robotics that needs to be solved to control a robotic arm to perform tasks it is designated to do. In a 2-dimensional input space, with a two-joint robotic arm and given the desired coordinate, the problem reduces to finding the two angles involved. The first angle is between the first arm and the ground (or whatever it is attached to). The second angle is between the first arm and the second arm.

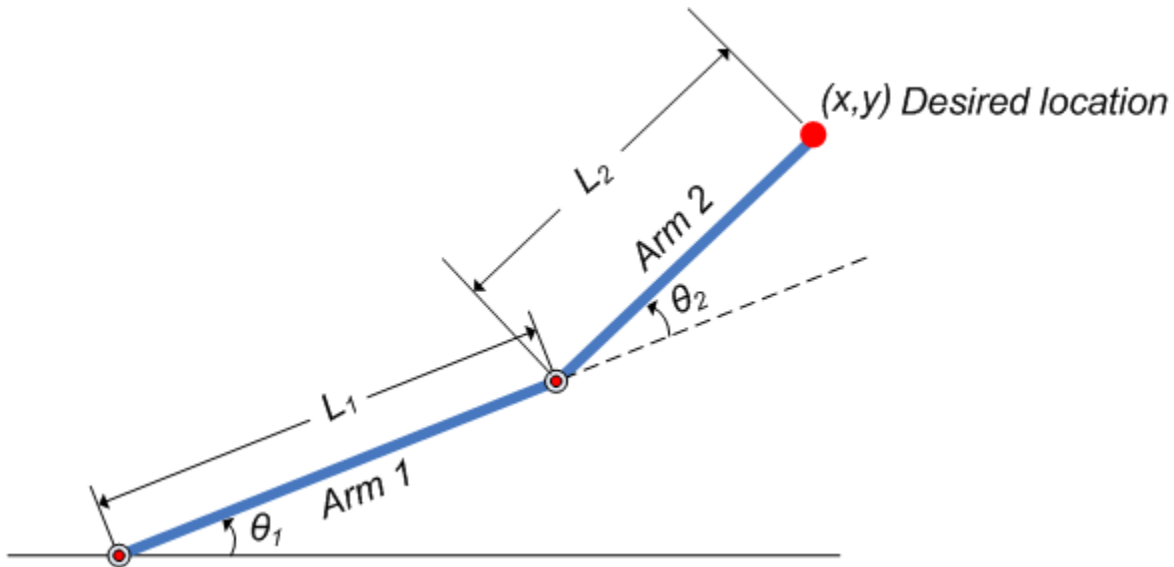


Figure 1: Illustration showing the two-joint robotic arm with the two angles, θ_1 and θ_2

Why Use Fuzzy Logic?

For simple structures like the two-joint robotic arm, it is possible to mathematically deduce the angles at the joints given the desired location of the tip of the arm. However with more complex structures (eg: n-joint robotic arms operating in a 3-dimensional input space) deducing a mathematical solution for the inverse kinematics may prove challenging.

Using fuzzy logic, we can construct a fuzzy inference system that deduces the inverse kinematics if the forward kinematics of the problem is known, hence sidestepping the need to develop an analytical solution. Also, the fuzzy solution is easily understandable and does not require special background knowledge to comprehend and evaluate it.

In the following section, a broad outline for developing such a solution is described, and later, the detailed steps are elaborated.

Overview of Fuzzy Solution

Since the forward kinematics formulae for the two-joint robotic arm are known, x and y coordinates of the tip of the arm are deduced for the entire range of angles of rotation of the two joints. The coordinates and the angles are saved to be used as training data to train an ANFIS (adaptive neuro-fuzzy inference system) network.

During training, the ANFIS network learns to map the coordinates (x , y) to the angles (θ_1 , θ_2). The trained ANFIS network is then used as a part of a larger control system to control the robotic arm. Knowing the desired location of the robotic arm, the control system uses the trained ANFIS network to deduce the angular positions of the joints and applies force to the joints of the robotic arm accordingly to move it to the desired location.

What Is ANFIS?

ANFIS stands for adaptive neuro-fuzzy inference system. It is a hybrid neuro-fuzzy technique that brings learning capabilities of neural networks to fuzzy inference systems. The learning algorithm tunes the membership functions of a Sugeno-type fuzzy inference system using the training input/output data.

In this case, the input/output data refers to the "coordinates/angles" dataset. The coordinates act as input to the ANFIS and the angles act as the output. The learning algorithm teaches the ANFIS to map the coordinates to the angles through a process called training. At the end of training, the trained ANFIS network would have learned the input-output map and be ready to be deployed into the larger control system solution.

Data Generation

Let θ_1 be the angle between the first arm and the ground. Let θ_2 be the angle between the second arm and the first arm (Refer to Figure 1 for illustration). Let the length of the first arm be l_1 and that of the second arm be l_2 .

Assume that the first joint has limited freedom to rotate and it can rotate between 0 and 90 degrees. Similarly, assume that the second joint has limited freedom to rotate and can rotate between 0 and 180 degrees. (This assumption takes away the need to handle some special cases which will confuse the discourse.) Hence, $0 \leq \theta_1 \leq \pi/2$ and $0 \leq \theta_2 \leq \pi$.

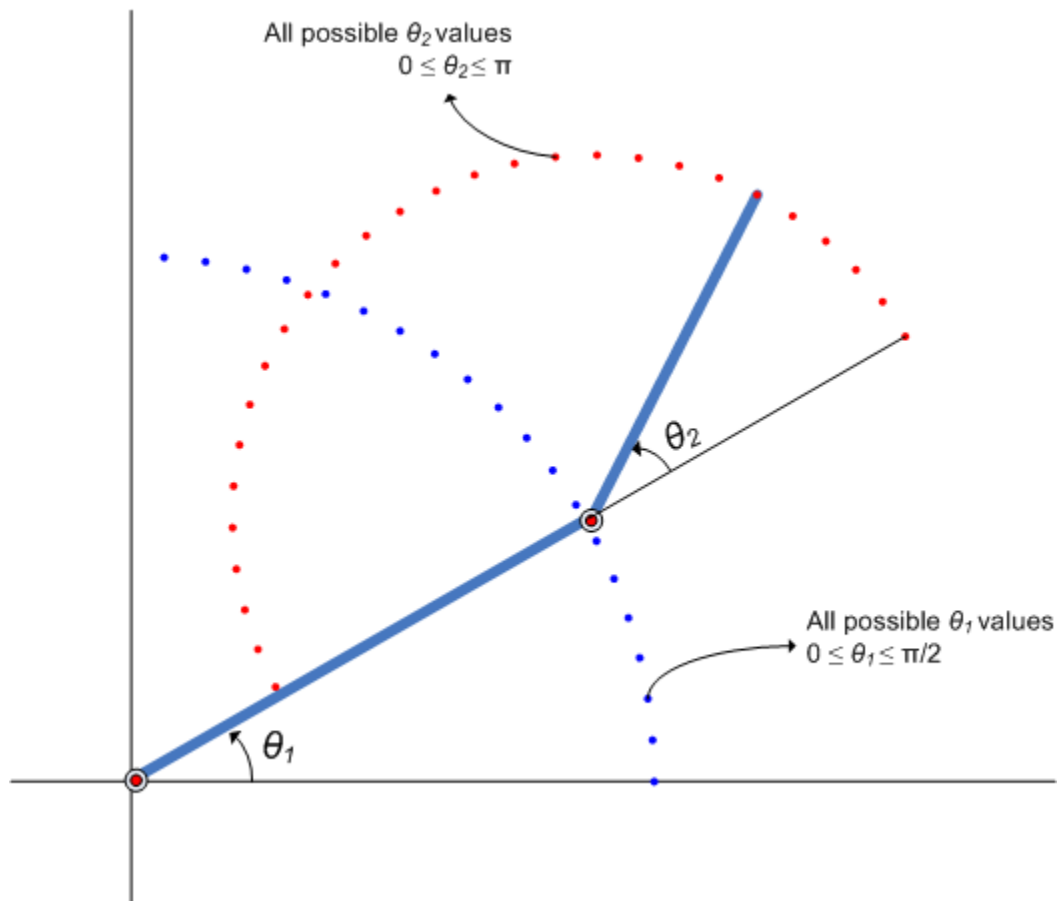


Figure 2: Illustration showing all possible `theta1` and `theta2` values.

Now, for every combination of `theta1` and `theta2` values the x and y coordinates are deduced using forward kinematics formulae.

The following code snippet shows how data is generated for all combination of `theta1` and `theta2` values and saved into a matrix to be used as training data. The reason for saving the data in two matrices is explained in the following section.

```
l1 = 10; % length of first arm
l2 = 7; % length of second arm

theta1 = 0:0.1:pi/2; % all possible theta1 values
theta2 = 0:0.1:pi; % all possible theta2 values

[THETA1,THETA2] = meshgrid(theta1,theta2); % generate a grid of theta1 and theta2 values

X = l1 * cos(THETA1) + l2 * cos(THETA1 + THETA2); % compute x coordinates
Y = l1 * sin(THETA1) + l2 * sin(THETA1 + THETA2); % compute y coordinates

data1 = [X(:) Y(:) THETA1(:)]; % create x-y-theta1 dataset
data2 = [X(:) Y(:) THETA2(:)]; % create x-y-theta2 dataset
```

[Click here for unvectorized code](#)

The following plot shows all the X-Y data points generated by cycling through different combinations of `theta1` and `theta2` and deducing x and y coordinates for each. The plot can be generated by using the code-snippet shown below. The plot is illustrated further for easier understanding.

```
plot(X(:),Y(:),'r. ');
axis equal;
xlabel('X','fontsize',10)
ylabel('Y','fontsize',10)
title('X-Y coordinates generated for all theta1 and theta2 combinations using forward
```

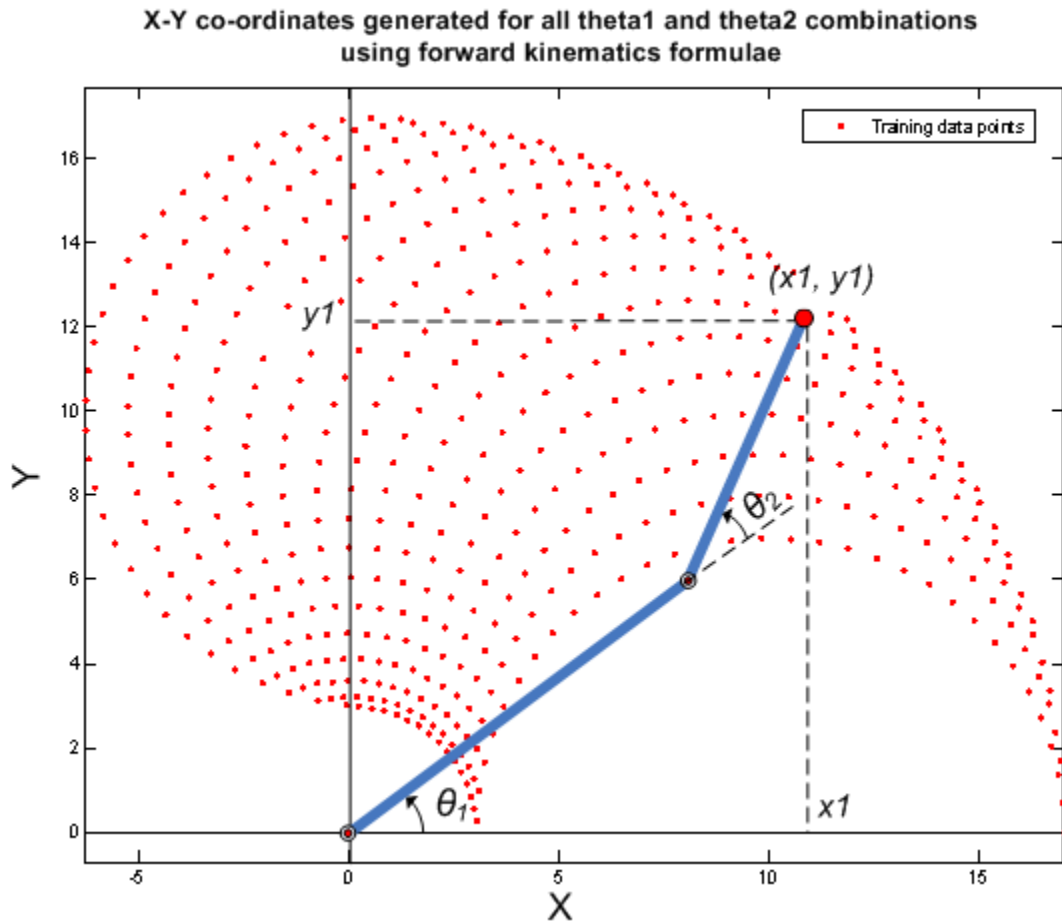



Figure 3: X-Y coordinates generated for all theta1 and theta2 combinations using forward kinematics formulae

Building ANFIS Networks

One approach to building an ANFIS solution for this problem, is to build two ANFIS networks, one to predict theta1 and the other to predict theta2.

In order for the ANFIS networks to be able to predict the angles they have to be trained with sample input-output data. The first ANFIS network will be trained with X and Y coordinates as input and corresponding theta1 values as output. The matrix data1

contains the `x-y-theta1` dataset required to train the first ANFIS network. Therefore `data1` will be used as the dataset to train the first ANFIS network.

Similarly, the second ANFIS network will be trained with X and Y coordinates as input and corresponding `theta2` values as output. The matrix `data2` contains the `x-y-theta2` dataset required to train the second ANFIS network. Therefore `data2` will be used as the dataset to train the second ANFIS network.

To train an ANFIS network, first specify the training options using the `anfisOptions` command. For this example, specify an FIS object with 7 membership functions for each input variable. Train the system for 150 epochs and suppress the Command Window display of training information.

```
opt = anfisOptions;  
opt.InitialFIS = 7;  
opt.EpochNumber = 150;  
opt.DisplayANFISInformation = 0;  
opt.DisplayErrorValues = 0;  
opt.DisplayStepSize = 0;  
opt.DisplayFinalResults = 0;
```

Train an ANFIS system using the first set of training data, `data1`.

```
disp('--> Training first ANFIS network.')
```

```
--> Training first ANFIS network.
```

```
anfis1 = anfis(data1,opt);
```

Change the number of input membership functions and train an ANFIS system using the second set of training data, `data2`.

```
disp('--> Training second ANFIS network.')
```

```
--> Training second ANFIS network.
```

```
opt.InitialFIS = 6;  
anfis2 = anfis(data2,opt);
```

For this example, the number of input membership functions and training epochs were selected based on experimentation with different potential values.

`anfis1` and `anfis2` represent the two trained ANFIS networks that will be deployed in the larger control system.

Once the training is complete, the two ANFIS networks have learned to approximate the angles (`theta1`, `theta2`) as a function of the coordinates (`x`, `y`). One advantage of using the fuzzy approach is that the ANFIS network can now approximate the angles for coordinates that are similar but not exactly the same as it was trained with. For example, the trained ANFIS networks are now capable of approximating the angles for coordinates that lie between two points that were included in the training dataset. This will allow the final controller to move the arm smoothly in the input space.

We now have two trained ANFIS networks which are ready to be deployed into the larger system that will utilize these networks to control the robotic arms.

Validating ANFIS Networks

Having trained the networks, an important follow up step is to validate the networks to determine how well the ANFIS networks would perform inside the larger control system.

Since this example problem deals with a two-joint robotic arm whose inverse kinematics formulae can be derived, it is possible to test the answers that the ANFIS networks produce with the answers from the derived formulae.

Let's assume that it is important for the ANFIS networks to have low errors within the operating range $0 < x < 2$ and $8 < y < 10$.

```
x = 0:0.1:2; % x coordinates for validation
y = 8:0.1:10; % y coordinates for validation
```

The `theta1` and `theta2` values are deduced mathematically from the `x` and `y` coordinates using inverse kinematics formulae.

```
[X,Y] = meshgrid(x,y);

c2 = (X.^2 + Y.^2 - l1^2 - l2^2)/(2*l1*l2);
s2 = sqrt(1 - c2.^2);
THETA2D = atan2(s2,c2); % theta2 is deduced

k1 = l1 + l2.*c2;
k2 = l2*s2;
THETA1D = atan2(Y,X) - atan2(k2,k1); % theta1 is deduced
```

[Click here for unvectorized code](#)

`THETA1D` and `THETA2D` are the variables that hold the values of `theta1` and `theta2` deduced using the inverse kinematics formulae.

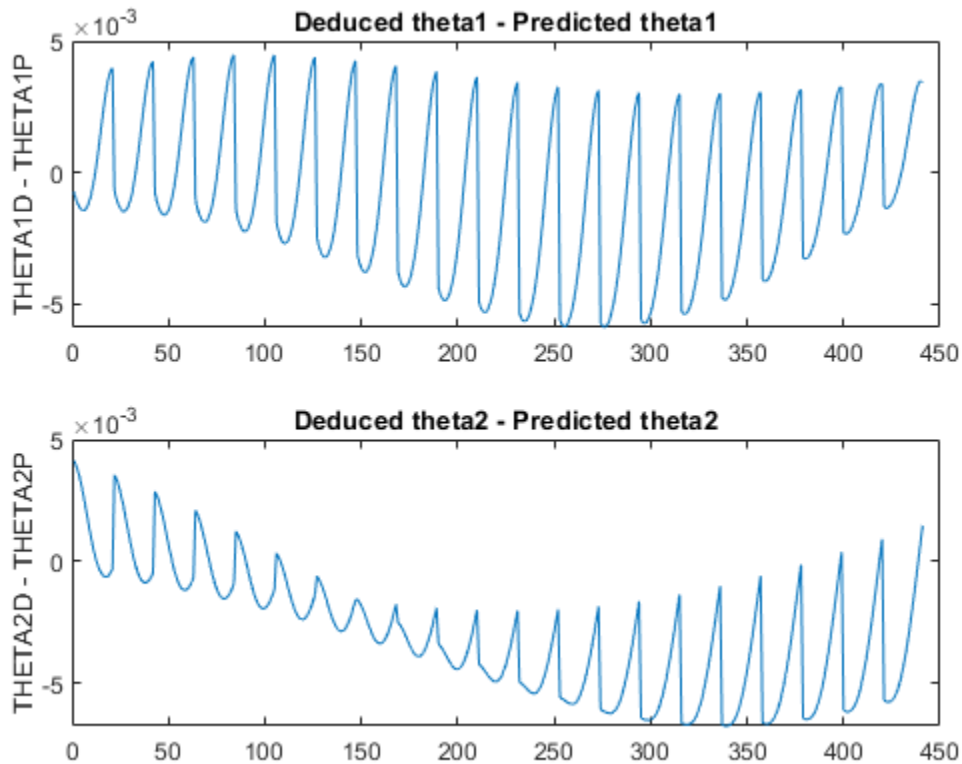
theta1 and theta2 values predicted by the trained anfis networks are obtained by using the command `evalfis` which evaluates a FIS for the given inputs.

Here, `evalfis` is used to find out the FIS outputs for the same x-y values used earlier in the inverse kinematics formulae.

```
XY = [X(:) Y(:)];  
THETA1P = evalfis(anfis1,XY); % theta1 predicted by anfis1  
THETA2P = evalfis(anfis2,XY); % theta2 predicted by anfis2
```

Now, we can see how close the FIS outputs are with respect to the deduced values.

```
theta1diff = THETA1D(:) - THETA1P;  
theta2diff = THETA2D(:) - THETA2P;  
  
subplot(2,1,1);  
plot(theta1diff);  
ylabel('THETA1D - THETA1P','fontsize',10)  
title('Deduced theta1 - Predicted theta1','fontsize',10)  
  
subplot(2,1,2);  
plot(theta2diff);  
ylabel('THETA2D - THETA2P','fontsize',10)  
title('Deduced theta2 - Predicted theta2','fontsize',10)
```



The errors are in the $1e-3$ range which is a fairly good number for the application it is being used in. However this may not be acceptable for another application, in which case the parameters to the `anfis` function may be tweaked until an acceptable solution is arrived at. Also, other techniques like input selection and alternate ways to model the problem may be explored.

Building a Solution Around the Trained ANFIS Networks

Now given a specific task, such as robots picking up an object in an assembly line, the larger control system will use the trained ANFIS networks as a reference, much like a lookup table, to determine what the angles of the arms must be, given a desired location for the tip of the arm. Knowing the desired angles and the current angles of the joints, the

system will apply force appropriately on the joints of the arms to move them towards the desired location.

The `invkin` command launches a GUI that shows how the two trained ANFIS networks perform when asked to trace an ellipse.

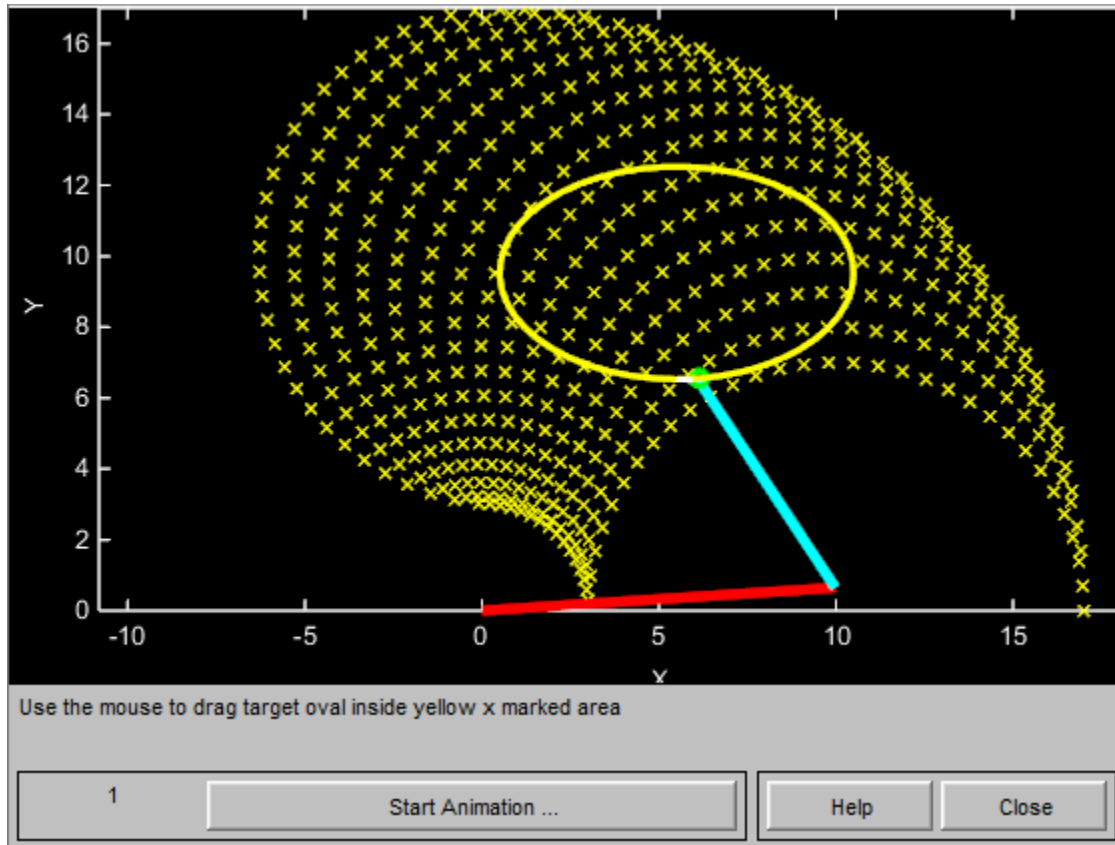


Figure 4: GUI for Inverse Kinematics Modeling.

The two ANFIS networks used in the example have been pretrained and are deployed into a larger system that controls the tip of the two-joint robot arm to trace an ellipse in the input space.

The ellipse to be traced can be moved around. Move the ellipse to a slightly different location and observe how the system responds by moving the tip of the robotic arm from

its current location to the closest point on the new location of the ellipse. Also observe that the system responds smoothly as long as the ellipse to be traced lies within the 'x' marked spots which represent the data grid that was used to train the networks. Once the ellipse is moved outside the range of data it was trained with, the ANFIS networks respond unpredictably. This emphasizes the importance of having relevant and representative data for training. Data must be generated based on the expected range of operation to avoid such unpredictability and instability issues.

See Also

`anfis` | `evalfis`

More About

- “Neuro-Adaptive Learning and ANFIS” on page 3-2
- “Comparison of `anfis` and Neuro-Fuzzy Designer Functionality” on page 3-7

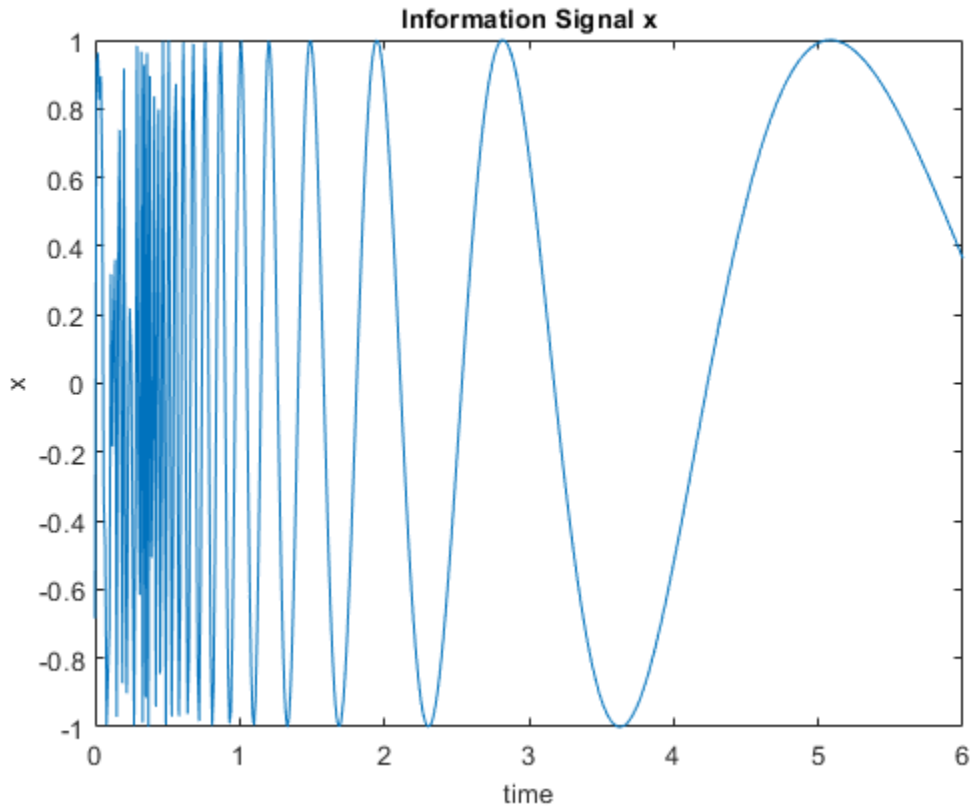
Adaptive Noise Cancellation Using ANFIS

This example shows how to do adaptive nonlinear noise cancellation using the `anfis` and `genfis` commands.

Signal and Noise

Define a hypothetical information signal, `x`, sampled at 100Hz over 6 seconds.

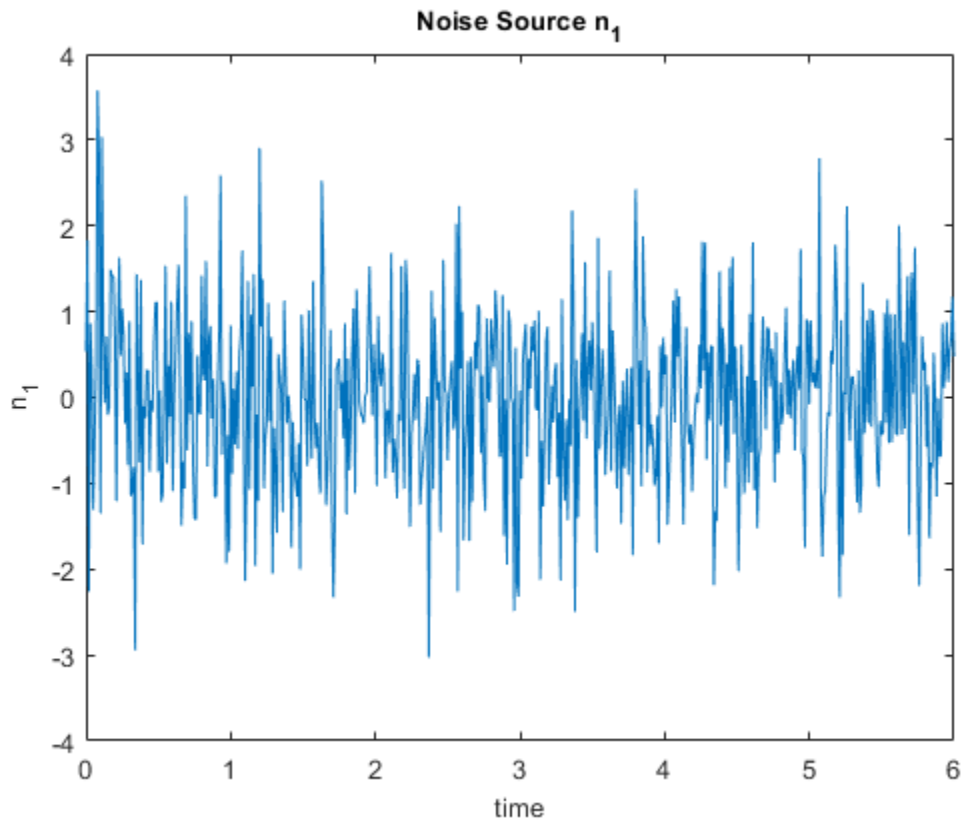
```
time = (0:0.01:6)';  
x = sin(40./(time+0.01));  
plot(time,x)  
title('Information Signal x','fontsize',10)  
xlabel('time','fontsize',10)  
ylabel('x','fontsize',10)
```

Assume that x cannot be measured without an interference signal, n_2 , which is generated from another noise source, n_1 , via a certain unknown nonlinear process.

The plot below shows noise source n_1 .

```
n1 = randn(size(time));  
plot(time,n1)  
title('Noise Source n_1','fontsize',10)  
xlabel('time','fontsize',10)  
ylabel('n_1','fontsize',10)
```



Assume that the interference signal, n_2 , that appears in the measured signal is generated via an unknown nonlinear equation:

$$n_2(k) = \frac{4 \sin(n_1(k)) \cdot n_1(k-1)}{1 + n_1(k-1)^2}$$

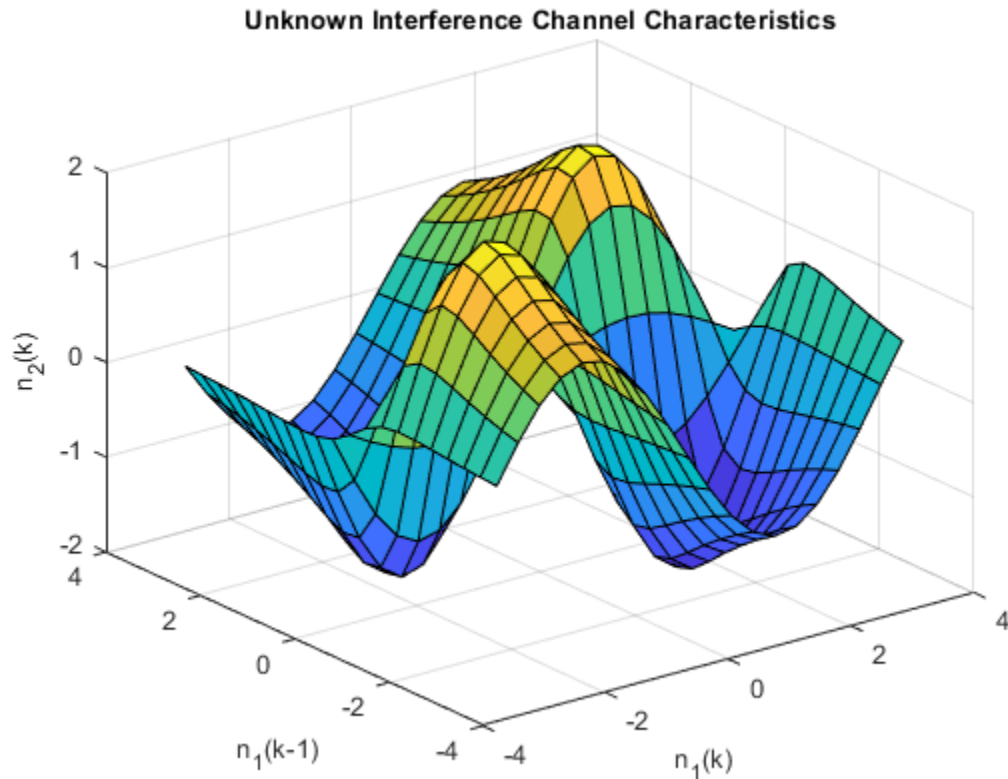
Plot this nonlinear characteristic as a surface.

```
domain = linspace(min(n1),max(n1),20);
[xx,yy] = meshgrid(domain,domain);
zz = 4*sin(xx).*yy./(1+yy.^2);
```

```

surf(xx,yy,zz);
xlabel('n_1(k)', 'fontsize',10);
ylabel('n_1(k-1)', 'fontsize',10);
zlabel('n_2(k)', 'fontsize',10);
title('Unknown Interference Channel Characteristics', 'fontsize',10);

```



Compute the interference signal, n_2 , from the noise source, n_1 , and plot both signals.

```

n1d0 = n1; % n1 with delay 0
n1d1 = [0; n1d0(1:length(n1d0)-1)]; % n1 with delay 1
n2 = 4*sin(n1d0).*n1d1./(1+n1d1.^2); % interference

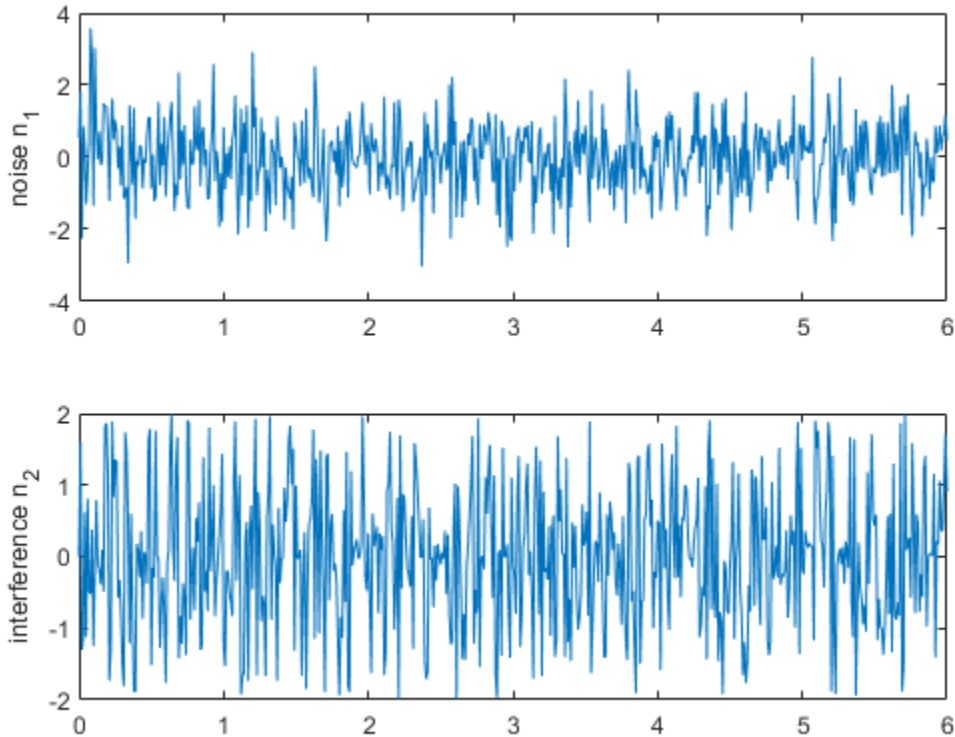
```

```

subplot(2,1,1)
plot(time,n1);

```

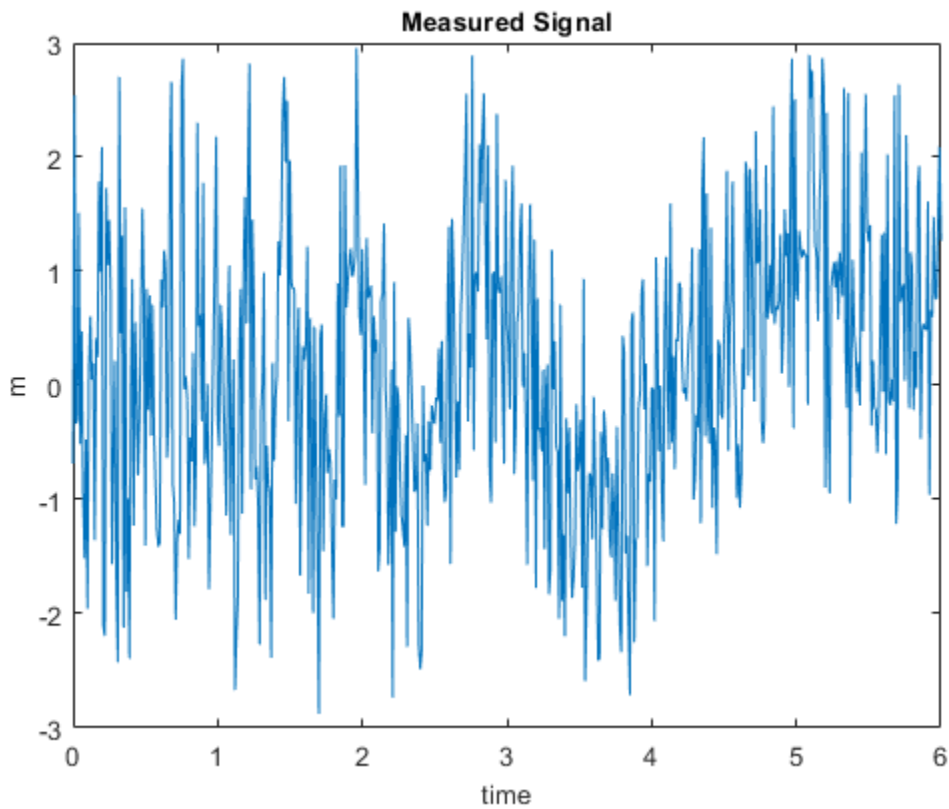
```
ylabel('noise n_1','fontsize',10);  
subplot(2,1,2)  
plot(time,n2);  
ylabel('interference n_2','fontsize',10);
```



n_2 is related to n_1 via the highly nonlinear process shown previously; from the plots, it is hard to see if these two signals are correlated in any way.

The measured signal, m , is the sum of the original information signal, x , and the interference, n_2 . However, we do not know n_2 . The only signals available to us are the noise signal, n_1 , and the measured signal m .

```
m = x + n2;           % measured signal
subplot(1,1,1)
plot(time, m)
title('Measured Signal','fontsize',10)
xlabel('time','fontsize',10)
ylabel('m','fontsize',10)
```



You can recover the original information signal, x , using adaptive noise cancellation via ANFIS training.

Build the ANFIS Model

Use the `anfis` command to identify the nonlinear relationship between n_1 and n_2 . While n_2 is not directly available, you can assume that `m` is a "contaminated" version of n_2 for training. This assumption treats `x` as "noise" in this kind of nonlinear fitting.

Assume the order of the nonlinear channel is known (in this case, 2), so you can use a 2-input ANFIS model for training.

Define the training data. The first two columns of `data` are the inputs to the ANFIS model, n_1 and a delayed version of n_1 . The final column of `data` is the measured signal, `m`.

```
delayed_n1 = [0; n1(1:length(n1)-1)];  
data = [delayed_n1 n1 m];
```

Generate the initial FIS object. By default, the grid partitioning algorithm uses two membership functions for each input variable, which produces four fuzzy rules for learning.

```
genOpt = genfisOptions('GridPartition');  
inFIS = genfis(data(:,1:end-1),data(:,end),genOpt);
```

Tune the FIS using the `anfis` command with an initial training step size of 0.2.

```
trainOpt = anfisOptions('InitialFIS',inFIS,'InitialStepSize',0.2);  
outFIS = anfis(data,trainOpt);
```

```
ANFIS info:  
  Number of nodes: 21  
  Number of linear parameters: 12  
  Number of nonlinear parameters: 12  
  Total number of parameters: 24  
  Number of training data pairs: 601  
  Number of checking data pairs: 0  
  Number of fuzzy rules: 4
```

```
Start training ANFIS ...
```

```
 1      0.761817  
 2      0.748426
```

```

3      0.739315
4      0.733993
5      0.729492
Step size increases to 0.220000 after epoch 5.
6      0.725382
7      0.721269
8      0.717621
9      0.714474
Step size increases to 0.242000 after epoch 9.
10     0.71207

```

Designated epoch number reached --> ANFIS training completed at epoch 10.

Minimal training RMSE = 0.712070

The tuned FIS, outFIS, models the second-order relationship between n_1 and n_2 .

Evaluate Model

Calculate the estimated interference signal, `estimated_n2`, by evaluating the tuned FIS using the original training data.

```
estimated_n2 = evalfis(outFIS,data(:,1:2));
```

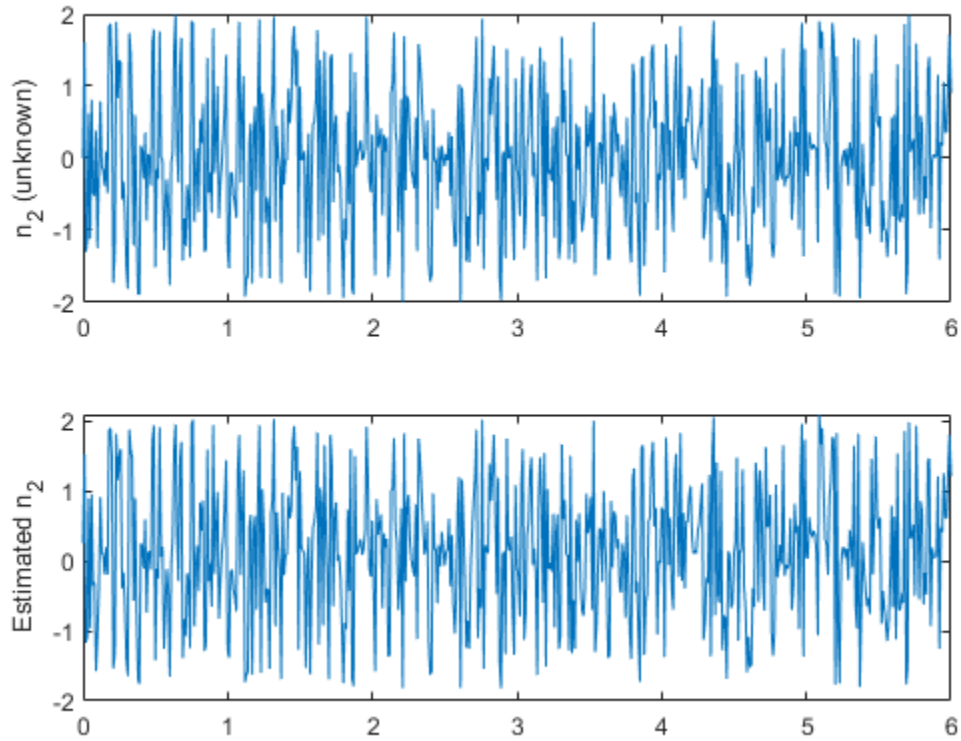
Plot the and actual n_2 signal and the estimated version from the ANFIS output.

```

subplot(2,1,1)
plot(time, n2)
ylabel('n_2 (unknown)');

subplot(2,1,2)
plot(time, estimated_n2)
ylabel('Estimated n_2');

```

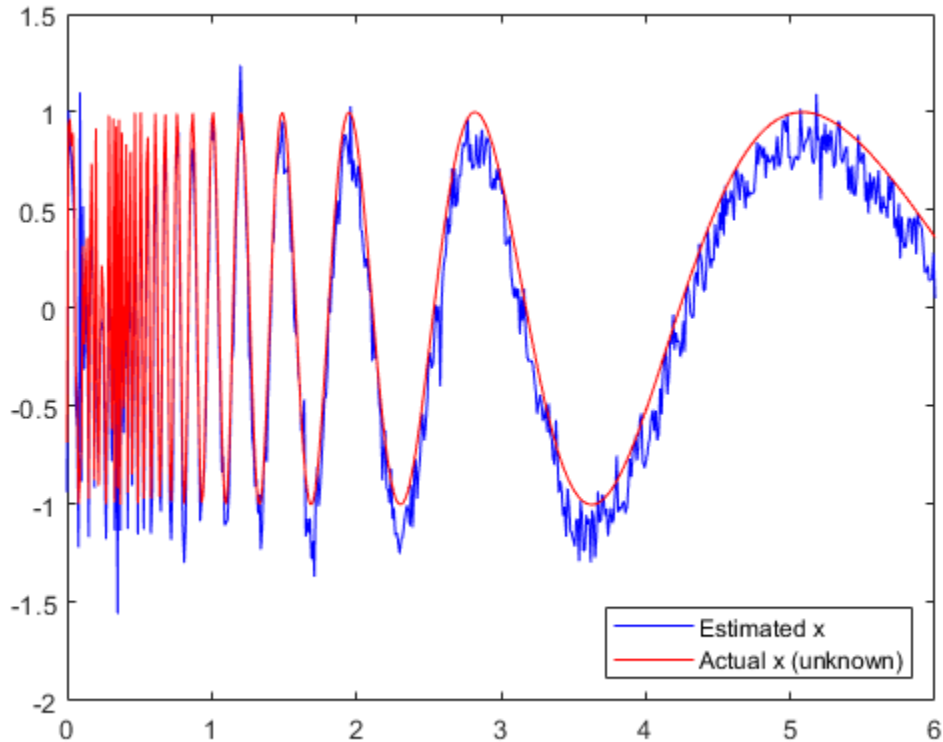


The estimated information signal is equal to the difference between the measured signal, m , and the estimated interference (ANFIS output).

```
estimated_x = m - estimated_n2;
```

Compare the original information signal, x , and the estimate, $estimated_x$.

```
figure  
plot(time,estimated_x,'b',time,x,'r')  
legend('Estimated x','Actual x (unknown)','Location','SouthEast')
```

Without extensive training, the ANFIS produces a good estimate of the information signal.

See Also

`anfis` | `evalfis` | `genfis`

More About

- “Neuro-Adaptive Learning and ANFIS” on page 3-2
- “Comparison of `anfis` and Neuro-Fuzzy Designer Functionality” on page 3-7

Nonlinear System Identification

This example shows how to use `anfis` command for nonlinear dynamic system identification.

This example requires System Identification Toolbox™, as a comparison is made between a nonlinear ANFIS and a linear ARX model.

Problem Setup

Exit if System Identification Toolbox is not available.

```
if ~fuzzychecktoolboxinstalled('ident')
    errordlg('DRYDEMO needs the System Identification Toolbox.');
```

```
return;
end
```

The data set for ANFIS and ARX modeling was obtained from a laboratory device called Feedback's Process Trainer PT 326, as described in Chapter 17 of Prof. Lennart Ljung's book "System Identification, Theory for the User", Prentice-Hall, 1987. The device functions like a hair dryer: air is fanned through a tube and heated at the inlet. The air temperature is measured by a thermocouple at the outlet. The input $u(k)$ is the voltage over a mesh of resistor wires to heat incoming air; the output $y(k)$ is the outlet air temperature.



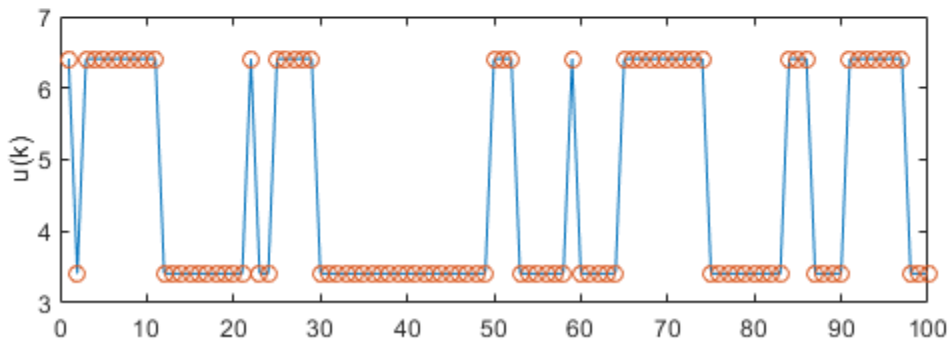
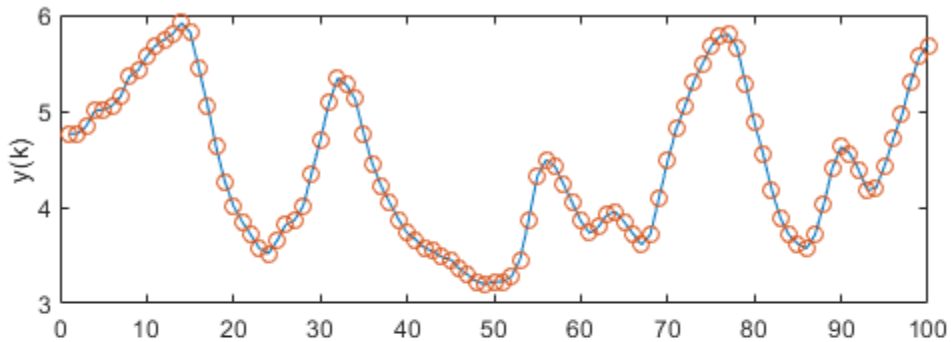
Here are the results of the test.

```
load drydemodata
data_n = length(y2);
output = y2;
input = [[0; y2(1:data_n-1)] ...
         [0; 0; y2(1:data_n-2)] ...
         [0; 0; 0; y2(1:data_n-3)] ...
         [0; 0; 0; 0; y2(1:data_n-4)] ...
         [0; u2(1:data_n-1)] ...
         [0; 0; u2(1:data_n-2)] ...
```

```

    [0; 0; 0; u2(1:data_n-3)] ...
    [0; 0; 0; 0; u2(1:data_n-4)] ...
    [0; 0; 0; 0; 0; u2(1:data_n-5)] ...
    [0; 0; 0; 0; 0; 0; u2(1:data_n-6)];
data = [input output];
data(1:6, :) = [];
input_name = char('y(k-1)', 'y(k-2)', 'y(k-3)', 'y(k-4)', 'u(k-1)', 'u(k-2)', 'u(k-3)', 'u(k-4)');
index = 1:100;
subplot(2,1,1)
plot(index,y2(index),'- ',index,y2(index),'o')
ylabel('y(k)', 'fontsize',10)
subplot(2,1,2)
plot(index,u2(index),'- ',index,u2(index),'o')
ylabel('u(k)', 'fontsize',10)

```



The data points were collected at a sampling time of 0.08 seconds. One thousand input-output data points were collected from the process as the input $u(k)$ was chosen to be a binary random signal shifting between 3.41 and 6.41 V. The probability of shifting the input at each sample was 0.2. The data set is available from the System Identification Toolbox, and the above plots show the output temperature $y(k)$ and input voltage $u(t)$ for the first 100 time steps.

ARX Model Identification

A conventional method is to remove the means from the data and assume a linear model of the form:

$$y(k)+a_1*y(k-1)+\dots+a_m*y(k-m)=b_1*u(k-d)+\dots+b_n*u(k-d-n+1)$$

where a_i ($i = 1$ to m) and b_j ($j = 1$ to n) are linear parameters to be determined by least-squares methods. This structure is called the ARX model and it is exactly specified by three integers $[m, n, d]$. To find an ARX model for the dryer device, the data set was divided into a training ($k = 1$ to 300) and a checking ($k = 301$ to 600) set. An exhaustive search was performed to find the best combination of $[m, n, d]$, where each of the integer is allowed to be changed from 1 to 10 independently. The best ARX model thus found is specified by $[m, n, d] = [5, 10, 2]$, with a training RMSE of 0.1122 and a checking RMSE of 0.0749. The above figure shows the fitting results of the best ARX model.

```
trn_data_n = 300;
total_data_n = 600;
z = [y2 u2];
z = dtrend(z);
ave = mean(y2);
ze = z(1:trn_data_n,:);
zv = z(trn_data_n+1:total_data_n,:);
T = 0.08;

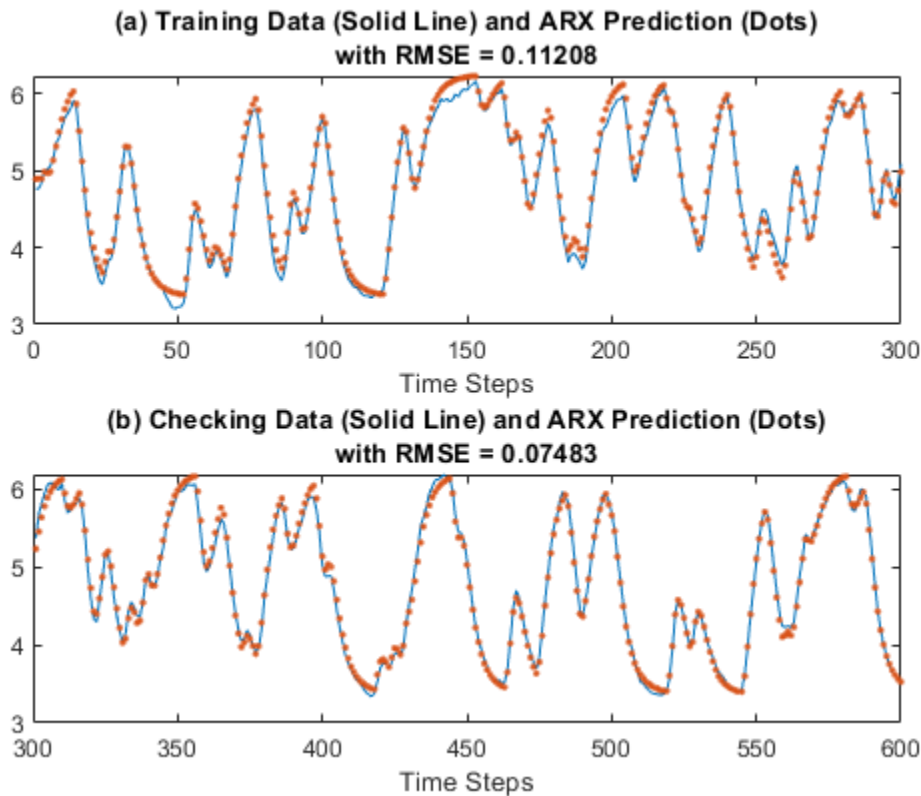
% Run through all different models
V = arxstruc(ze,zv,struc(1:10,1:10,1:10));
% Find the best model
nn = selstruc(V,0);
% Time domain plot
th = arx(ze,nn);
th.Ts = 0.08;
u = z(:,2);
y = z(:,1)+ave;
yp = sim(u,th)+ave;
```

```
xlabel = 'Time Steps';

subplot(2,1,1)
index = 1:trn_data_n;
plot(index, y(index), index, yp(index), '.')
rmse = norm(y(index)-yp(index))/sqrt(length(index));
title(sprintf(['(a) Training Data (Solid Line) and ARX Prediction (Dots)\nwith RMSE =
disp(['[na nb d] = ' num2str(nn)])
xlabel(xlbl, 'fontsize',10)

subplot(2,1,2)
index = (trn_data_n+1):(total_data_n);
plot(index,y(index),index,yp(index),'.')
rmse = norm(y(index)-yp(index))/sqrt(length(index));
title(sprintf(['(b) Checking Data (Solid Line) and ARX Prediction (Dots)\nwith RMSE =
xlabel(xlbl, 'fontsize',10)

[na nb d] = 5 10 2
```



ANFIS Model Identification

The ARX model is inherently linear and the most significant advantage is that we can perform model structure and parameter identification rapidly. The performance in the above plots appears to be satisfactory. However, if a better performance level is desired, we might want to resort to a nonlinear model. In particular, we are going to use a neuro-fuzzy modeling approach, ANFIS, to see if we can push the performance level with a fuzzy inference system.

To use ANFIS for system identification, the first thing we need to do is select the input. That is, to determine which variables should be the input arguments to an ANFIS model. For simplicity, we suppose that there are 10 input candidates ($y(k-1)$, $y(k-2)$, $y(k-3)$, $y(k-4)$, $u(k-1)$, $u(k-2)$, $u(k-3)$, $u(k-4)$, $u(k-5)$, $u(k-6)$), and the output to be predicted is $y(k)$. A

heuristic approach to input selection is called sequential forward search, in which each input is selected sequentially to optimize the total squared error. This can be done by the function `seqsrch`; the result is shown in the above plot, where 3 inputs ($y(k-1)$, $u(k-3)$, and $u(k-4)$) are selected with a training RMSE of 0.0609 and checking RMSE of 0.0604.

```
trn_data_n = 300;
trn_data = data(1:trn_data_n,:);
chk_data = data(trn_data_n+1:trn_data_n+300,:);
[~,elapsed_time] = seqsrch(3,trn_data,chk_data,input_name); % #ok<*ASGLU>
fprintf('\nElapsed time = %f\n',elapsed_time);
winH1 = gcf;
```

```
Selecting input 1 ...
ANFIS model 1: y(k-1) --> trn=0.2043, chk=0.1888
ANFIS model 2: y(k-2) --> trn=0.3819, chk=0.3541
ANFIS model 3: y(k-3) --> trn=0.5245, chk=0.4903
ANFIS model 4: y(k-4) --> trn=0.6308, chk=0.5977
ANFIS model 5: u(k-1) --> trn=0.8271, chk=0.8434
ANFIS model 6: u(k-2) --> trn=0.7976, chk=0.8087
ANFIS model 7: u(k-3) --> trn=0.7266, chk=0.7349
ANFIS model 8: u(k-4) --> trn=0.6215, chk=0.6346
ANFIS model 9: u(k-5) --> trn=0.5419, chk=0.5650
ANFIS model 10: u(k-6) --> trn=0.5304, chk=0.5601
Currently selected inputs: y(k-1)
```

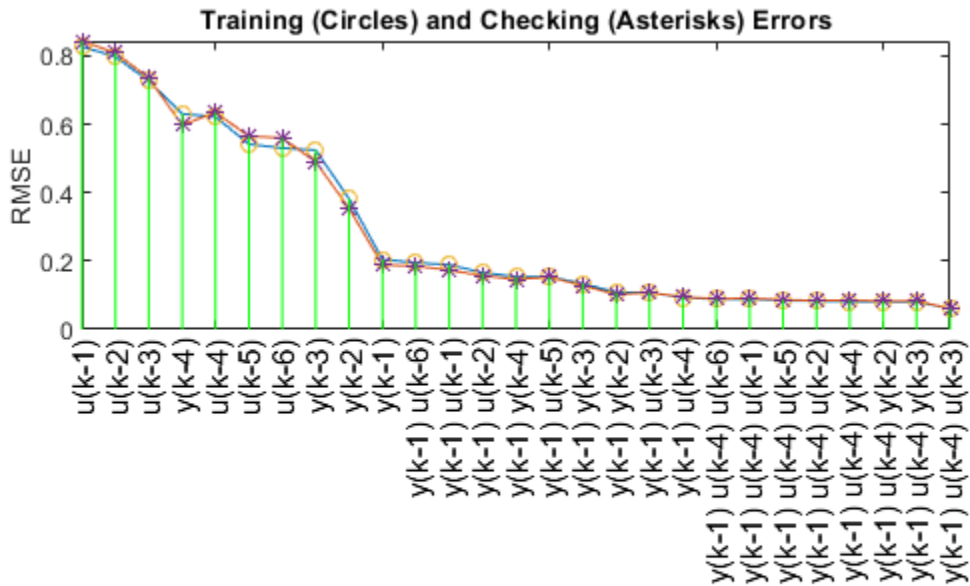
```
Selecting input 2 ...
ANFIS model 11: y(k-1) y(k-2) --> trn=0.1085, chk=0.1024
ANFIS model 12: y(k-1) y(k-3) --> trn=0.1339, chk=0.1283
ANFIS model 13: y(k-1) y(k-4) --> trn=0.1542, chk=0.1461
ANFIS model 14: y(k-1) u(k-1) --> trn=0.1892, chk=0.1734
ANFIS model 15: y(k-1) u(k-2) --> trn=0.1663, chk=0.1574
ANFIS model 16: y(k-1) u(k-3) --> trn=0.1082, chk=0.1077
ANFIS model 17: y(k-1) u(k-4) --> trn=0.0925, chk=0.0948
ANFIS model 18: y(k-1) u(k-5) --> trn=0.1533, chk=0.1531
ANFIS model 19: y(k-1) u(k-6) --> trn=0.1952, chk=0.1853
Currently selected inputs: y(k-1) u(k-4)
```

```
Selecting input 3 ...
ANFIS model 20: y(k-1) u(k-4) y(k-2) --> trn=0.0808, chk=0.0822
ANFIS model 21: y(k-1) u(k-4) y(k-3) --> trn=0.0806, chk=0.0836
ANFIS model 22: y(k-1) u(k-4) y(k-4) --> trn=0.0817, chk=0.0855
ANFIS model 23: y(k-1) u(k-4) u(k-1) --> trn=0.0886, chk=0.0912
ANFIS model 24: y(k-1) u(k-4) u(k-2) --> trn=0.0835, chk=0.0843
```

```

ANFIS model 25: y(k-1) u(k-4) u(k-3) --> trn=0.0609, chk=0.0604
ANFIS model 26: y(k-1) u(k-4) u(k-5) --> trn=0.0848, chk=0.0867
ANFIS model 27: y(k-1) u(k-4) u(k-6) --> trn=0.0890, chk=0.0894
Currently selected inputs: y(k-1) u(k-3) u(k-4)
    
```

Elapsed time = 17.427000



For input selection, another more computationally intensive approach is to do an exhaustive search on all possible combinations of the input candidates. The function that performs exhaustive search is `exhsrch`, which selects 3 inputs from 10 candidates. However, `exhsrch` usually involves a significant amount of computation if all combinations are tried. For instance, if 3 is selected out of 10, the total number of ANFIS models is $C(10, 3) = 120$.

Fortunately, for dynamic system identification, we do know that the inputs should not come from either of the following two sets of input candidates exclusively:

$$Y = \{y(k-1), y(k-2), y(k-3), y(k-4)\}$$

$$U = \{u(k-1), u(k-2), u(k-3), u(k-4), u(k-5), u(k-6)\}$$

A reasonable guess would be to take two inputs from Y and one from U to form the inputs to ANFIS; the total number of ANFIS models is then $C(4,2)*6=36$, which is much less. The above plot shows that the selected inputs are $y(k-1)$, $y(k-2)$ and $u(k-3)$, with a training RMSE of 0.0474 and checking RMSE of 0.0485, which are better than ARX models and ANFIS via sequential forward search.

```
group1 = [1 2 3 4];    % y(k-1), y(k-2), y(k-3), y(k-4)
group2 = [1 2 3 4];    % y(k-1), y(k-2), y(k-3), y(k-4)
group3 = [5 6 7 8 9 10]; % u(k-1) through y(k-6)

anfis_n = 6*length(group3);
index = zeros(anfis_n,3);
trn_error = zeros(anfis_n,1);
chk_error = zeros(anfis_n,1);
% ===== Training options
% Create option set for generating initial FIS.
genOpt = genfisOptions('GridPartition','NumMembershipFunctions',2, ...
    'InputMembershipFunctionType','gbellmf');
% Create option set for |anfis| command and set options that remain constant
% for different training scenarios.
anfisOpt = anfisOptions('EpochNumber',1,...
    'InitialStepSize',0.1,...
    'StepSizeDecreaseRate',0.5,...
    'StepSizeIncreaseRate',1.5,...
    'DisplayANFISInformation',0,...
    'DisplayErrorValues',0,...
    'DisplayStepSize',0,...
    'DisplayFinalResults',0);
% ===== Train ANFIS with different input variables
fprintf('\nTrain %d ANFIS models, each with 3 inputs selected from 10 candidates...\n',
    anfis_n);
model = 1;
for i = 1:length(group1)
    for j = i+1:length(group2)
        for k = 1:length(group3)
            in1 = deblank(input_name(group1(i),:));
            in2 = deblank(input_name(group2(j),:));
```

```

        in3 = deblank(input_name(group3(k),:));
        index(model, :) = [group1(i) group2(j) group3(k)];
        trn_data = data(1:trn_data_n, [group1(i) group2(j) group3(k) size(data,2)]);
        chk_data = data(trn_data_n+1:trn_data_n+300, [group1(i) group2(j) group3(k) size(data,2)]);
        in_fismat = genfis(trn_data(:,1:end-1),trn_data(:,end),genOpt);
        % Set initial FIS and validation data in option set for ANFIS training.
        anfisOpt.InitialFIS = in_fismat;
        anfisOpt.ValidationData = chk_data;
        [~, t_err, ~, ~, c_err] = anfis(trn_data,anfisOpt);
        trn_error(model) = min(t_err);
        chk_error(model) = min(c_err);
        fprintf('ANFIS model = %d: %s %s %s',model,in1,in2,in3);
        fprintf(' --> trn=%.4f',trn_error(model));
        fprintf(' chk=%.4f',chk_error(model));
        fprintf('\n');
        model = model+1;
    end
end
end

% ===== Reordering according to training error
[~, b] = sort(trn_error);
b = flipud(b); % List according to decreasing trn error
trn_error = trn_error(b);
chk_error = chk_error(b);
index = index(b,:);

% ===== Display training and checking errors
x = (1:anfis_n)';
subplot(2,1,1)
plot(x, trn_error, '-o', x, chk_error, '-*', ...
     x, trn_error, 'o', x, chk_error, '*')
tmp = x(:,ones(1,3))';
X = tmp(:);
tmp = [zeros(anfis_n,1) max(trn_error,chk_error) nan*ones(anfis_n,1)]';
Y = tmp(:);
hold on
plot(X,Y,'g')
hold off
axis([1 anfis_n -inf inf])
h_gca = gca;
h_gca.XTickLabel = [];

% ===== Add text of input variables

```

```

for k = 1:anfis_n
    text(x(k),  $\bar{\theta}$ , ...
         [input_name(index(k,1),:) ' ' ...
          input_name(index(k,2),:) ' ' ...
          input_name(index(k,3),:)]);
end
h = findobj(gcf, 'type', 'text');
set(h, 'rot', 90, 'fontsize', 11, 'hori', 'right');

drawnow

% ===== Generate input_index for bjtrain.m
[a, b] = min(trn_error);
input_index = index(b,:);
title('Training (Circles) and Checking (Asterisks) Errors', 'fontsize', 10)
ylabel('RMSE', 'fontsize', 10)

```

Train 36 ANFIS models, each with 3 inputs selected from 10 candidates...

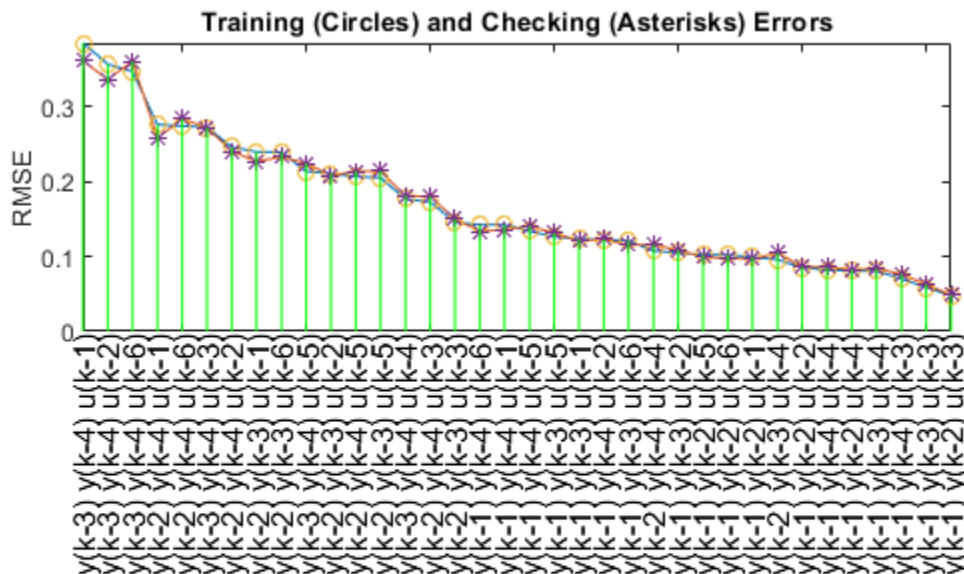
```

ANFIS model = 1: y(k-1) y(k-2) u(k-1) --> trn=0.0990, chk=0.0962
ANFIS model = 2: y(k-1) y(k-2) u(k-2) --> trn=0.0852, chk=0.0862
ANFIS model = 3: y(k-1) y(k-2) u(k-3) --> trn=0.0474, chk=0.0485
ANFIS model = 4: y(k-1) y(k-2) u(k-4) --> trn=0.0808, chk=0.0822
ANFIS model = 5: y(k-1) y(k-2) u(k-5) --> trn=0.1023, chk=0.0991
ANFIS model = 6: y(k-1) y(k-2) u(k-6) --> trn=0.1021, chk=0.0974
ANFIS model = 7: y(k-1) y(k-3) u(k-1) --> trn=0.1231, chk=0.1206
ANFIS model = 8: y(k-1) y(k-3) u(k-2) --> trn=0.1047, chk=0.1085
ANFIS model = 9: y(k-1) y(k-3) u(k-3) --> trn=0.0587, chk=0.0626
ANFIS model = 10: y(k-1) y(k-3) u(k-4) --> trn=0.0806, chk=0.0836
ANFIS model = 11: y(k-1) y(k-3) u(k-5) --> trn=0.1261, chk=0.1311
ANFIS model = 12: y(k-1) y(k-3) u(k-6) --> trn=0.1210, chk=0.1151
ANFIS model = 13: y(k-1) y(k-4) u(k-1) --> trn=0.1420, chk=0.1353
ANFIS model = 14: y(k-1) y(k-4) u(k-2) --> trn=0.1224, chk=0.1229
ANFIS model = 15: y(k-1) y(k-4) u(k-3) --> trn=0.0700, chk=0.0765
ANFIS model = 16: y(k-1) y(k-4) u(k-4) --> trn=0.0817, chk=0.0855
ANFIS model = 17: y(k-1) y(k-4) u(k-5) --> trn=0.1337, chk=0.1405
ANFIS model = 18: y(k-1) y(k-4) u(k-6) --> trn=0.1421, chk=0.1333
ANFIS model = 19: y(k-2) y(k-3) u(k-1) --> trn=0.2393, chk=0.2264
ANFIS model = 20: y(k-2) y(k-3) u(k-2) --> trn=0.2104, chk=0.2077
ANFIS model = 21: y(k-2) y(k-3) u(k-3) --> trn=0.1452, chk=0.1497
ANFIS model = 22: y(k-2) y(k-3) u(k-4) --> trn=0.0958, chk=0.1047
ANFIS model = 23: y(k-2) y(k-3) u(k-5) --> trn=0.2048, chk=0.2135
ANFIS model = 24: y(k-2) y(k-3) u(k-6) --> trn=0.2388, chk=0.2326
ANFIS model = 25: y(k-2) y(k-4) u(k-1) --> trn=0.2756, chk=0.2574

```

```

ANFIS model = 26: y(k-2) y(k-4) u(k-2) --> trn=0.2455, chk=0.2400
ANFIS model = 27: y(k-2) y(k-4) u(k-3) --> trn=0.1726, chk=0.1797
ANFIS model = 28: y(k-2) y(k-4) u(k-4) --> trn=0.1074, chk=0.1157
ANFIS model = 29: y(k-2) y(k-4) u(k-5) --> trn=0.2061, chk=0.2133
ANFIS model = 30: y(k-2) y(k-4) u(k-6) --> trn=0.2737, chk=0.2836
ANFIS model = 31: y(k-3) y(k-4) u(k-1) --> trn=0.3842, chk=0.3605
ANFIS model = 32: y(k-3) y(k-4) u(k-2) --> trn=0.3561, chk=0.3358
ANFIS model = 33: y(k-3) y(k-4) u(k-3) --> trn=0.2719, chk=0.2714
ANFIS model = 34: y(k-3) y(k-4) u(k-4) --> trn=0.1763, chk=0.1808
ANFIS model = 35: y(k-3) y(k-4) u(k-5) --> trn=0.2132, chk=0.2240
ANFIS model = 36: y(k-3) y(k-4) u(k-6) --> trn=0.3460, chk=0.3601
    
```



This window shows ANFIS predictions on both training and checking data sets. Obviously the performance is better than those of the ARX model.

```

if ishghandle(winH1),delete(winH1);
end

trn_data = data(1:trn_data_n,[input_index, size(data,2)]);
chk_data = data(trn_data_n+1:600,[input_index, size(data,2)]);

% generate FIS matrix
in_fismat = genfis(trn_data(:,1:end-1),trn_data(:,end));
anfisOpt = anfisOptions('InitialFIS',in_fismat,...
    'EpochNumber',1,...
    'InitialStepSize',0.01,...
    'StepSizeDecreaseRate',0.5,...
    'StepSizeIncreaseRate',1.5,...
    'ValidationData',chk_data);
[trn_out_fismat,trn_error,step_size,chk_out_fismat,chk_error] = ...
    anfis(trn_data,anfisOpt);

subplot(2,1,1)
index = 1:trn_data_n;
plot(index,y(index),index,yp(index),'.')
rmse = norm(y(index)-yp(index))/sqrt(length(index));
title(sprintf(['(a) Training Data (Solid Line) and ARX Prediction (Dots)\nwith RMSE =
disp(['[na nb d] = ' num2str(nn)])
xlabel('Time Steps','fontsize',10)
subplot(2,1,2)
index = (trn_data_n+1):(total_data_n);
plot(index, y(index),index,yp(index),'.')
rmse = norm(y(index)-yp(index))/sqrt(length(index));
title(sprintf(['(b) Checking Data (Solid Line) and ARX Prediction (Dots)\nwith RMSE =
xlabel('Time Steps','fontsize',10)

ANFIS info:
    Number of nodes: 34
    Number of linear parameters: 32
    Number of nonlinear parameters: 18
    Total number of parameters: 50
    Number of training data pairs: 300
    Number of checking data pairs: 300
    Number of fuzzy rules: 8

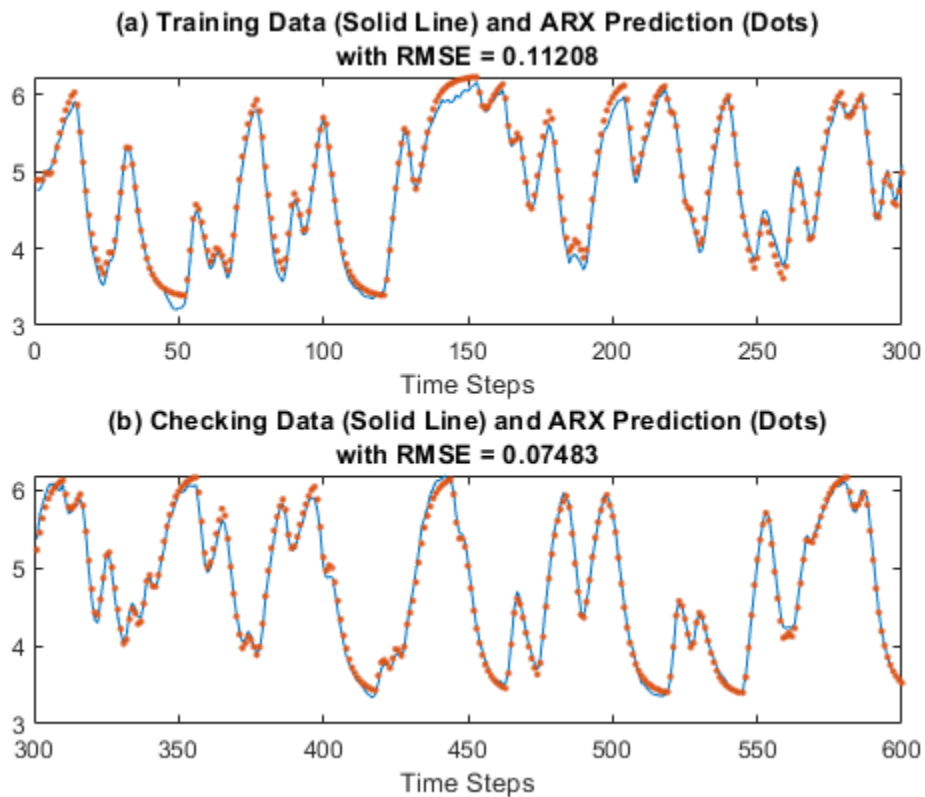
Start training ANFIS ...

    1         0.0474113         0.0485325

```

Designated epoch number reached --> ANFIS training completed at epoch 1.

Minimal training RMSE = 0.047411
 Minimal checking RMSE = 0.0485325
 [na nb d] = 5 10 2



```
y_hat = evalfis(chk_out_fismat,data(1:600,input_index));

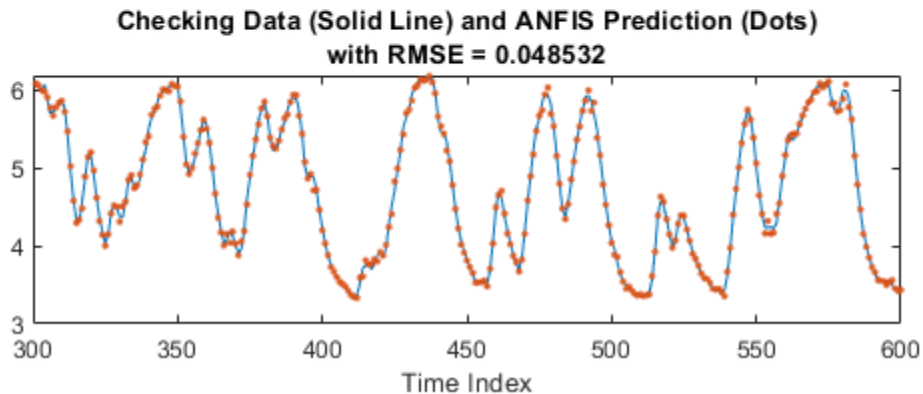
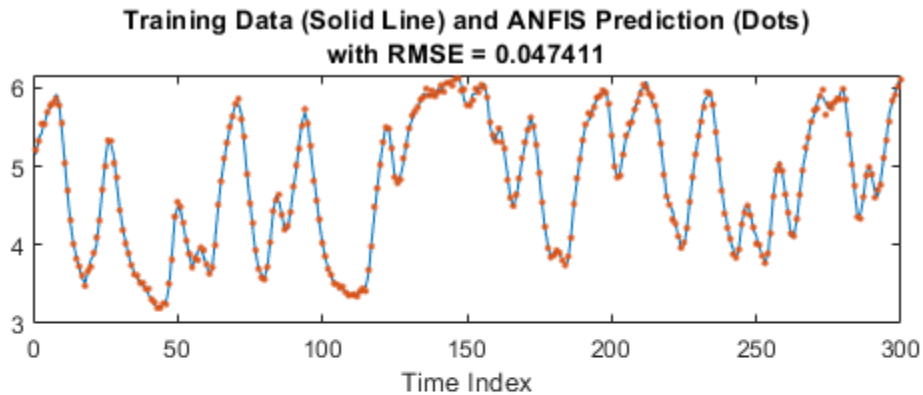
subplot(2,1,1)
index = 1:trn_data_n;
plot(index,data(index,size(data,2)),'-','...
      index,y_hat(index),'.','')
rmse = norm(y_hat(index)-data(index,size(data,2)))/sqrt(length(index));
```

```

title(sprintf(['Training Data (Solid Line) and ANFIS Prediction (Dots)\nwith RMSE = '
xlabel('Time Index','fontsize',10)
ylabel('')

subplot(2,1,2)
index = trn_data_n+1:600;
plot(index,data(index,size(data,2)),'-',index,y_hat(index),'.')
rmse = norm(y_hat(index)-data(index,size(data,2)))/sqrt(length(index));
title(sprintf(['Checking Data (Solid Line) and ANFIS Prediction (Dots)\nwith RMSE = '
xlabel('Time Index','fontsize',10)
ylabel('')

```



Conclusion

The table above is a comparison among various modeling approaches. The ARX modeling spends the least amount of time to reach the worst precision, and the ANFIS modeling via exhaustive search takes the most amount of time to reach the best precision. In other words, if fast modeling is the goal, then ARX is the right choice. But if precision is the utmost concern, then we should go with ANFIS, which is designed for nonlinear modeling and higher precision.

Models	ARX Model	ANFIS Model (Sequential Search)	ANFIS Model (Exhaustive Search)
No. of Input Arguments	14	3	3
Training RMSE	0.1122	0.0609	0.0474
Checking RMSE	0.0749	0.0604	0.0485
No. of Linear Parameters	15	32	32
No. of Nonlinear Parameters	0	18	18
Computation Time (Pentium-Pro 200 MHz, 64 MB RAM)	1.6 S	10.3 S	21.5 S

See Also

`anfis` | `evalfis` | `genfis`

More About

- “Neuro-Adaptive Learning and ANFIS” on page 3-2

Gas Mileage Prediction

This example shows how to predict of fuel consumption (miles per gallon) for automobiles, using data from previously recorded observations.

Introduction

Automobile MPG (miles per gallon) prediction is a typical nonlinear regression problem, in which several attributes of an automobile's profile information are used to predict another continuous attribute, the fuel consumption in MPG. The training data is available in the UCI (Univ. of California at Irvine) Machine Learning Repository at <http://www.ics.uci.edu/~mllearn/MLRepository.html>. It contains data collected from automobiles of various makes and models.

The table shown above is several observations or samples from the MPG data set. The six input attributes are no. of cylinders, displacement, horsepower, weight, acceleration, and model year. The output variable to be predicted is the fuel consumption in MPG. (The automobile's manufacturers and models in the first column of the table are not used for prediction).

Partitioning Data

The data set is obtained from the original data file 'auto-gas.dat'. The dataset is then partitioned into a training set (odd-indexed samples) and a checking set (even-indexed samples).

```
[data,input_name] = loadgas;  
trn_data = data(1:2:end,:);  
chk_data = data(2:2:end,:);
```

Input Selection

The function `exhsrch` performs an exhaustive search within the available inputs to select the set of inputs that most influence the fuel consumption. The first parameter to the function specifies the number of input combinations to be tried during the search. Essentially, `exhsrch` builds an ANFIS model for each combination and trains it for one epoch and reports the performance achieved. In the following example, `exhsrch` is used to determine the one most influential input attribute in predicting the output.

```
exhsrch(1,trn_data,chk_data,input_name);
```

Train 6 ANFIS models, each with 1 inputs selected from 6 candidates...

ANFIS model 1: Cylinder --> trn=4.6400, chk=4.7255
ANFIS model 2: Disp --> trn=4.3106, chk=4.4316
ANFIS model 3: Power --> trn=4.5399, chk=4.1713
ANFIS model 4: Weight --> trn=4.2577, chk=4.0863
ANFIS model 5: Acceler --> trn=6.9789, chk=6.9317
ANFIS model 6: Year --> trn=6.2255, chk=6.1693

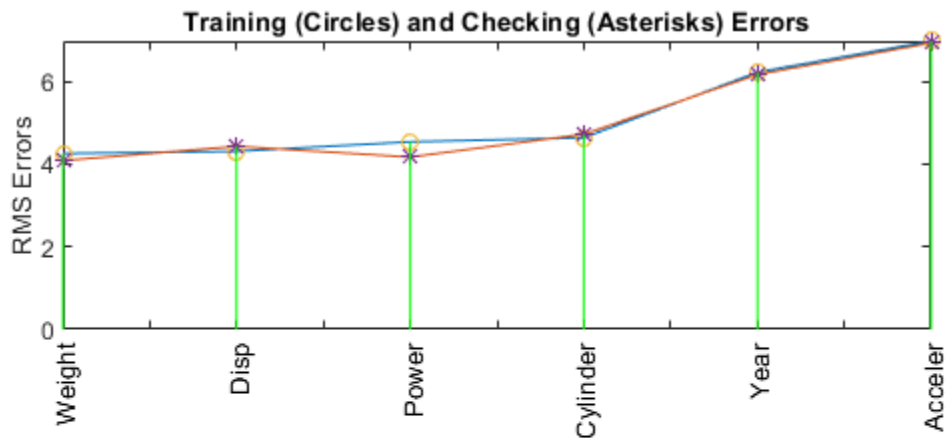


Figure 1: Every input variable's influence on fuel consumption

The left-most input variable in Figure 1 has the least error or in other words the most relevance with respect to the output.

The plot and results from the function clearly indicate that the input attribute 'Weight' is the most influential. The training and checking errors are comparable, which implies that

there is no overfitting. This means we can push a little further and explore if we can select more than one input attribute to build the ANFIS model.

Intuitively, we can simply select `Weight` and `Disp` directly since they have the least errors as shown in the plot. However, this will not necessarily be the optimal combination of two inputs that result in the minimal training error. To verify this, we can use `exhsrch` to search for the optimal combination of 2 input attributes.

```
input_index = exhsrch(2,trn_data,chk_data,input_name);
```

```
Train 15 ANFIS models, each with 2 inputs selected from 6 candidates...
```

```
ANFIS model 1: Cylinder Disp --> trn=3.9320, chk=4.7920
ANFIS model 2: Cylinder Power --> trn=3.7364, chk=4.8683
ANFIS model 3: Cylinder Weight --> trn=3.8741, chk=4.6763
ANFIS model 4: Cylinder Acceler --> trn=4.3287, chk=5.9625
ANFIS model 5: Cylinder Year --> trn=3.7129, chk=4.5946
ANFIS model 6: Disp Power --> trn=3.8087, chk=3.8594
ANFIS model 7: Disp Weight --> trn=4.0271, chk=4.6350
ANFIS model 8: Disp Acceler --> trn=4.0782, chk=4.4890
ANFIS model 9: Disp Year --> trn=2.9565, chk=3.3905
ANFIS model 10: Power Weight --> trn=3.9310, chk=4.2976
ANFIS model 11: Power Acceler --> trn=4.2740, chk=3.8738
ANFIS model 12: Power Year --> trn=3.3796, chk=3.3505
ANFIS model 13: Weight Acceler --> trn=4.0875, chk=4.0095
ANFIS model 14: Weight Year --> trn=2.7657, chk=2.9953
ANFIS model 15: Acceler Year --> trn=5.6242, chk=5.6481
```

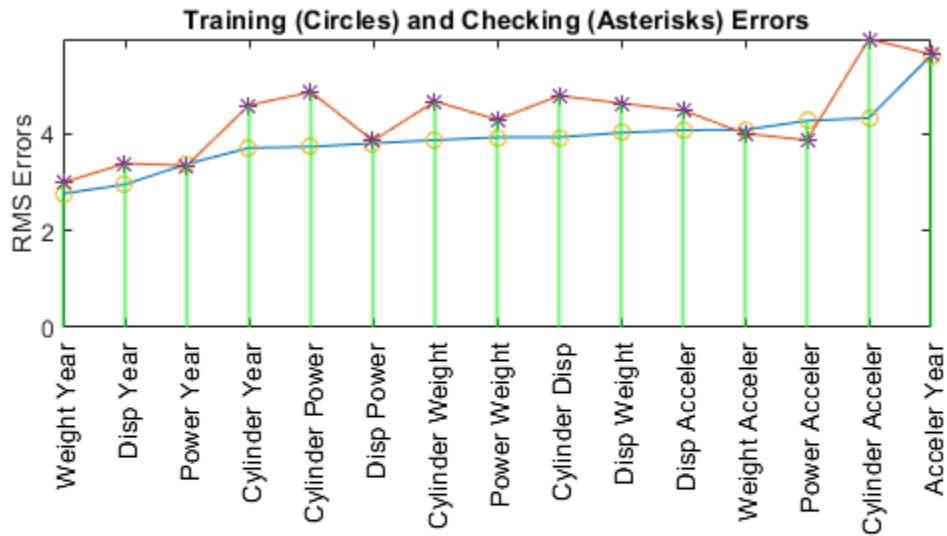


Figure 2: All two input variable combinations and their influence on fuel consumption

The results from `exhsrch` indicate that `Weight` and `Year` form the optimal combination of two input attributes. The training and checking errors are getting distinguished, indicating the outset of overfitting. It may not be prudent to use more than two inputs for building the ANFIS model. We can test this premise to verify its validity.

```
exhsrch(3, trn_data, chk_data, input_name);
```

Train 20 ANFIS models, each with 3 inputs selected from 6 candidates...

```
ANFIS model 1: Cylinder Disp Power --> trn=3.4446, chk=11.5329
ANFIS model 2: Cylinder Disp Weight --> trn=3.6686, chk=4.8922
ANFIS model 3: Cylinder Disp Acceler --> trn=3.6610, chk=5.2384
```

ANFIS model 4: Cylinder Disp Year --> trn=2.5463, chk=4.9001
ANFIS model 5: Cylinder Power Weight --> trn=3.4797, chk=9.3761
ANFIS model 6: Cylinder Power Acceler --> trn=3.5432, chk=4.4804
ANFIS model 7: Cylinder Power Year --> trn=2.6300, chk=3.6300
ANFIS model 8: Cylinder Weight Acceler --> trn=3.5708, chk=4.8379
ANFIS model 9: Cylinder Weight Year --> trn=2.4951, chk=4.0435
ANFIS model 10: Cylinder Acceler Year --> trn=3.2698, chk=6.2616
ANFIS model 11: Disp Power Weight --> trn=3.5879, chk=7.4942
ANFIS model 12: Disp Power Acceler --> trn=3.5395, chk=3.9953
ANFIS model 13: Disp Power Year --> trn=2.4607, chk=3.3563
ANFIS model 14: Disp Weight Acceler --> trn=3.6075, chk=4.2318
ANFIS model 15: Disp Weight Year --> trn=2.5617, chk=3.7866
ANFIS model 16: Disp Acceler Year --> trn=2.4149, chk=3.2480
ANFIS model 17: Power Weight Acceler --> trn=3.7884, chk=4.0480
ANFIS model 18: Power Weight Year --> trn=2.4371, chk=3.2852
ANFIS model 19: Power Acceler Year --> trn=2.7276, chk=3.2580
ANFIS model 20: Weight Acceler Year --> trn=2.3603, chk=2.9152

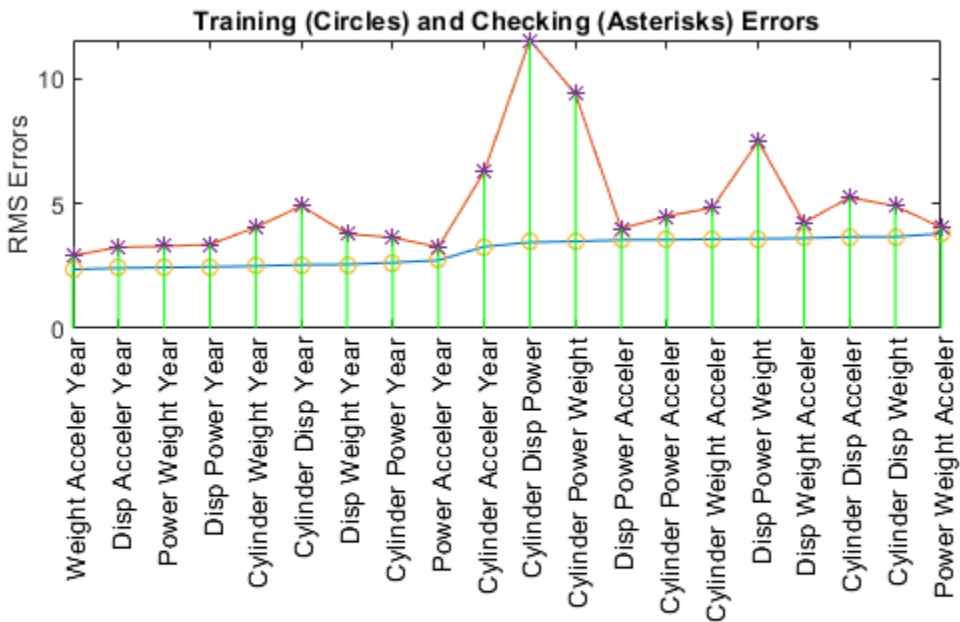


Figure 3: All three input variable combinations and their influence on fuel consumption

The plot shows the result of selecting three inputs, in which `Weight`, `Year`, and `Acceler` are selected as the best combination of three input variables. However, the minimal training (and checking) error do not reduce significantly from that of the best two-input model, which indicates that the newly added attribute `Acceler` does not improve the prediction much. For better generalization, we always prefer a model with a simple structure. Therefore we will stick to the two-input ANFIS for further exploration.

We then extract the selected input attributes from the original training and checking datasets.

```
close all;
new_trn_data = trn_data(:,[input_index, size(trn_data,2)]);
new_chk_data = chk_data(:,[input_index, size(chk_data,2)]);
```

Training ANFIS Model

The function `exhsrch` only trains each ANFIS for a single epoch in order to be able to quickly find the right inputs. Now that the inputs are fixed, we can spend more time on ANFIS training (100 epochs).

The `genfis` function generates a initial FIS from the training data, which is then fine-tuned by ANFIS to generate the final model.

```
in_fismat = genfis(new_trn_data(:,1:end-1),new_trn_data(:,end));
anfisOpt = anfisOptions('InitialFIS',in_fismat,'EpochNumber',100,...
    'StepSizeDecreaseRate',0.5,...
    'StepSizeIncreaseRate',1.5,...
    'ValidationData',new_chk_data,...
    'DisplayANFISInformation',0,...
    'DisplayErrorValues',0,...
    'DisplayStepSize',0,...
    'DisplayFinalResults',0);
[trn_out_fismat,trn_error,step_size,chk_out_fismat,chk_error] = ...
    anfis(new_trn_data,anfisOpt);
```

ANFIS returns the error with respect to training data and checking data in the list of its output parameters. The plot of the errors provides useful information about the training process.

```
[a,b] = min(chk_error);
plot(1:100,trn_error,'g-',1:100,chk_error,'r-',b,a,'ko')
title('Training (green) and checking (red) error curve','fontsize',10)
xlabel('Epoch numbers','fontsize',10)
ylabel('RMS errors','fontsize',10)
```

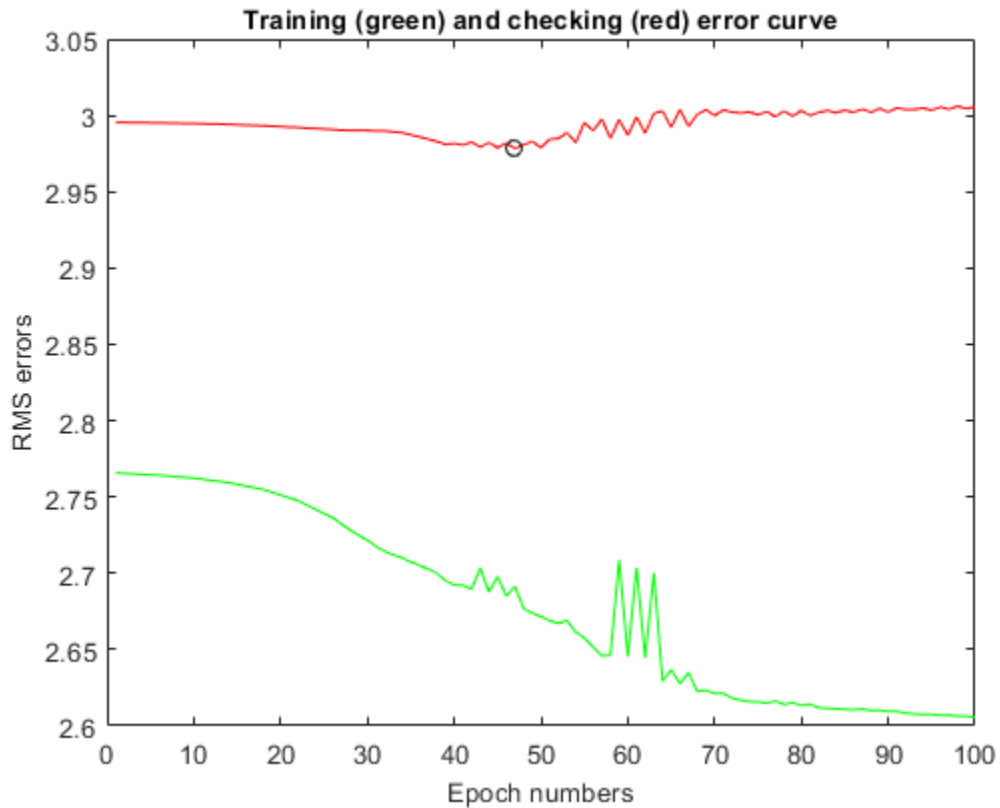


Figure 4: ANFIS training and checking errors

The plot above shows the error curves for 100 epochs of ANFIS training. The green curve gives the training errors and the red curve gives the checking errors. The minimal checking error occurs at about epoch 45, which is indicated by a circle. Notice that the checking error curve goes up after 50 epochs, indicating that further training over fits the data and produces worse generalization

ANFIS vs Linear Regression

A good exercise at this point would be to check the performance of the ANFIS model with a linear regression model.

The ANFIS prediction can be compared against a linear regression model by comparing their respective RMSE (Root mean square) values against checking data.

```
% Performing Linear Regression
N = size(trn_data,1);
A = [trn_data(:,1:6) ones(N,1)];
B = trn_data(:,7);
coef = A\B; % Solving for regression parameters from training data

Nc = size(chk_data,1);
A_ck = [chk_data(:,1:6) ones(Nc,1)];
B_ck = chk_data(:,7);
lr_rmse = norm(A_ck*coef-B_ck)/sqrt(Nc);
% Printing results
fprintf('\nRMSE against checking data\nANFIS : %1.3f\tLinear Regression : %1.3f\n',a,lr_rmse);

RMSE against checking data
ANFIS : 2.978    Linear Regression : 3.444
```

It can be seen that the ANFIS model outperforms the linear regression model.

Analyzing ANFIS Model

The variable `chk_out_fismat` represents the snapshot of the ANFIS model at the minimal checking error during the training process. The input-output surface of the model is shown in the plot below.

```
chk_out_fismat.Inputs(1).Name = "Weight";
chk_out_fismat.Inputs(2).Name = "Year";
chk_out_fismat.Outputs(1).Name = "MPG";

% Generating the FIS output surface plot
gensurf(chk_out_fismat);
```

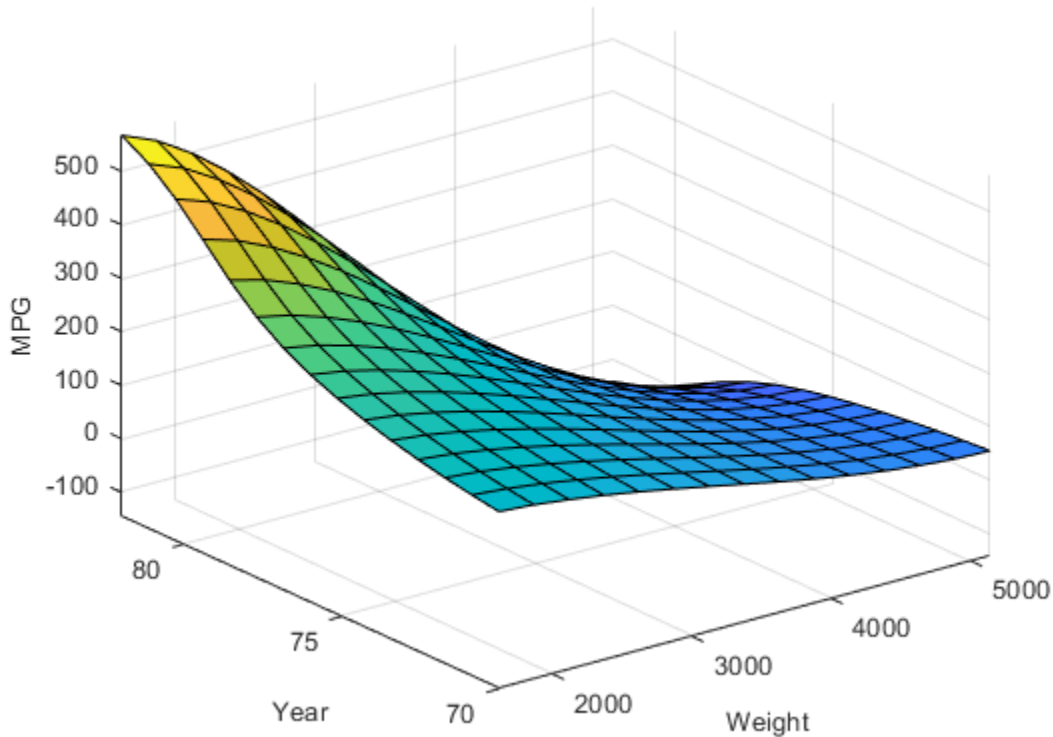


Figure 5: Input-Output surface for trained FIS

The input-output surface shown above is a nonlinear and monotonic surface and illustrates how the ANFIS model will respond to varying values of 'weight' and 'year'.

Limitations and Cautions

We can see some spurious effects at the far-end corner of the surface. The elevated corner says that the heavier an automobile is, the more gas-efficient it will be. This is totally counter-intuitive, and it is a direct result from lack of data.

```
plot(new_trn_data(:,1),new_trn_data(:, 2),'bo', ...
     new_chk_data(:,1),new_chk_data(:, 2),'rx')
xlabel('Weight','fontsize',10)
```

```
ylabel('Year','fontsize',10)
title('Training (o) and checking (x) data','fontsize',10)
```

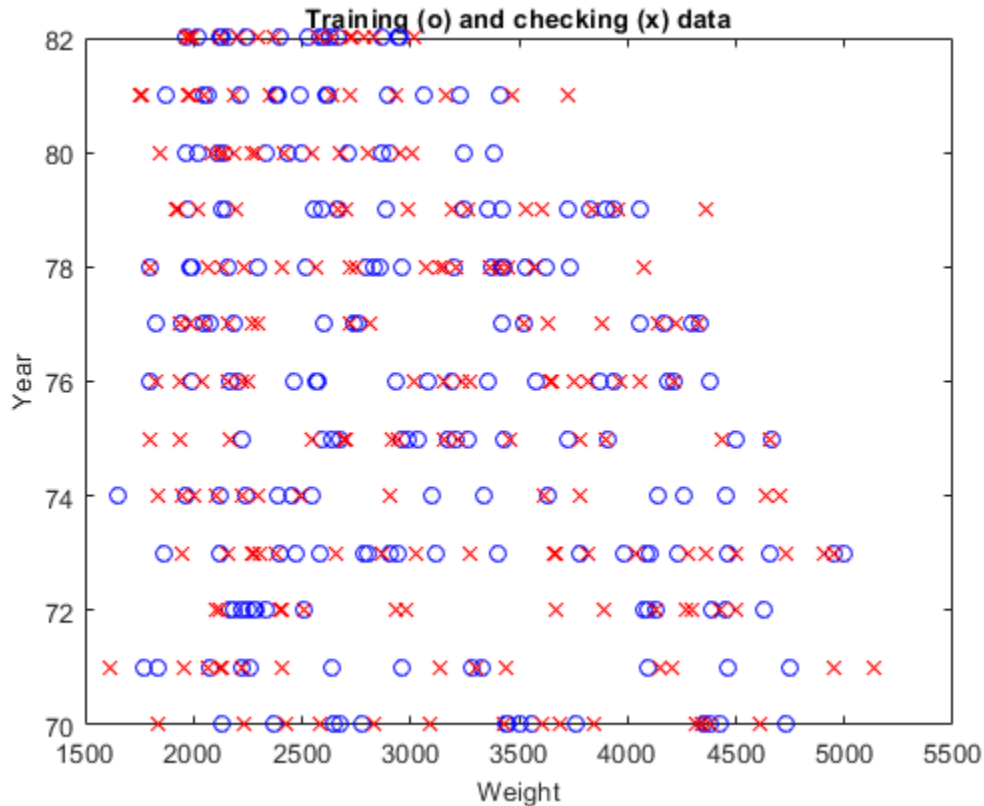


Figure 6: Weight vs Year plot showing lack of data in the upper-right corner

This plot above shows the data distribution. The lack of training data at the upper right corner causes the spurious ANFIS surface mentioned earlier. Therefore the prediction by ANFIS should always be interpreted with the data distribution in mind.

See Also

anfis | evalfis | genfis

More About

- “Neuro-Adaptive Learning and ANFIS” on page 3-2

Data Clustering

- “Fuzzy Clustering” on page 4-2
- “Cluster Quasi-Random Data Using Fuzzy C-Means Clustering” on page 4-4
- “Adjust Fuzzy Overlap in Fuzzy C-Means Clustering” on page 4-8
- “Fuzzy C-Means Clustering” on page 4-12
- “Fuzzy C-Means Clustering for Iris Data” on page 4-16
- “Model Suburban Commuting Using Subtractive Clustering” on page 4-21
- “Modeling Traffic Patterns using Subtractive Clustering” on page 4-33
- “Data Clustering Using Clustering Tool” on page 4-47

Fuzzy Clustering

What Is Data Clustering?

Clustering of numerical data forms the basis of many classification and system modeling algorithms. The purpose of clustering is to identify natural groupings of data from a large data set to produce a concise representation of a system's behavior.

Fuzzy Logic Toolbox tools allow you to find clusters in input-output training data. You can use the cluster information to generate a Sugeno-type fuzzy inference system that best models the data behavior using a minimum number of rules. The rules partition themselves according to the fuzzy qualities associated with each of the data clusters. To automatically generate this type of FIS, use the `genfis` command.

Fuzzy C-Means Clustering

Fuzzy c-means (FCM) is a data clustering technique wherein each data point belongs to a cluster to some degree that is specified by a membership grade. This technique was originally introduced by Jim Bezdek in 1981 [1] as an improvement on earlier clustering methods. It provides a method that shows how to group data points that populate some multidimensional space into a specific number of different clusters.

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

The command line function `fcm` outputs a list of cluster centers and several membership grades for each data point. You can use the information returned by `fcm` to help you build a fuzzy inference system by creating membership functions to represent the fuzzy qualities of each cluster. To generate a Sugeno-type fuzzy inference system that models the behavior of input/output data, you can configure the `genfis` command to use FCM clustering.

Subtractive Clustering

If you do not 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 for a set of data [2]. The cluster estimates, which are obtained from the `subclust` function, can be used to initialize iterative optimization-based clustering methods (`fcm`) and model identification methods (`likeanfis`). The `subclust` function finds the clusters using the subtractive clustering method.

To generate a Sugeno-type fuzzy inference system that models the behavior of input/output data, you can configure the `genfis` command to use subtractive clustering.

References

- [1] Bezdek, J.C., *Pattern Recognition with Fuzzy Objective Function Algorithms*, Plenum Press, New York, 1981.
- [2] Chiu, S., "Fuzzy Model Identification Based on Cluster Estimation," *Journal of Intelligent & Fuzzy Systems*, Vol. 2, No. 3, Sept. 1994.

See Also

`fcm` | `genfis` | `subclust`

More About

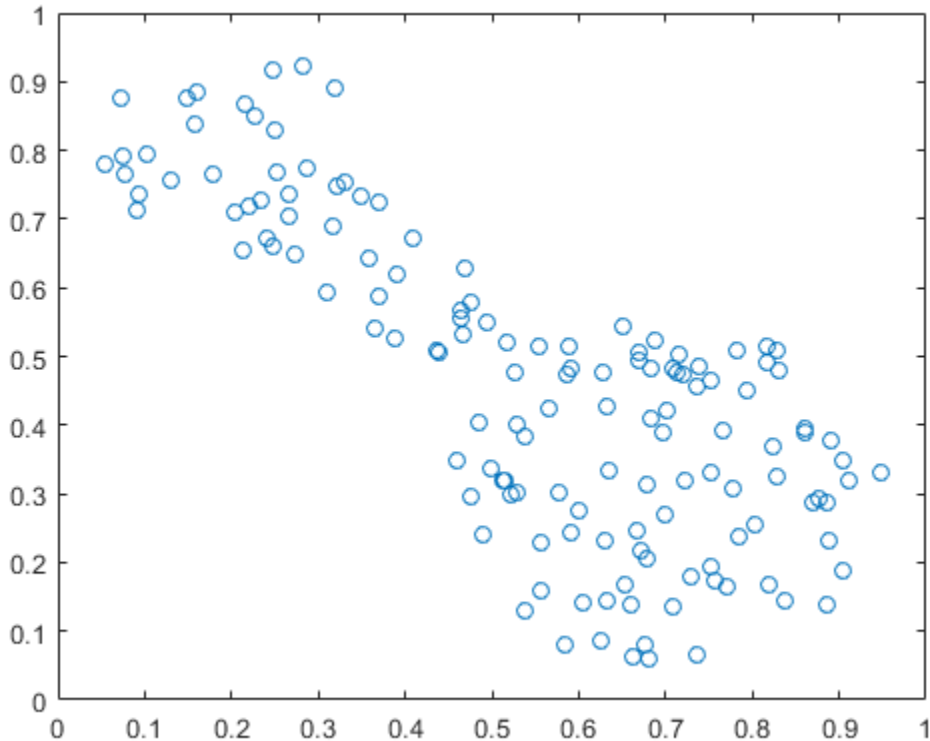
- "Cluster Quasi-Random Data Using Fuzzy C-Means Clustering" on page 4-4
- "Model Suburban Commuting Using Subtractive Clustering" on page 4-21
- "Data Clustering Using Clustering Tool" on page 4-47

Cluster Quasi-Random Data Using Fuzzy C-Means Clustering

This example shows how FCM clustering works using quasi-random two-dimensional data.

Load the data set and plot it.

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



Next, invoke the command-line function, `fcm`, to find two clusters in this data set until the objective function is no longer decreasing much at all.


```
[center,U,objFcn] = fcm(fcndata,2);  
  
Iteration count = 1, obj. fcn = 8.970479  
Iteration count = 2, obj. fcn = 7.197402  
Iteration count = 3, obj. fcn = 6.325579  
Iteration count = 4, obj. fcn = 4.586142  
Iteration count = 5, obj. fcn = 3.893114  
Iteration count = 6, obj. fcn = 3.810804  
Iteration count = 7, obj. fcn = 3.799801  
Iteration count = 8, obj. fcn = 3.797862  
Iteration count = 9, obj. fcn = 3.797508  
Iteration count = 10, obj. fcn = 3.797444  
Iteration count = 11, obj. fcn = 3.797432  
Iteration count = 12, obj. fcn = 3.797430
```

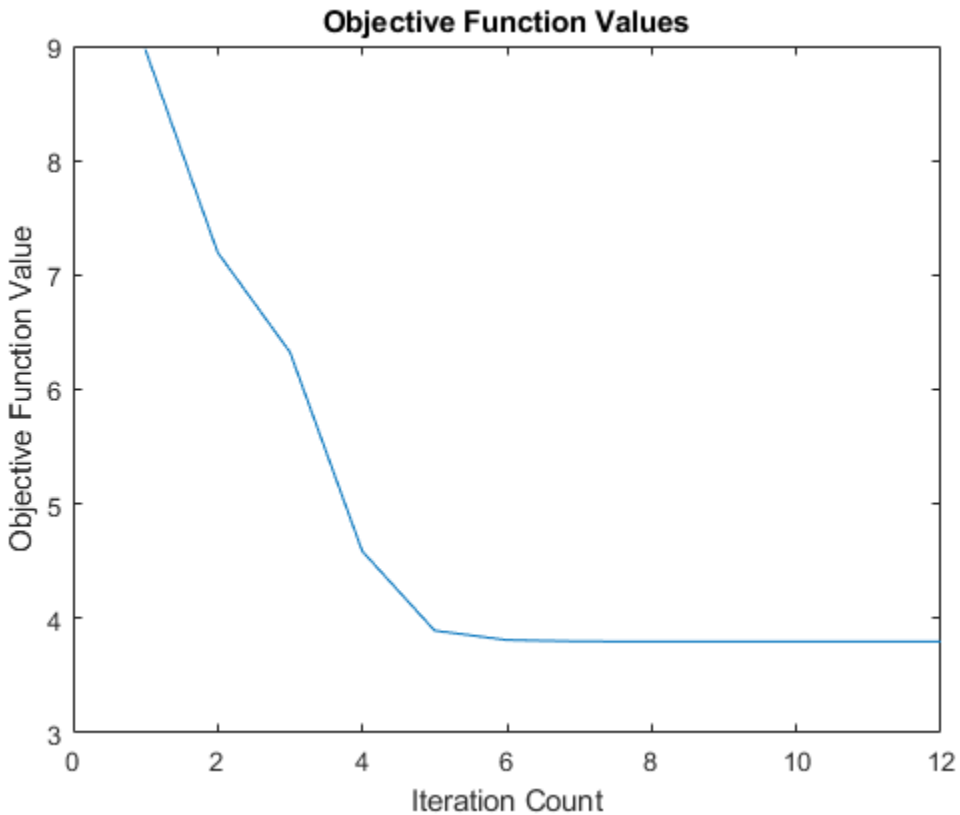
`center` contains the coordinates of the two cluster centers, `U` contains the membership grades for each of the data points, and `objFcn` contains a history of the objective function across the iterations.

The `fcm` function is an iteration loop built on top of the following routines:

- `initfcm` - initializes the problem
- `distfcm` - performs Euclidean distance calculation
- `stepfcm` - performs one iteration of clustering

To view the progress of the clustering, plot the objective function.

```
figure  
plot(objFcn)  
title('Objective Function Values')  
xlabel('Iteration Count')  
ylabel('Objective Function Value')
```

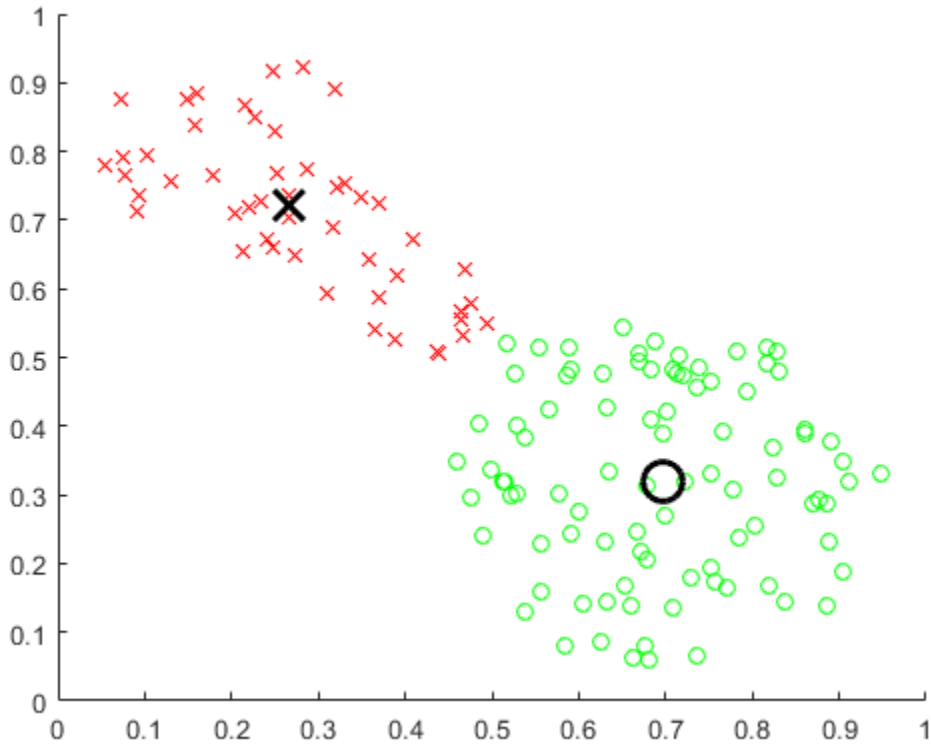


Finally, plot the two cluster centers found by the `fcm` function. The large characters in the plot indicate cluster centers.

```

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

```



Note: Every time you run this example, the `fcm` function initializes with different initial conditions. This behavior swaps the order in which the cluster centers are computed and plotted.

See Also

`fcm`

More About

- “Fuzzy Clustering” on page 4-2

Adjust Fuzzy Overlap in Fuzzy C-Means Clustering

This example shows how to adjust the amount of fuzzy overlap when performing fuzzy c-means clustering.

Create a random data set. For reproducibility, initialize the random number generator to its default value.

```
rng('default')  
data = rand(100,2);
```

Specify fuzzy partition matrix exponents.

```
M = [1.1 2.0 3.0 4.0];
```

The exponent values in M must be greater than 1, with smaller values specifying a lower degree of fuzzy overlap. In other words, as M approaches 1, the boundaries between the clusters become more crisp.

For each overlap exponent:

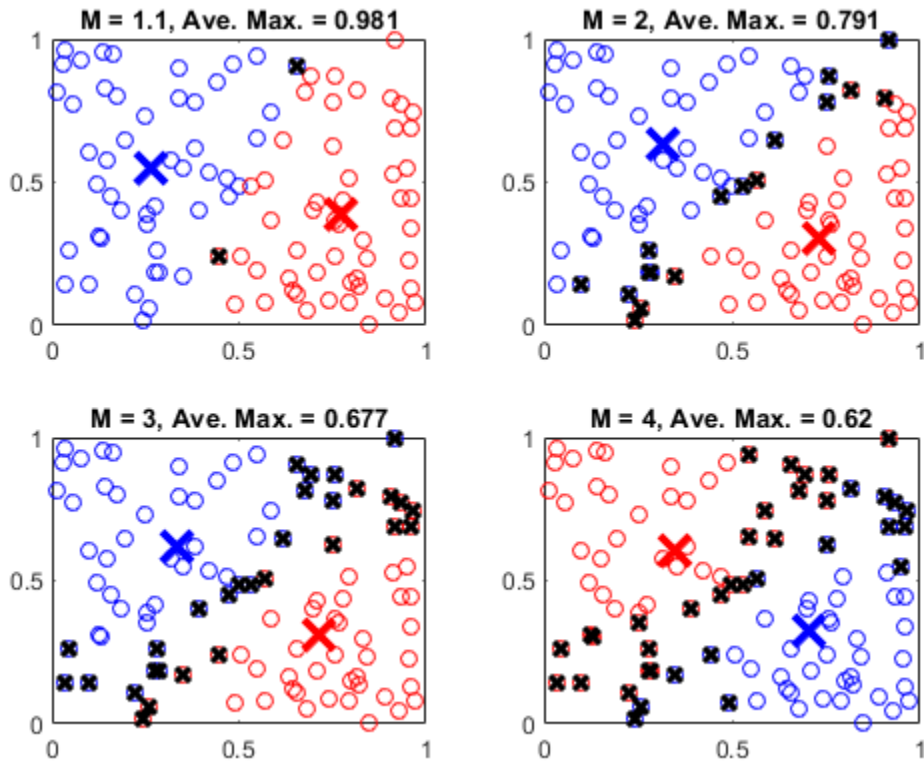
- Cluster the data.
- Classify each data point into the cluster for which it has the highest degree of membership.
- Find the data points with maximum membership values below 0.6. These points have a more fuzzy classification.
- To quantify the degree of fuzzy overlap, calculate the average maximum membership value across all data points. A higher average maximum membership value indicates that there is less fuzzy overlap.
- Plot the clustering results.

```
for i = 1:4  
    % Cluster the data.  
    options = [M(i) NaN NaN 0];  
    [centers,U] = fcm(data,2,options);  
  
    % Classify the data points.  
    maxU = max(U);  
    index1 = find(U(1,:) == maxU);  
    index2 = find(U(2,:) == maxU);
```

```
% Find data points with lower maximum membership values.
index3 = find(maxU < 0.6);

% Calculate the average maximum membership value.
averageMax = mean(maxU);

% Plot the results.
subplot(2,2,i)
plot(data(index1,1),data(index1,2),'ob')
hold on
plot(data(index2,1),data(index2,2),'or')
plot(data(index3,1),data(index3,2),'xk','LineWidth',2)
plot(centers(1,1),centers(1,2),'xb','MarkerSize',15,'LineWidth',3)
plot(centers(2,1),centers(2,2),'xr','MarkerSize',15,'LineWidth',3)
hold off
title(['M = ' num2str(M(i)) ', Ave. Max. = ' num2str(averageMax,3)])
end
```



A given data point is classified into the cluster for which it has the highest membership value, as indicated by $\max U$. A maximum membership value of 0.5 indicates that the point belongs to both clusters equally. The data points marked with a black **x** have maximum membership values below 0.6. These points have a greater degree of uncertainty in their cluster membership.

More data points with low maximum membership values indicate a greater degree of fuzzy overlap in the clustering result. The average maximum membership value, averageMax , provides a quantitative description of the overlap. An averageMax value of 1 indicates crisp clusters, with smaller values indicating more overlap.

See Also

fcm

More About

- “Fuzzy Clustering” on page 4-2
- “Cluster Quasi-Random Data Using Fuzzy C-Means Clustering” on page 4-4

Fuzzy C-Means Clustering

This example shows how to perform fuzzy c-means clustering on 2-dimensional data.

What Is Fuzzy C-Means Clustering?

Clustering of numerical data forms the basis of many classification and system modeling algorithms. The purpose of clustering is to identify natural groupings of data from a large data set to produce a concise representation of a system's behavior.

Fuzzy c-means (FCM) is a data clustering technique in which a dataset is grouped into n clusters with every datapoint in the dataset belonging to every cluster to a certain degree. For example, a certain datapoint that lies close to the center of a cluster will have a high degree of belonging or membership to that cluster and another datapoint that lies far away from the center of a cluster will have a low degree of belonging or membership to that cluster.

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

Interactive Fuzzy C-Means Clustering Example

Using the `fcmdemo` command, you can launch a GUI that lets you try out various parameter settings for the fuzzy c-means algorithm and observe the effect on the resulting 2-D clustering. You can choose a sample data set and an arbitrary number of clusters from the drop down menus on the right, and then click "Start" to start the fuzzy c-means clustering process. The clustering itself is performed by the `fcm` function.

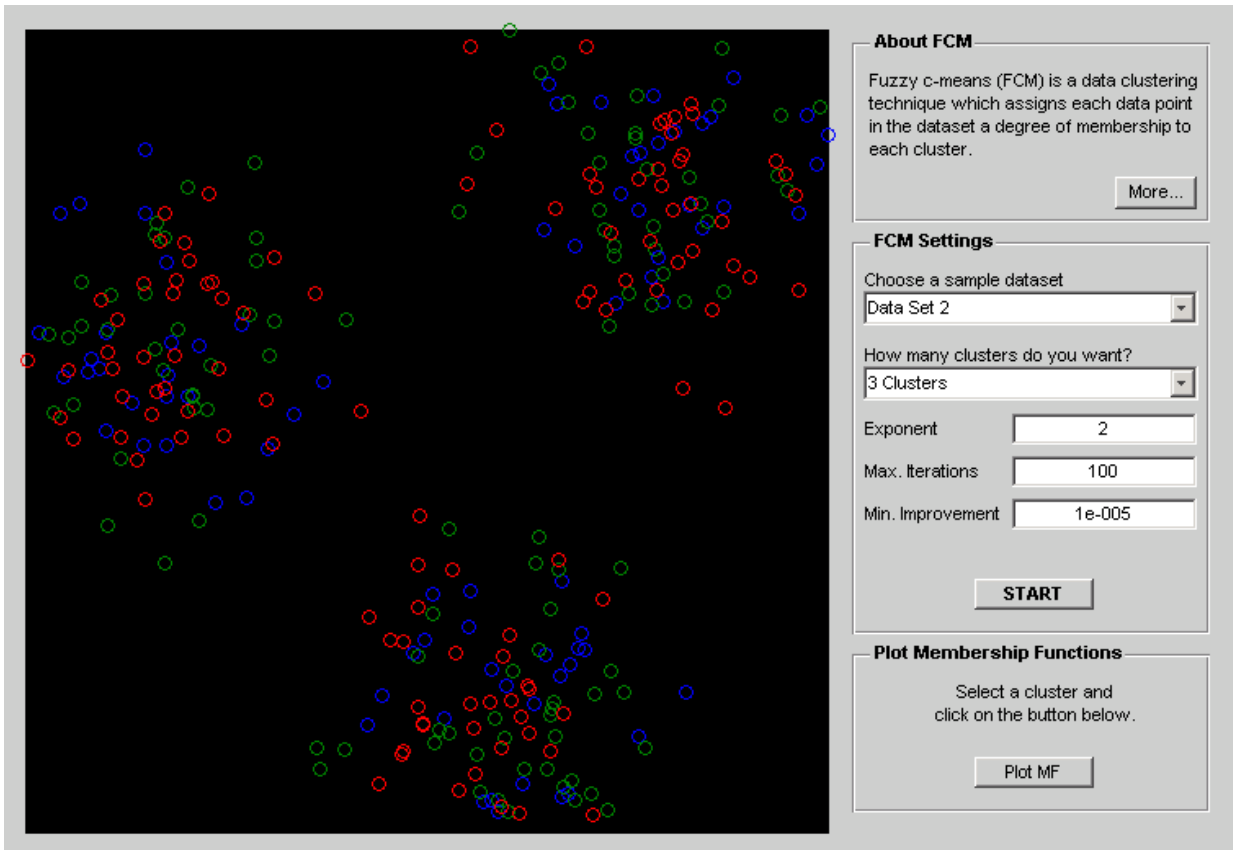


Figure 1: GUI for Fuzzy C-Means Clustering.

Once the clustering is done, you can select one of the clusters by clicking on it, and view the membership function surface by clicking the "Plot MF" button. To get a better viewing angle, click and drag inside the figure to rotate the MF surface.

You can also tune the 3 optional parameters for the FCM algorithm (exponent, maximum number of iterations and minimum amount of improvement) from the GUI and observe how the clustering process is consequently altered.

Performing Fuzzy C-Means Clustering on Your Own Data

The function `fcm` takes a data set and a desired number of clusters and returns optimal cluster centers and membership grades for each data point. You can use this information to build a fuzzy inference system by creating membership functions that represent the fuzzy qualities of each cluster.

Here is the underlying code that performs the clustering.

```
data = load('fcmdata.dat'); % load some sample data
n_clusters = 3; % number of clusters
[center,U,obj_fcn] = fcm(data, n_clusters);
```

```
Iteration count = 1, obj. fcn = 6.379151
Iteration count = 2, obj. fcn = 4.907101
Iteration count = 3, obj. fcn = 4.847428
Iteration count = 4, obj. fcn = 4.447136
Iteration count = 5, obj. fcn = 3.306271
Iteration count = 6, obj. fcn = 2.422911
Iteration count = 7, obj. fcn = 2.180720
Iteration count = 8, obj. fcn = 2.109423
Iteration count = 9, obj. fcn = 2.084711
Iteration count = 10, obj. fcn = 2.075537
Iteration count = 11, obj. fcn = 2.071419
Iteration count = 12, obj. fcn = 2.069188
Iteration count = 13, obj. fcn = 2.067795
Iteration count = 14, obj. fcn = 2.066845
Iteration count = 15, obj. fcn = 2.066166
Iteration count = 16, obj. fcn = 2.065670
Iteration count = 17, obj. fcn = 2.065304
Iteration count = 18, obj. fcn = 2.065032
Iteration count = 19, obj. fcn = 2.064830
Iteration count = 20, obj. fcn = 2.064679
Iteration count = 21, obj. fcn = 2.064567
Iteration count = 22, obj. fcn = 2.064482
Iteration count = 23, obj. fcn = 2.064419
Iteration count = 24, obj. fcn = 2.064372
Iteration count = 25, obj. fcn = 2.064337
Iteration count = 26, obj. fcn = 2.064310
Iteration count = 27, obj. fcn = 2.064291
Iteration count = 28, obj. fcn = 2.064276
```

Iteration count = 29, obj. fcn = 2.064265
Iteration count = 30, obj. fcn = 2.064256

See Also

fcM

More About

- “Fuzzy Clustering” on page 4-2

Fuzzy C-Means Clustering for Iris Data

This example shows how to use Fuzzy C-Means clustering for Iris dataset.

Load Data

The dataset is obtained from the data file 'iris.dat'. This dataset was collected by botanist Anderson and contains random samples of flowers belonging to three species of iris flowers *setosa*, *versicolor*, and *virginica*. For each of the species, 50 observations for sepal length, sepal width, petal length, and petal width are recorded.

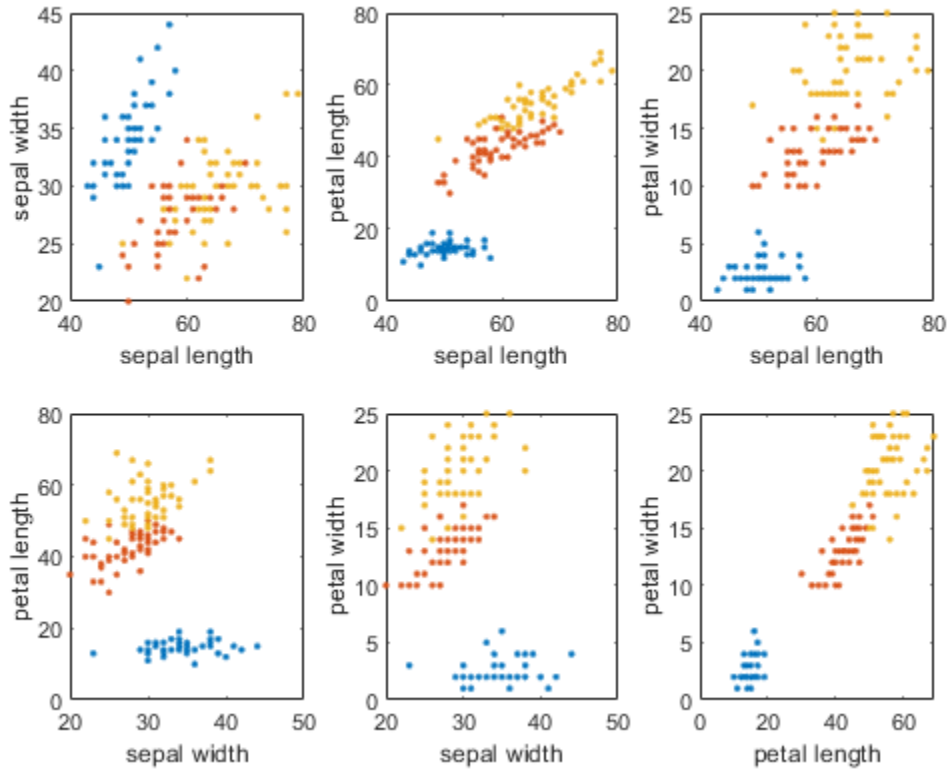
The dataset is partitioned into three groups named *setosa*, *versicolor*, and *virginica*. This is shown in the following code snippet.

```
load iris.dat
setosa = iris((iris(:,5)==1),:);           % data for setosa
versicolor = iris((iris(:,5)==2),:);     % data for versicolor
virginica = iris((iris(:,5)==3),:);      % data for virginica
obsv_n = size(iris, 1);                  % total number of observations
```

Plot Data in 2-D

The data to be clustered is 4-dimensional data and represents sepal length, sepal width, petal length, and petal width. From each of the three groups (*setosa*, *versicolor* and *virginica*), two characteristics (for example, sepal length vs. sepal width) of the flowers are plotted in a 2-dimensional plot. This is done using the following code snippet.

```
Characteristics = {'sepal length', 'sepal width', 'petal length', 'petal width'};
pairs = [1 2; 1 3; 1 4; 2 3; 2 4; 3 4];
h = figure;
for j = 1:6,
    x = pairs(j, 1);
    y = pairs(j, 2);
    subplot(2,3,j);
    plot([setosa(:,x) versicolor(:,x) virginica(:,x)],...
         [setosa(:,y) versicolor(:,y) virginica(:,y)]], '.');
    xlabel(Characteristics{x}, 'FontSize', 10);
    ylabel(Characteristics{y}, 'FontSize', 10);
end
```



Setup Parameters

Next, the parameters required for Fuzzy C-Means clustering such as number of clusters, exponent for the partition matrix, maximum number of iterations and minimum improvement are defined and set. This are shown in the following code snippet.

```
cluster_n = 3;           % Number of clusters
expo = 2.0;             % Exponent for U
max_iter = 100;        % Max. iteration
min_impro = 1e-6;     % Min. improvement
```

Compute Clusters

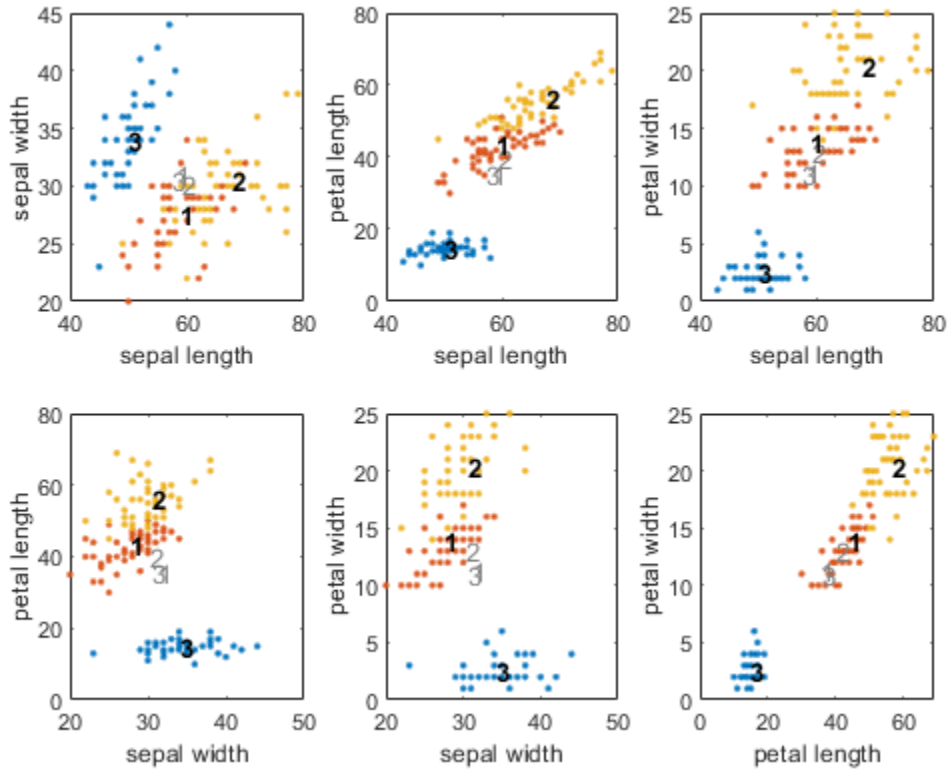
Fuzzy C-Means clustering is an iterative process. First, the initial fuzzy partition matrix is generated and the initial fuzzy cluster centers are calculated. In each step of the iteration, the cluster centers and the membership grade point are updated and the objective function is minimized to find the best location for the clusters. The process stops when the maximum number of iterations is reached, or when the objective function improvement between two consecutive iterations is less than the minimum amount of improvement specified. This is shown in the following code snippet.

```
% initialize fuzzy partition
U = initfcm(cluster_n, obsv_n);
% plot the data if the figure window is closed
if ishghandle(h)
    figure(h);
else
    for j = 1:6,
        x = pairs(j, 1);
        y = pairs(j, 2);
        subplot(2,3,j);
        plot([setosa(:,x) versicolor(:,x) virginica(:,x)],...
             [setosa(:,y) versicolor(:,y) virginica(:,y)], '.');
        xlabel(Characteristics{x}, 'FontSize',10);
        ylabel(Characteristics{y}, 'FontSize',10);
    end
end
% iteration
for i = 1:max_iter,
    [U, center, obj] = stepfcm(iris, U, cluster_n, expo);
    fprintf('Iteration count = %d, obj. fcn = %f\n', i, obj);
    % refresh centers
    if i>1 && (abs(obj - lastobj) < min_impro)
        for j = 1:6,
            subplot(2,3,j);
            for k = 1:cluster_n,
                text(center(k, pairs(j,1)), center(k,pairs(j,2)), int2str(k), 'FontWei
            end
        end
        break;
    elseif i==1
        for j = 1:6,
            subplot(2,3,j);
            for k = 1:cluster_n,
                text(center(k, pairs(j,1)), center(k,pairs(j,2)), int2str(k), 'color',
```

```
        end
      end
    end
    lastobj = obj;
end
```

```
Iteration count = 1, obj. fcn = 28838.424340

Iteration count = 2, obj. fcn = 21010.880067
Iteration count = 3, obj. fcn = 15272.280943
Iteration count = 4, obj. fcn = 11029.756194
Iteration count = 5, obj. fcn = 10550.015503
Iteration count = 6, obj. fcn = 10301.776800
Iteration count = 7, obj. fcn = 9283.793786
Iteration count = 8, obj. fcn = 7344.379868
Iteration count = 9, obj. fcn = 6575.117093
Iteration count = 10, obj. fcn = 6295.215539
Iteration count = 11, obj. fcn = 6167.772051
Iteration count = 12, obj. fcn = 6107.998500
Iteration count = 13, obj. fcn = 6080.461019
Iteration count = 14, obj. fcn = 6068.116247
Iteration count = 15, obj. fcn = 6062.713326
Iteration count = 16, obj. fcn = 6060.390433
Iteration count = 17, obj. fcn = 6059.403978
Iteration count = 18, obj. fcn = 6058.988494
Iteration count = 19, obj. fcn = 6058.814438
Iteration count = 20, obj. fcn = 6058.741777
Iteration count = 21, obj. fcn = 6058.711512
Iteration count = 22, obj. fcn = 6058.698925
Iteration count = 23, obj. fcn = 6058.693695
Iteration count = 24, obj. fcn = 6058.691523
Iteration count = 25, obj. fcn = 6058.690622
Iteration count = 26, obj. fcn = 6058.690247
Iteration count = 27, obj. fcn = 6058.690092
Iteration count = 28, obj. fcn = 6058.690028
Iteration count = 29, obj. fcn = 6058.690001
Iteration count = 30, obj. fcn = 6058.689990
Iteration count = 31, obj. fcn = 6058.689985
Iteration count = 32, obj. fcn = 6058.689983
Iteration count = 33, obj. fcn = 6058.689983
```



The figure shows the initial and final fuzzy cluster centers. The bold numbers represent the final fuzzy cluster centers obtained by updating them iteratively.

See Also

fcm

More About

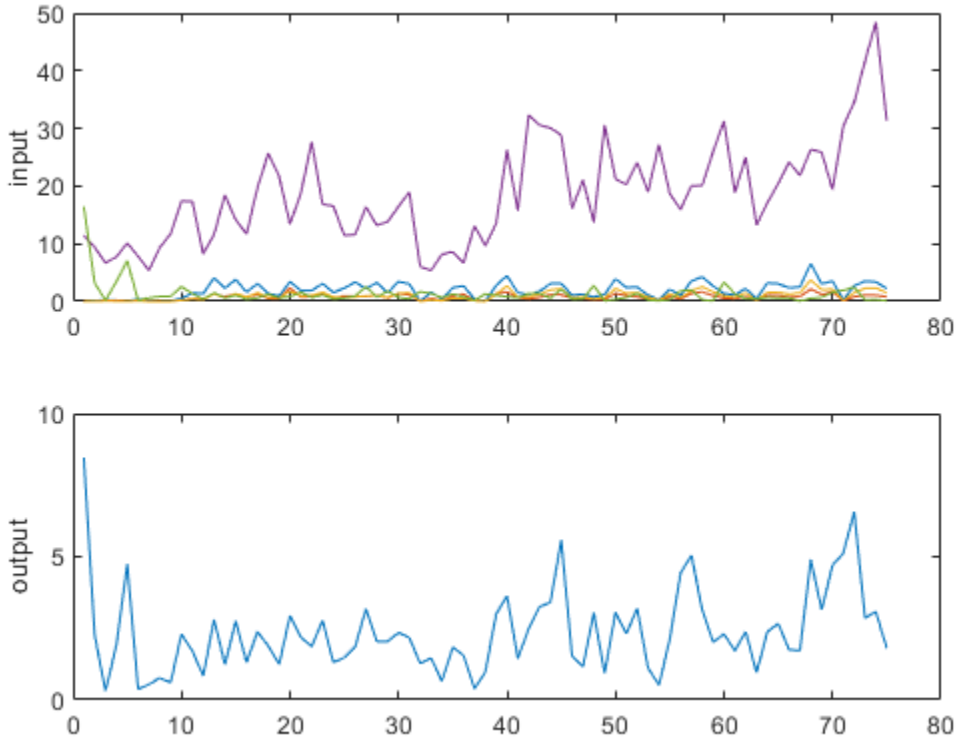
- “Fuzzy Clustering” on page 4-2

Model Suburban Commuting Using Subtractive Clustering

This example shows how to model the relationship between the number of automobile trips generated from an area and the demographics of the area using the `genfis` function. 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 and plot the data.

```
mytripdata
subplot(2,1,1)
plot(datin)
ylabel('input')
subplot(2,1,2)
plot(datout)
ylabel('output')
```



The `mytripdata` command creates several variables in the workspace. Of the original 100 data points, use 75 data points as training data (`datin` and `datout`) and 25 data points as checking data (as well as for test data to validate the model). The checking data input/output pair variables are `chkdatin` and `chkdatout`.

Generate a model from the data using subtractive clustering using the `genfis` command.

First, create a `genfisOptions` option set for subtractive clustering, specifying `ClusterInfluenceRange` range property. The `ClusterInfluenceRange` property indicates the range of influence of a cluster when you consider the data space as a unit hypercube. Specifying a small cluster radius usually yields many small clusters in the data, and results in many rules. Specifying a large cluster radius usually yields a few large clusters in the data, and results in fewer rules.

```
opt = genfisOptions('SubtractiveClustering', 'ClusterInfluenceRange', 0.5);
```

Generate the FIS model using the training data and the specified options.

```
fismat = genfis(datin, datout, opt);
```

The `genfis` command uses a one-pass method that does not perform any iterative optimization. The model type for the generated FIS object is a first order Sugeno model with three rules.

Verify the model. Here, `trnRMSE` is the root mean squared error of the system generated by the training data.

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

```
trnRMSE = 0.5276
```

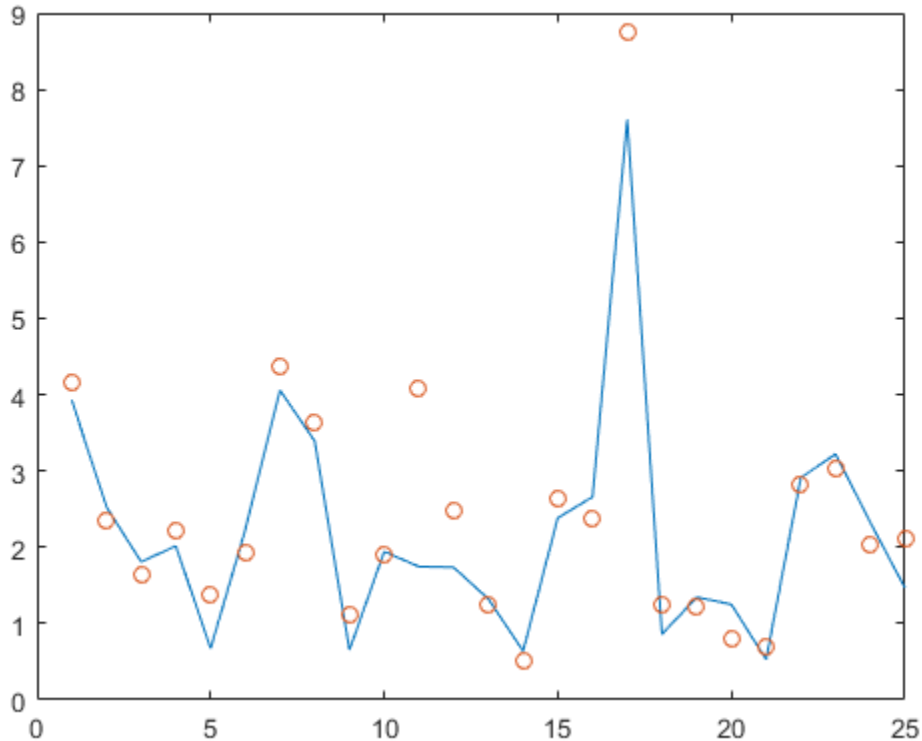
Next, apply the test data to the FIS to validate the model. In this example, the validation data is used for both checking and testing the FIS parameters. Here, `chkRMSE` is the root mean squared error of the system generated by the validation data.

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

```
chkRMSE = 0.6179
```

Plot the output of the model, `chkfuzout`, against the validation data, `chkdatout`.

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



The model output and validation data are shown as circles and solid blue line, respectively. The plot shows that the model does not perform well on the validation data.

At this point, you can use the optimization capability of `anfis` to improve the model. First, try using a relatively short training period (20 epochs) without using validation data, and then test the resulting FIS model against the testing data.

```
anfis0pt = anfisOptions('InitialFIS',fismat,'EpochNumber',20,...  
                       'InitialStepSize',0.1);  
fismat2 = anfis([datin datout],anfis0pt);
```

ANFIS info:

Number of nodes: 44

Number of linear parameters: 18

Number of nonlinear parameters: 30
 Total number of parameters: 48
 Number of training data pairs: 75
 Number of checking data pairs: 0
 Number of fuzzy rules: 3

Start training ANFIS ...

```

1      0.527607
2      0.513727
3      0.492996
4      0.499985
5      0.490585
6      0.492924
7      0.48733
Step size decreases to 0.090000 after epoch 7.
8      0.485036
9      0.480813
10     0.475097
Step size increases to 0.099000 after epoch 10.
11     0.469759
12     0.462516
13     0.451177
14     0.447856
Step size increases to 0.108900 after epoch 14.
15     0.444357
16     0.433904
17     0.433739
18     0.420408
Step size increases to 0.119790 after epoch 18.
19     0.420512
20     0.420275
    
```

Designated epoch number reached --> ANFIS training completed at epoch 20.

Minimal training RMSE = 0.420275

After the training is complete, validate the model.

```

fuzout2 = evalfis(fismat2,datin);
trnRMSE2 = norm(fuzout2-datout)/sqrt(length(fuzout2))
    
```

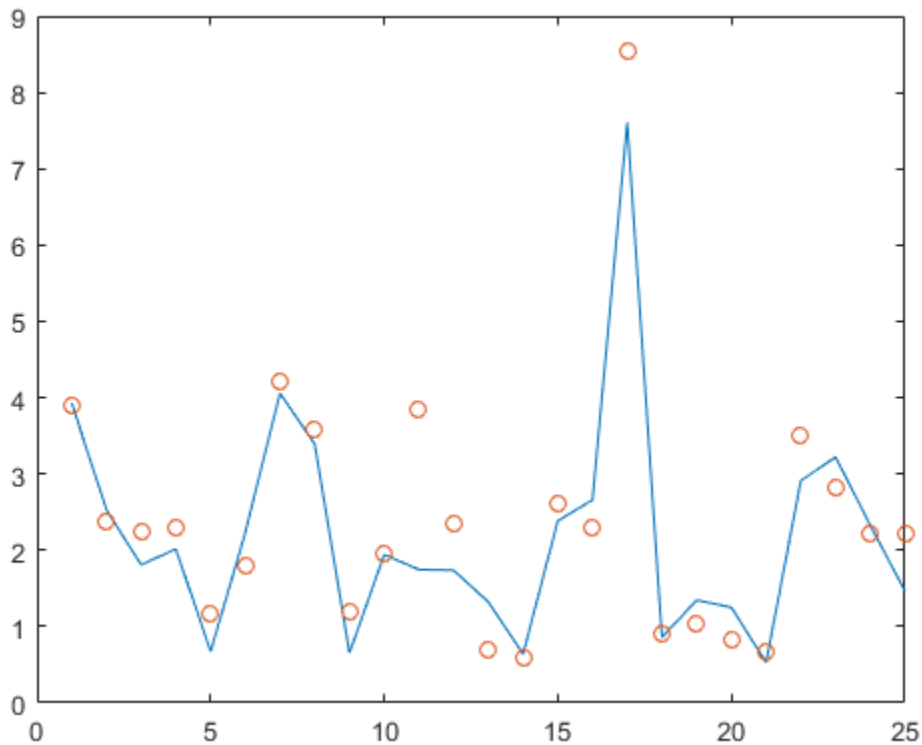
trnRMSE2 = 0.4203

```
chkfuzout2 = evalfis(fismat2,chkdatin);  
chkRMSE2 = norm(chkfuzout2-chkdatout)/sqrt(length(chkfuzout2))
```

```
chkRMSE2 = 0.5894
```

The model has improved a lot with respect to the training data, but only a little with respect to the validation data. Plot the improved model output obtained using `anfis` against the testing data.

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



The model output and validation data are shown as circles and solid blue line, respectively. This plot shows that subtractive clustering with `genfis` can be used as a standalone, fast method for generating a fuzzy model from data, or as a preprocessor to determine the initial rules for `anfis` training. An important advantage of using a clustering method to find rules is that the resultant rules are more tailored to the input data than they are in a FIS generated without clustering. This result reduces the problem of an excessive propagation of rules when the input data has a high dimension.

Overfitting can be detected when the checking error starts to increase while the training error continues to decrease.

To check the model for overfitting, use `anfis` with validation data to train the model for 200 epochs.

First configure the ANFIS training options by modifying the existing `anfisOptions` option set. Specify the epoch number and validation data. Since the number of training epochs is larger, suppress the display of training information to the Command Window.

```
anfisOpt.EpochNumber = 200;
anfisOpt.ValidationData = [chkdatin chkdatout];
anfisOpt.DisplayANFISInformation = 0;
anfisOpt.DisplayErrorValues = 0;
anfisOpt.DisplayStepSize = 0;
anfisOpt.DisplayFinalResults = 0;
```

Train the FIS.

```
[fismat3, trnErr, stepSize, fismat4, chkErr] = anfis([datin datout], anfisOpt);
```

Here,

- `fismat3` is the FIS object when the training error reaches a minimum.
- `fismat4` is the snapshot FIS object when the validation data error reaches a minimum.
- `stepSize` is a history of the training step sizes.
- `trnErr` is the RMSE using the training data
- `chkErr` is the RMSE using the validation data for each training epoch.

After the training completes, validate the model.

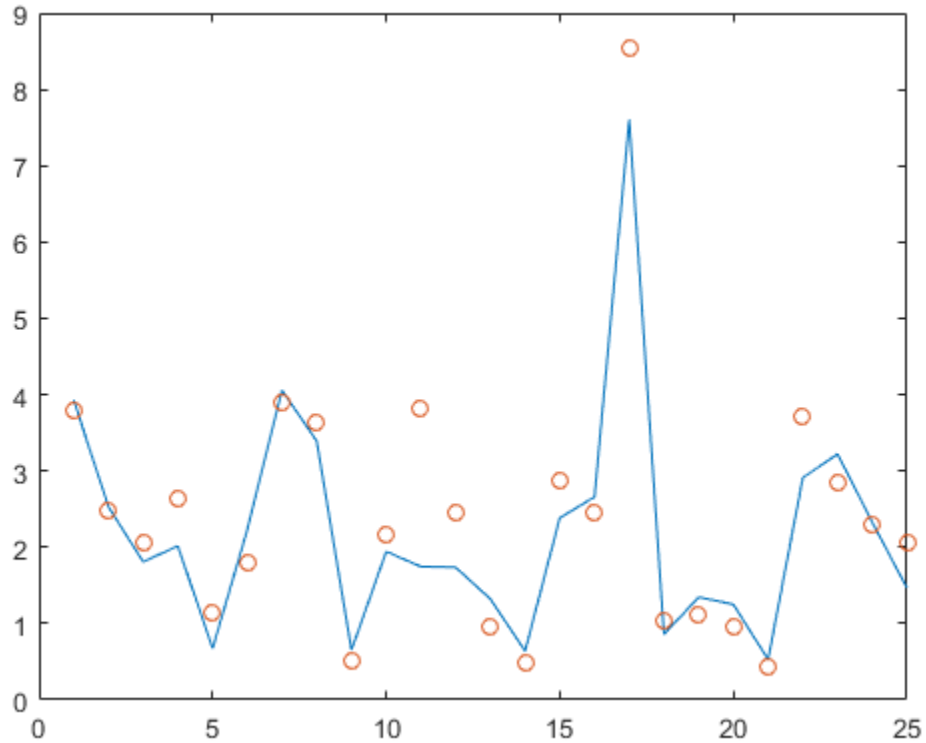
```
fuzout4 = evalfis(fismat4, datin);
trnRMSE4 = norm(fuzout4 - datout) / sqrt(length(fuzout4))
```

```
trnRMSE4 = 0.3393  
  
chkfuzout4 = evalfis(fismat4,chkdatin);  
chkRMSE4 = norm(chkfuzout4-chkdatout)/sqrt(length(chkfuzout4))  
  
chkRMSE4 = 0.5834
```

The error with the training data is the lowest thus far, and the error with the validation data is also slightly lower than before. This result suggests possible overfitting, which occurs when you fit the fuzzy system to the training data so well that it no longer does a good job of fitting the validation data. The result is a loss of generality.

View the improved model output. Plot the model output against the checking data.

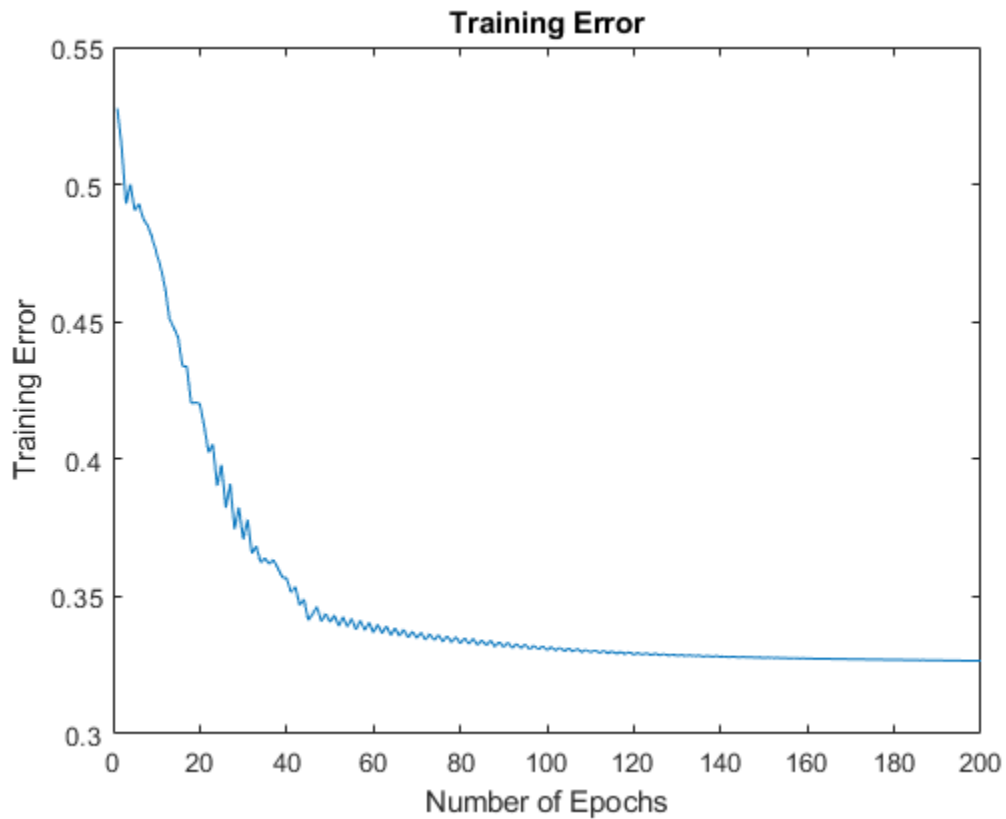
```
figure  
plot(chkdatout)  
hold on  
plot(chkfuzout4,'o')  
hold off
```

The model output and validation data are shown as circles and solid blue line, respectively.

Next, plot the training error, `trnErr`.

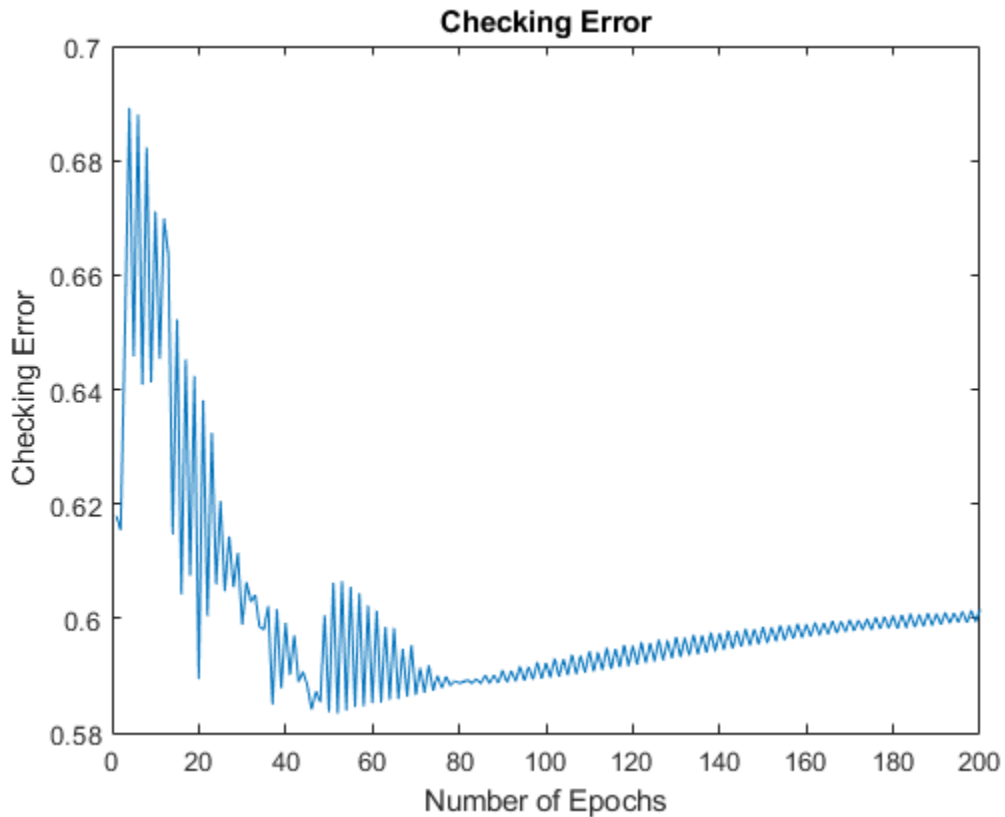
```
figure
plot(trnErr)
title('Training Error')
xlabel('Number of Epochs')
ylabel('Training Error')
```



This plot shows that the training error settles at about the 60th epoch point.

Plot the checking error, `chkErr`.

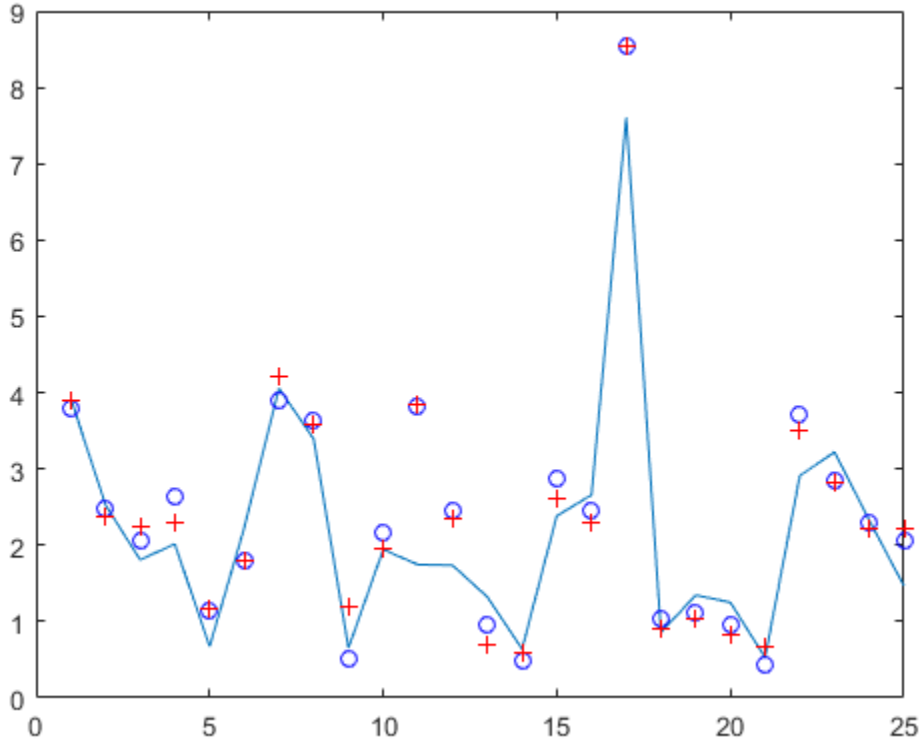
```
figure
plot(chkErr)
title('Checking Error')
xlabel('Number of Epochs')
ylabel('Checking Error')
```



The plot shows that the smallest value of the validation data error occurs at the 52nd epoch. After this point it increases slightly even as `anfis` continues to minimize the error against the training data all the way to the 200th epoch. Depending on the specified error tolerance, the plot also indicates the ability of the model to generalize the test data.

You can also compare the output of `fismat2` and `fismat4` against the validation data, `chkdatout`.

```
figure
plot(chkdatout)
hold on
plot(chkfuzout4, 'ob')
plot(chkfuzout2, '+r')
```



See Also

`anfis` | `subclust`

More About

- “Fuzzy Clustering” on page 4-2

Modeling Traffic Patterns using Subtractive Clustering

This example shows how to use subtractive clustering to model traffic patterns in an area based on the demographics of the area.

The Problem: Understanding Traffic Patterns

In this example we attempt to understand the relationship between the number of automobile trips generated from an area and the area's demographics. Demographic and trip data were collected from 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.

Hereon, the demographic factors will be addressed as inputs and the trips generated will be addressed as output. Hence our problem has five input variables (five demographic factors) and one output variable (number of trips generated).

The Data

Load the input and output variables used for this example into the workspace.

```
tripdata
```

Two variables are loaded in the workspace, `datin` and `datout`. `datin` has 5 columns representing the 5 input variables and `datout` has 1 column representing the 1 output variable.

```
subplot(2,1,1)
plot(datin)
legend('population', 'num. of dwelling units', 'vehicle ownership',...
       'median household income', 'total employment')
title('Input Variables', 'fontsize', 10)

subplot(2,1,2)
plot(datout)
legend('num of trips')
title('Output Variable', 'fontsize', 10)
```

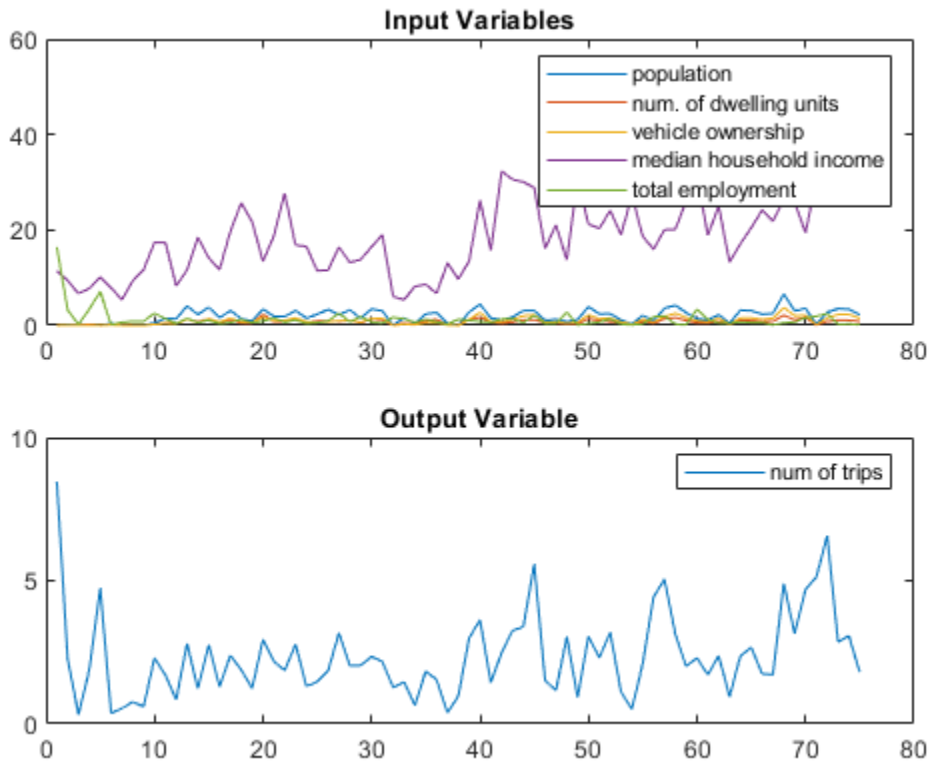


Figure 1: Input and Output variables

The number of rows in `datin` and `datout`, 75, represent the number of observations or samples or datapoints available. A row in `datin`, say row 11, constitutes a set of observed values of the 5 input variables (population, number of dwelling units, vehicle ownership, median household income and total employment) and the corresponding row, row 11, in `datout` represents the observed value for the number of trips generated given the observations made for the input variables.

We will model the relationship between the input variables (demographics) and the output variable (trips) by first clustering the data. The cluster centers will then be used as a basis to define a Fuzzy Inference System (FIS) which can then be used to explore and understand traffic patterns.

Why Clustering and Fuzzy Logic?

Clustering can be a very effective technique to identify natural groupings in data from a large data set, thereby allowing concise representation of relationships embedded in the data. In this example, clustering allows us to group traffic patterns into broad categories hence allowing for easier understandability.

Fuzzy logic is an effective paradigm to handle imprecision. It can be used to take fuzzy or imprecise observations for inputs and yet arrive at crisp and precise values for outputs. Also, the Fuzzy Inference System (FIS) is a simple and commonsensical way to build systems without using complex analytical equations.

In our example, fuzzy logic will be employed to capture the broad categories identified during clustering into a Fuzzy Inference System (FIS). The FIS will then act as a model that will reflect the relationship between demographics and auto trips.

Clustering and fuzzy logic together provide a simple yet powerful means to model the traffic relationship that we want to study.

Clustering the Data

`subclust` is the function that implements a clustering technique called subtractive clustering. Subtractive clustering, [1], is a fast, one-pass algorithm for estimating the number of clusters and the cluster centers in a dataset.

In this section, we will see how subtractive clustering is performed on a dataset and in the next section we will explore independently how clustering is used to build a Fuzzy Inference System(FIS).

```
[C,S] = subclust([datin datout],0.5);
```

The first argument to the `subclust` function is the data to be clustered. The second argument to the function is the `radii` which marks a cluster's radius of influence in the input space.

The variable `C` now holds all the centers of the clusters that have been identified by `subclust`. Each row of `C` contains the position of a cluster.

`C`

`C =`

1.8770	0.7630	0.9170	18.7500	1.5650	2.1830
0.3980	0.1510	0.1320	8.1590	0.6250	0.6480
3.1160	1.1930	1.4870	19.7330	0.6030	2.3850

In this case, `C` has 3 rows representing 3 clusters with 6 columns representing the positions of the clusters in each dimension.

`subclust` has hence identified 3 natural groupings in the demographic-trip dataset being considered. The following plot shows how the clusters have been identified in the 'total employment' and 'trips' dimensions of the input space.

```
clf
plot(datin(:,5),datout(:,1),'.',C(:,5),C(:,6),'r*')
legend('Data points','Cluster centers','Location','SouthEast')
xlabel('total employment','fontsize',10)
ylabel('num of trips','fontsize',10)
title('Data and Clusters in selected two dimensions of the input space','fontsize',10)
```

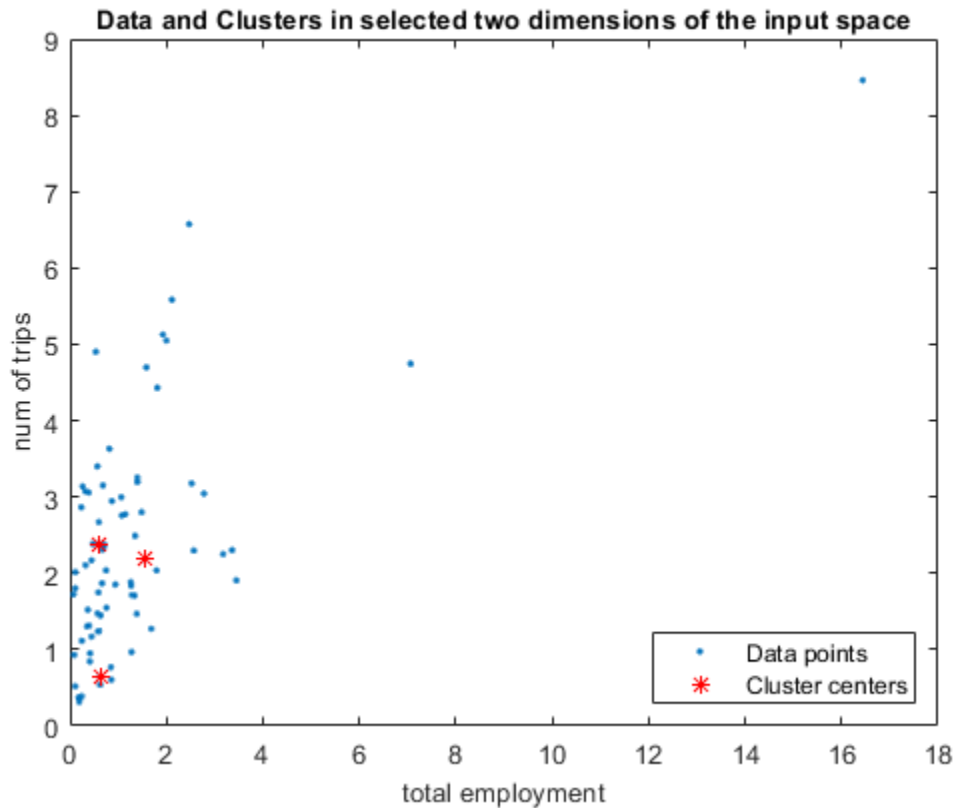



Figure 2: Cluster centers in the 'total employment' and 'trips' dimensions of the input space

The variable S 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.

S

$S =$

1.1621 0.4117 0.6555 7.6139 2.8931 1.4395

S in this case has 6 columns representing the influence of the cluster centers on each of the 6 dimensions.

Generating the Fuzzy Inference System (FIS)

`genfis` is the function that creates a FIS using subtractive clustering. `genfis` employs `subclust` behind the scenes to cluster the data and uses the cluster centers and their range of influences to build a FIS which will then be used to explore and understand traffic patterns.

```
myfis=genfis(datin,datout, ...  
            genfisOptions('SubtractiveClustering','ClusterInfluenceRange',0.5));
```

The first argument is the input variables matrix `datin`, the second argument is the output variables matrix `datout` and the third argument is the `radii` that should be used while using `subclust`.

`genfis` assigns default names for inputs, outputs and membership functions. For our understanding it is beneficial to rename the inputs and outputs meaningfully.

Assign names to the inputs and outputs.

```
myfis.Inputs(1).Name = "population";  
myfis.Inputs(2).Name = "dwelling units";  
myfis.Inputs(3).Name = "num vehicles";  
myfis.Inputs(4).Name = "income";  
myfis.Inputs(5).Name = "employment";  
myfis.Outputs(1).Name = "num of trips";
```

Understanding the Clusters-FIS Relationship

An FIS is composed of inputs, outputs, and rules. Each input and output can have any number of membership functions. The rules dictate the behavior of the fuzzy system based on inputs, outputs and membership functions. `genfis` constructs the FIS in an attempt to capture the position and influence of each cluster in the input space.

`myfis` is the FIS that `genfis` has generated. Since the dataset has 5 input variables and 1 output variable, `genfis` constructs a FIS with 5 inputs and 1 output. Each input and output has as many membership functions as the number of clusters that `subclust` has identified. As seen previously, for the current dataset `subclust` identified 3 clusters. Therefore each input and output will be characterized by 3 membership functions. Also, the number of rules equals the number of clusters and hence 3 rules are created.

We can now probe the FIS to understand how the clusters got converted internally into membership functions and rules using the Fuzzy Logic Designer app.

```
fuzzyLogicDesigner(myfis)
```

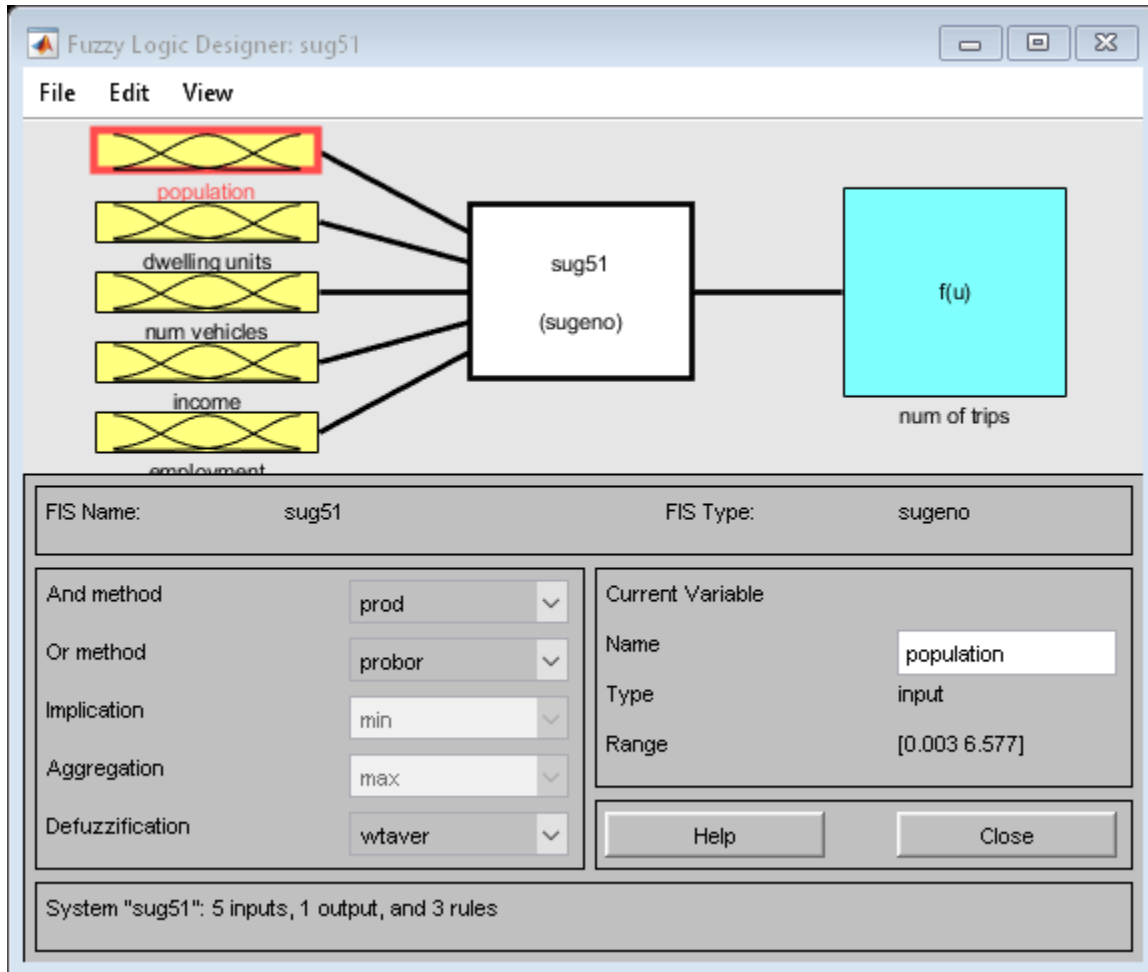


Figure 3: The graphical editor for building Fuzzy Inference Systems (FIS)

As can be seen, the FIS has 5 inputs and 1 output with the inputs mapped to the outputs through a rule base (white box in the figure).

Let's now try to analyze how the cluster centers and the membership functions are related.

`mfedit(myfis)`

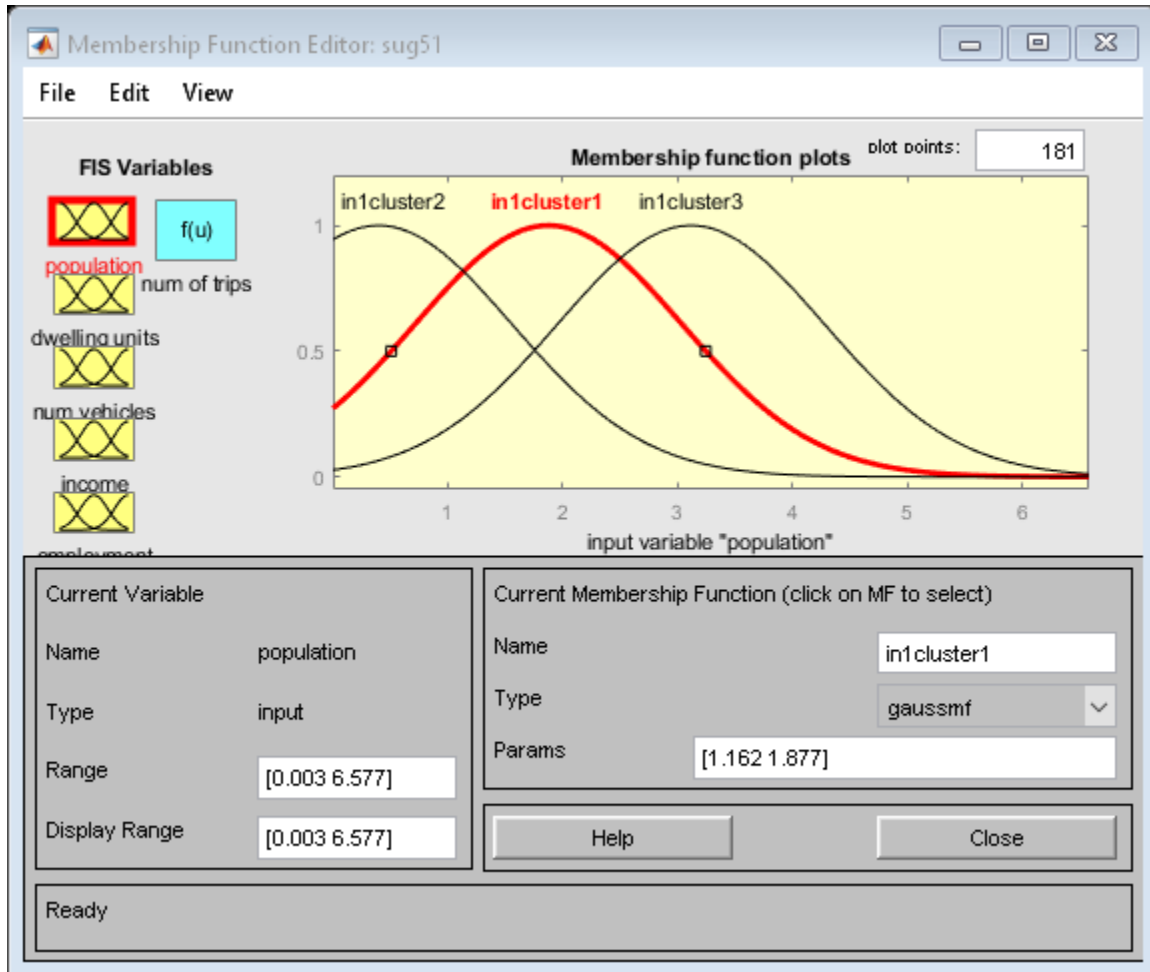


Figure 4: The graphical membership function editor

`mfedit(myfis)` launches the graphical membership function editor. It can also be launched by clicking on the inputs or the outputs in the FIS editor launched by `fuzzyLogicDesigner`.

Notice that all the inputs and outputs have exactly 3 membership functions. The 3 membership functions represent the 3 clusters that were identified by `subclust`.

Each input in the FIS represents an input variable in the input dataset `datin` and each output in the FIS represents an output variable in the output dataset `datout`.

By default, the first membership function, `in1cluster1`, of the first input `population` would be selected in the membership function editor. Notice that the membership function type is `gaussmf` (Gaussian type membership function) and the parameters of the membership function are `[1.162 1.877]`, where `1.162` represents the spread coefficient of the Gaussian curve and `1.877` represents the center of the Gaussian curve. `in1cluster1` captures the position and influence of the first cluster for the input variable `population`. ($C(1,1)=1.877$, $S(1)=1.1621$)

Similarly, the position and influence of the other 2 clusters for the input variable `population` are captured by the other two membership functions `in1cluster2` and `in1cluster3`.

The rest of the 4 inputs follow the exact pattern mimicking the position and influence of the 3 clusters along their respective dimensions in the dataset.

Now, let's explore how the fuzzy rules are constructed.

```
ruleedit(myfis)
```

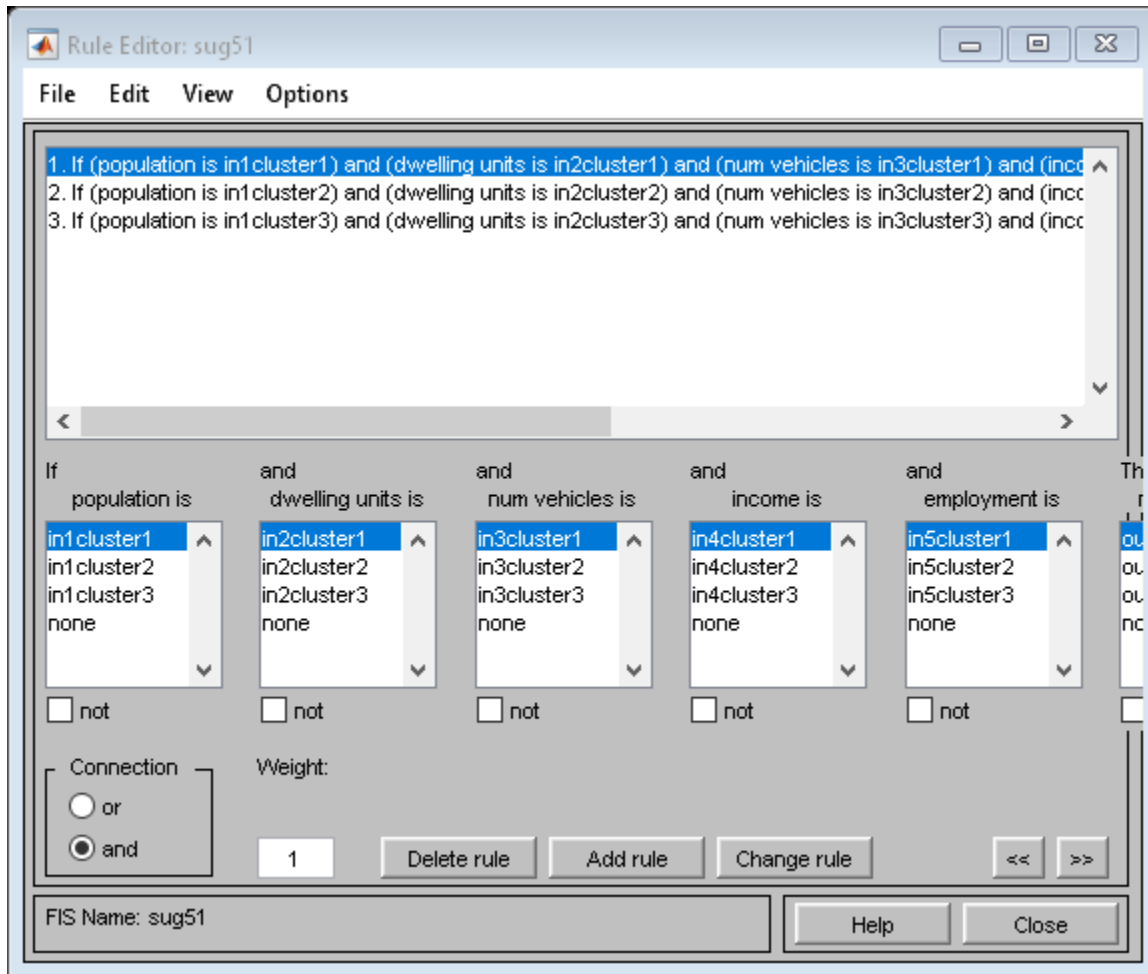


Figure 5: The graphical rule editor

ruleedit is the graphical fuzzy rule editor. As you can notice, there are exactly three rules. Each rule attempts to map a cluster in the input space to a cluster in the output space.

The first rule can be explained simply as follows. If the inputs to the FIS, `population`, `dwelling units`, `num vehicles`, `income`, and `employment`, strongly belong to their respective `cluster1` membership functions then the output, `num of trips`, must

strongly belong to its `cluster1` membership function. The (1) at the end of the rule is to indicate that the rule has a weight or an importance of "1". Weights can take any value between 0 and 1. Rules with lesser weights will count for less in the final output.

The significance of the rule is that it succinctly maps cluster 1 in the input space to cluster 1 in the output space. Similarly the other two rules map cluster 2 and cluster 3 in the input space to cluster 2 and cluster 3 in the output space.

If a datapoint closer to the first cluster, or in other words having strong membership to the first cluster, is fed as input to `myfis` then `rule1` will fire with more firing strength than the other two rules. Similarly, an input with strong membership to the second cluster will fire the second rule will with more firing strength than the other two rules and so on.

The output of the rules (firing strengths) are then used to generate the output of the FIS through the output membership functions.

The one output of the FIS, `num of trips`, has 3 linear membership functions representing the 3 clusters identified by `subclust`. The coefficients of the linear membership functions though are not taken directly from the cluster centers. Instead, they are estimated from the dataset using least squares estimation technique.

All 3 membership functions in this case will be of the form $a * \text{population} + b * \text{dwelling units} + c * \text{num vehicles} + d * \text{income} + e * \text{employment} + f$, where `a`, `b`, `c`, `d`, `e` and `f` represent the coefficients of the linear membership function. Click on any of the `num of trips` membership functions in the membership function editor to observe the parameters of these linear membership functions.

Using the FIS for Data Exploration

You can now use the FIS that has been constructed to understand the underlying dynamics of relationship being modeled.

```
surfview(myfis)
```

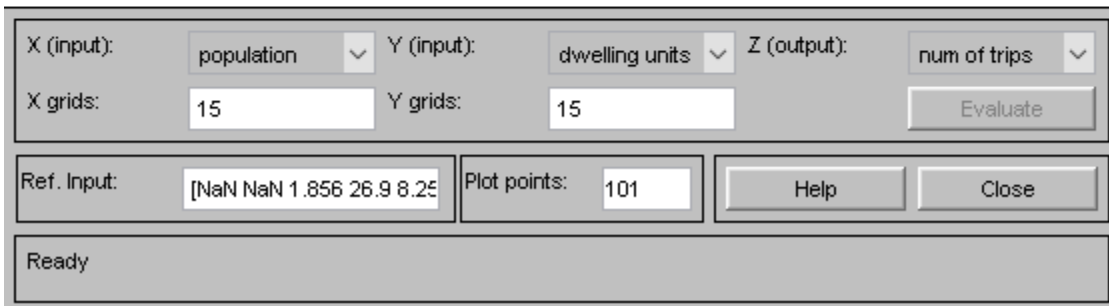
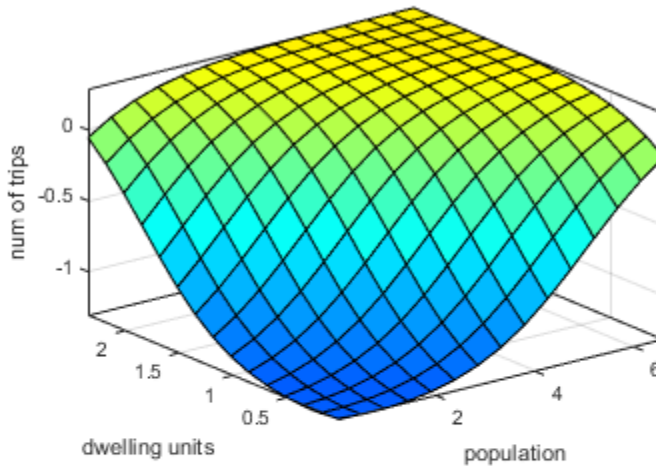


Figure 6: Input-Output Surface viewer

surfview is the surface viewer that helps view the input-output surface of the fuzzy system. In other words, this tool simulates the response of the fuzzy system for the entire range of inputs that the system is configured to work for. Thereafter, the output or the response of the FIS to the inputs are plotted against the inputs as a surface. This visualization is very helpful to understand how the system is going to behave for the entire range of values in the input space.

In the plot above the surface viewer shows the output surface for two inputs **population** and **num of dwelling units**. As you can see the number of auto trips increases with increase in population and dwelling units, which sounds very rational. You can change the

inputs in the X and Y drop-down boxes to observe the output surface with respect to the inputs you choose.

ruleview(myfis)

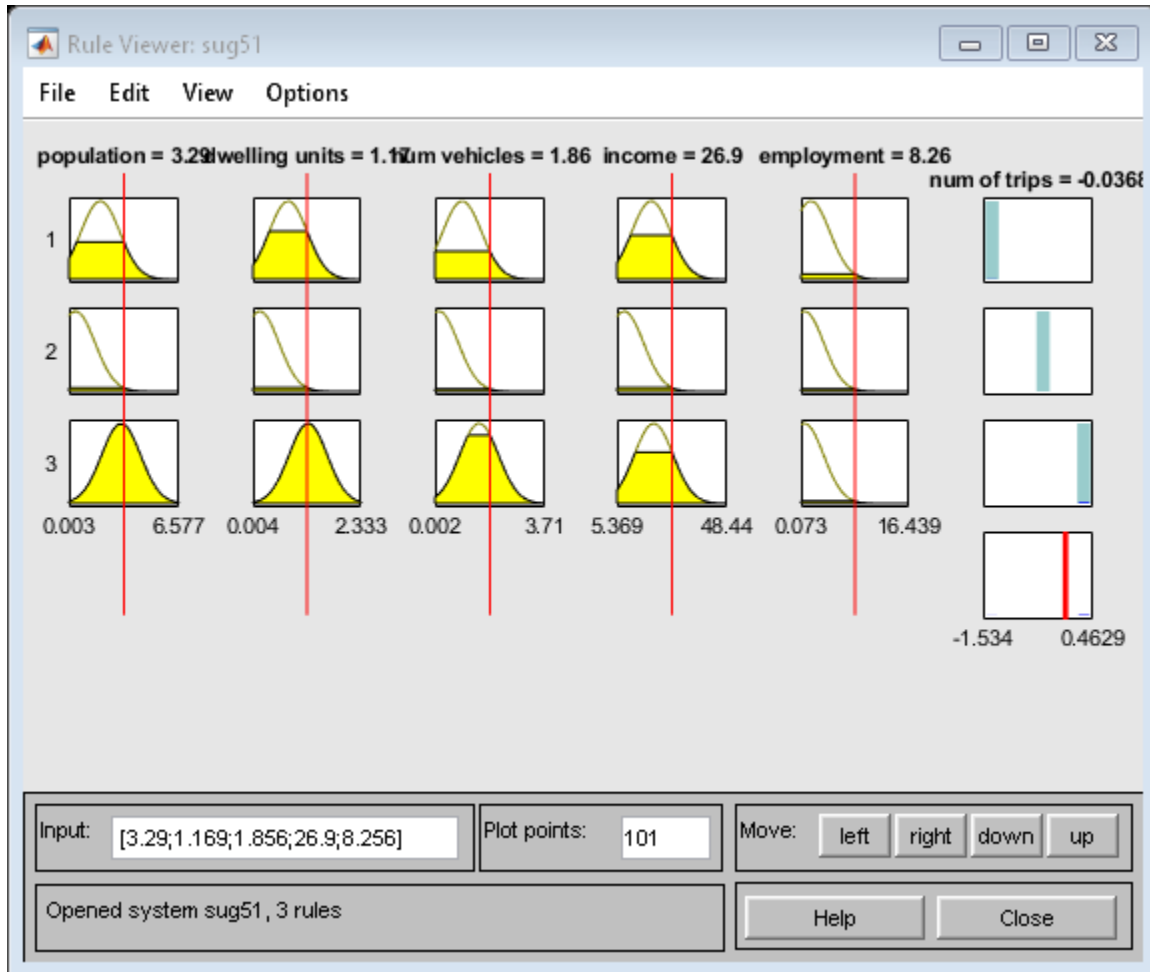


Figure 7: Rule viewer that simulates the entire fuzzy inference process

ruleview is the graphical simulator for simulating the FIS response for specific values of the input variables. Now, having built the fuzzy system, if we want to understand how

many trips will occur for a particular demographic setup, say an area with a particular population, a certain number of dwelling units and so on, this tool will help you simulate the FIS response for the input of your choice.

Another feature of this GUI tool is, it gives you a snapshot of the entire fuzzy inference process, right from how the membership functions are being satisfied in every rule to how the final output is being generated through defuzzification.

Conclusion

This example has attempted to convey how clustering and fuzzy logic can be employed as effective techniques for data modeling and analysis.

Fuzzy logic has also found various applications in other areas of technology like non-linear control, automatic control, signal processing, system identification, pattern recognition, time series prediction, data mining, financial applications etc.

Reference

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

See Also

`anfis` | `subclust`

More About

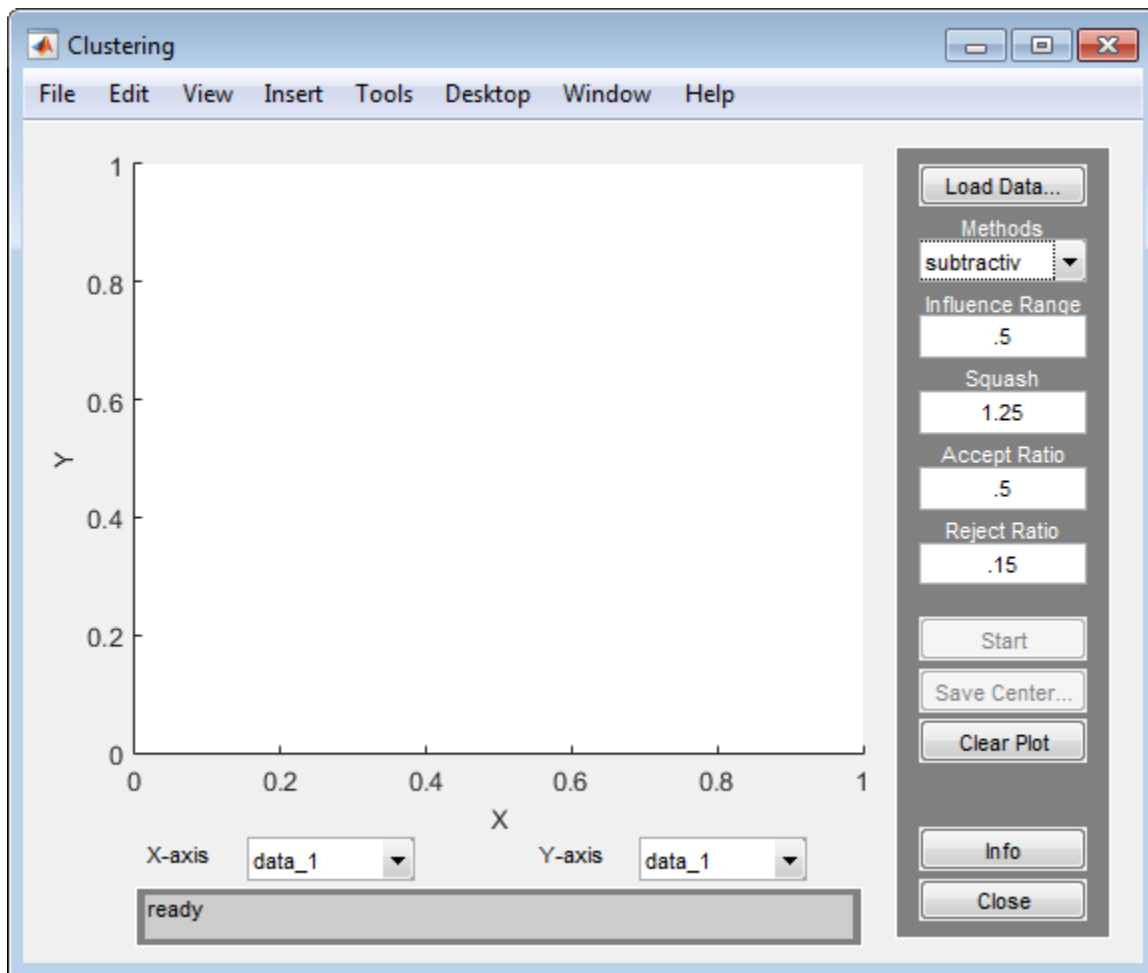
- "Fuzzy Clustering" on page 4-2

Data Clustering Using Clustering Tool

The Clustering tool implements the fuzzy data clustering functions `fcm` and `subclust`, and lets you perform clustering on data. For more information on the clustering methods, see “Fuzzy Clustering” on page 4-2.

To open the tool, at the MATLAB command line, type:

```
findcluster
```



Use the Clustering tool to perform the following tasks:

- 1 Load and plot the data.
- 2 Perform the clustering.
- 3 Save the cluster center.

Access the online help topics by clicking **Info** or using the **Help** menu.

Load and Plot Data

To load a data set, perform either of the following actions:

- Click **Load Data**, and select the file containing the data.
- Open the Clustering Tool with a data set directly by calling `findcluster` with the data set as an input argument.

The data set file must have the extension `.dat`. Each line of the data set file contains one data point. For example, if you have 5-dimensional data with 100 data points, the file contains 100 lines, and each line contains five values.

For example, enter:

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

The Clustering tool works on multidimensional data sets, but displays only two of those dimensions on the plot. To select other dimensions in the data set for plotting, you can use the drop-down lists under **X-axis** and **Y-axis**.

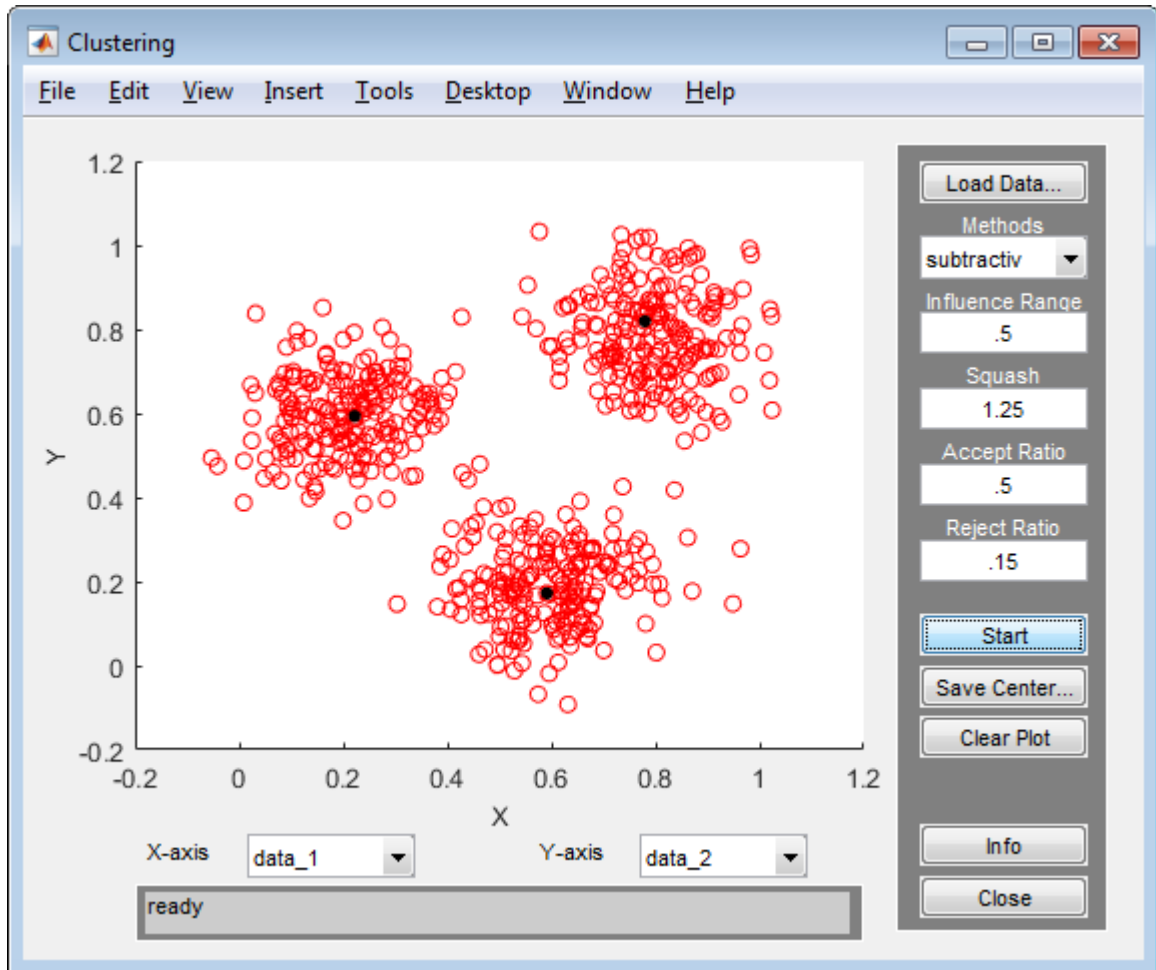
Cluster Data

To start clustering the data:

- 1 Choose the clustering function `fcm` (fuzzy C-Means clustering) or `subtractiv` (subtractive clustering) from the drop-down menu under **Methods**.
- 2 Set options for:
 - Fuzzy c-means clustering using the **Cluster Num**, **Max Iteration**, **Min**, and **Exponent** fields. For information on these options, see `fcm`.
 - Subtractive clustering using the **Influence Range**, **Squash**, **Aspect Ratio**, and **Reject Ratio** fields. For information on these options, see `subclust`.

3 Cluster the data by clicking **Start**.

Once the clustering is complete, the cluster centers appear in black as shown in the next figure.



Tip Using the Clustering tool, you can obtain only the computed cluster centers. To obtain additional information for:

- Fuzzy c-means clustering, such as the fuzzy partition matrix, cluster the data using `fcm`.
- Subtractive clustering, such as the range of influence in each data dimension, cluster the data using `subclust`.

To use the same clustering data with either `fcm` or `subclust`, first load the data file into the MATLAB workspace. For example, at the MATLAB command line, type:

```
load clusterdemo.dat
```

Save Cluster Centers

To save the cluster centers, click **Save Center**.

See Also

`fcm` | `findcluster` | `subclust`

More About

- “Fuzzy Clustering” on page 4-2

Fuzzy Logic in Simulink

- “Simulate Fuzzy Inference Systems in Simulink” on page 5-2
- “Water Level Control in a Tank” on page 5-15
- “Temperature Control in a Shower” on page 5-23
- “Implement Fuzzy PID Controller in Simulink Using Lookup Table” on page 5-30

Simulate Fuzzy Inference Systems in Simulink

You can simulate a fuzzy inference system (FIS) in Simulink using either the Fuzzy Logic Controller or Fuzzy Logic Controller with Ruleviewer blocks. Alternatively, you can evaluate fuzzy systems at the command line using `evalfis`.

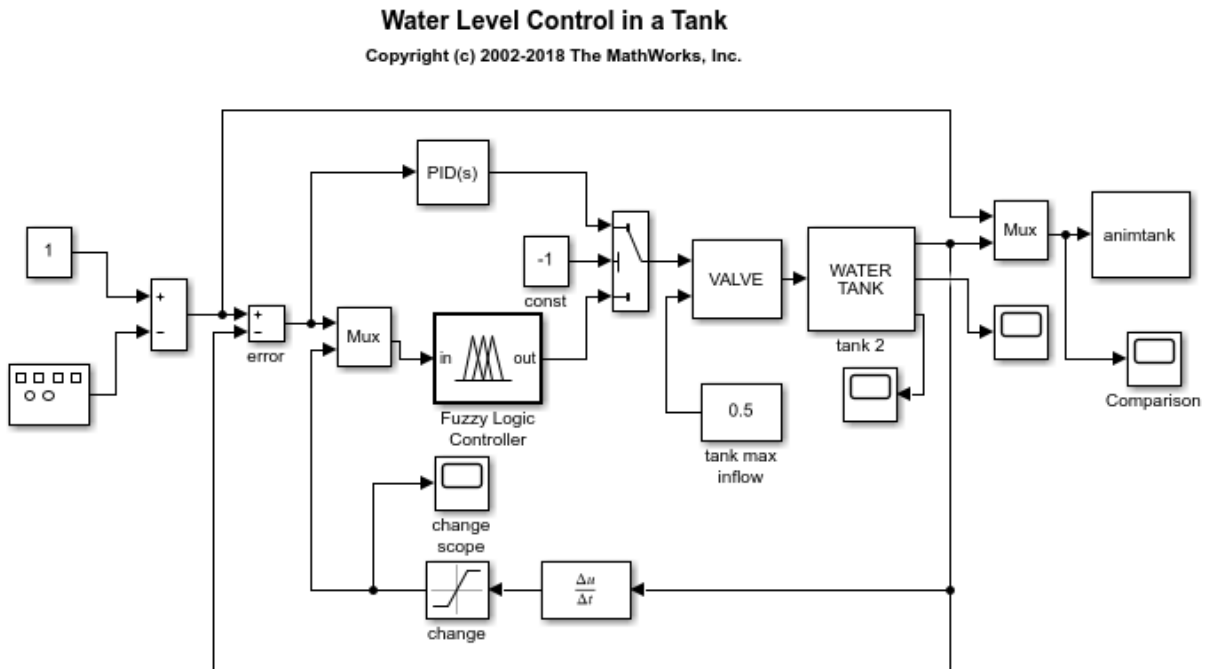
For more information on creating fuzzy inference systems, see “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14 and “Build Fuzzy Systems at the Command Line” on page 2-38.

Simulate Fuzzy Inference System

Once you have implemented a fuzzy inference system using **Fuzzy Logic Designer**, using **Neuro-Fuzzy Designer**, or at the command line, you can simulate the system in Simulink.

For this example, you control the level of water in a tank using a fuzzy inference system implemented using a Fuzzy Logic Controller block. Open the `sltank` model.

```
open_system('sltank')
```

For this system, you control the water that flows into the tank using a valve. The outflow rate depends on the diameter of the output pipe, which is constant, and the pressure in the tank, which varies with water level. Therefore, the system has nonlinear characteristics.

The two inputs to the fuzzy system are the water level error, `level`, and the rate of change of the water level, `rate`. The output of the fuzzy system is the rate at which the control valve is opening or closing, `valve`.

To implement a fuzzy inference system, specify the **FIS name** parameter of the Fuzzy Logic Controller block as the name of a FIS object in the MATLAB® workspace. In this example, the block uses the `mamfis` object `tank`.

For more information on this system, see “Water Level Control in a Tank” on page 5-15.

As a first attempt to control the water level, set the following rules in the FIS. These rules adjust the valve based on only the water level error.

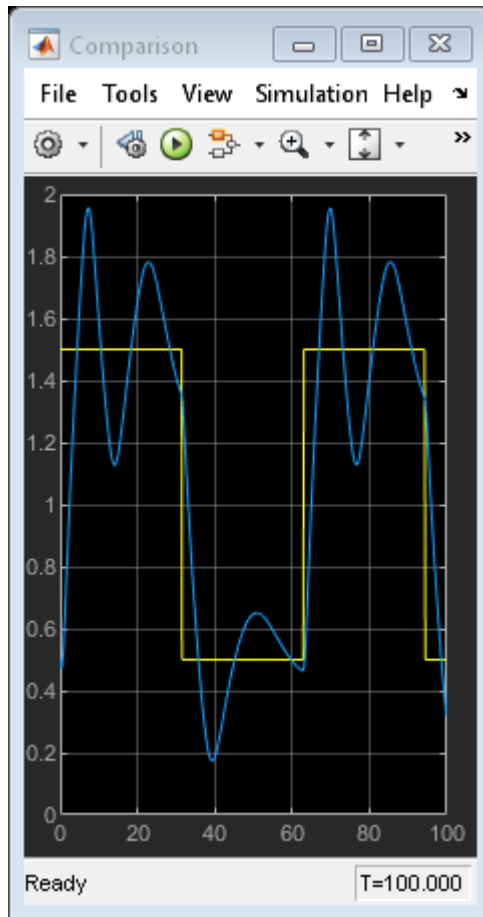
- If the water level is okay, then do not adjust the valve.
- If the water level is low, then open the valve quickly.
- If the water level is high, then close the valve quickly.

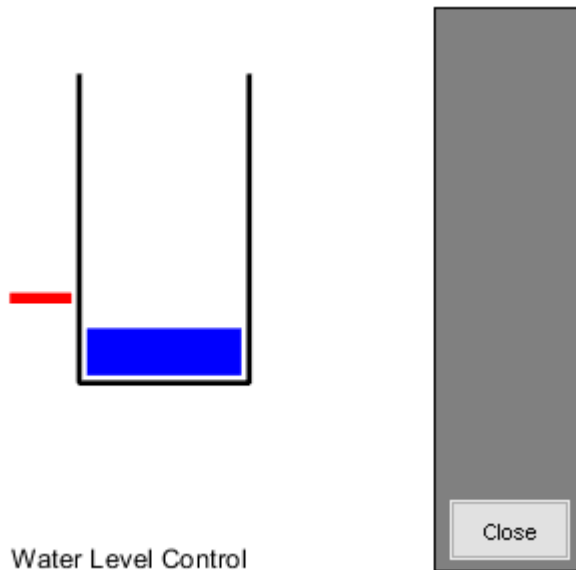
Specify the rules by creating a vector of `fisrule` objects and assigning it to the `Rules` property of the `tank` FIS object.

```
rule1 = "If level is okay then valve is no_change";  
rule2 = "If level is low then valve is open_fast";  
rule3 = "If level is high then valve is close_fast";  
rules = [rule1 rule2 rule3];  
tank.Rules = fisrule(rules);
```

Simulate the model, and view the water level.

```
open_system('sltank/Comparison')  
sim('sltank',100)
```





These rules are insufficient for controlling the system, since the water level oscillates around the setpoint.

To reduce the oscillations, add two more rules to the system. These rules adjust the valve based on the rate of change of the water level when the water level is near the setpoint.

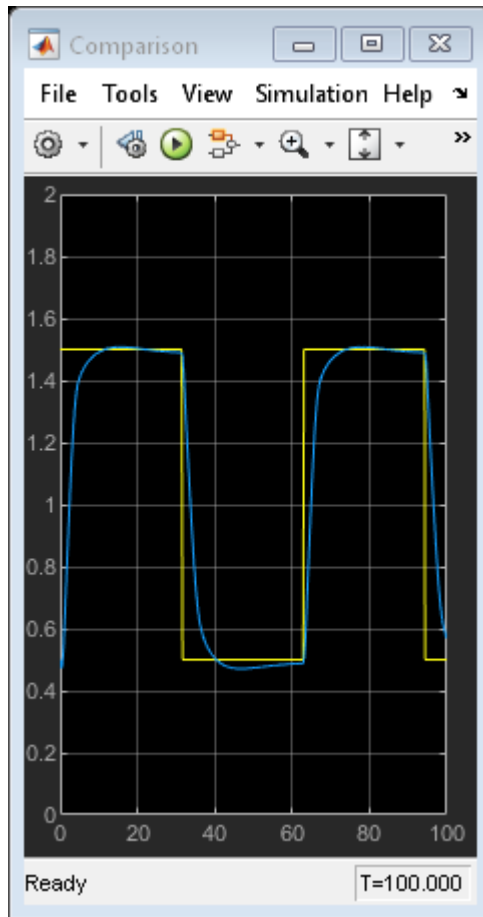
- If the water level is okay and increasing, then close the valve slowly.
- If the water level is okay and decreasing, then open the valve slowly.

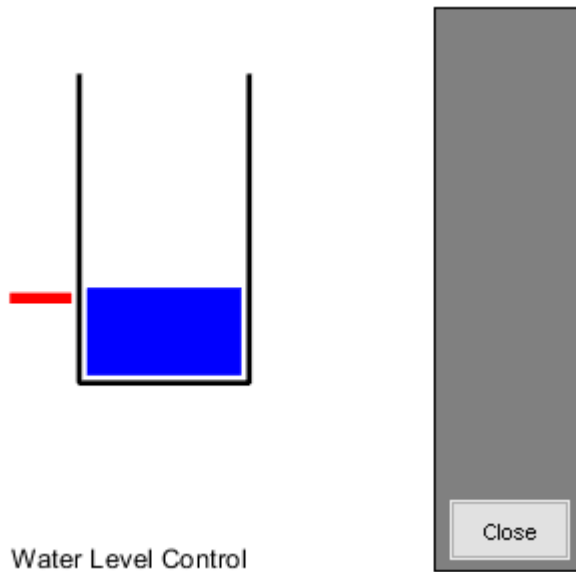
To add these rules, use the `addRule` function.

```
rule4 = "If level is okay and rate is positive then valve is close_slow";  
rule5 = "If level is okay and rate is negative then valve is open_slow";  
newRules = [rule4 rule5];  
tank = addRule(tank,newRules);
```

Simulate the model.

```
sim('sltank',100)
```





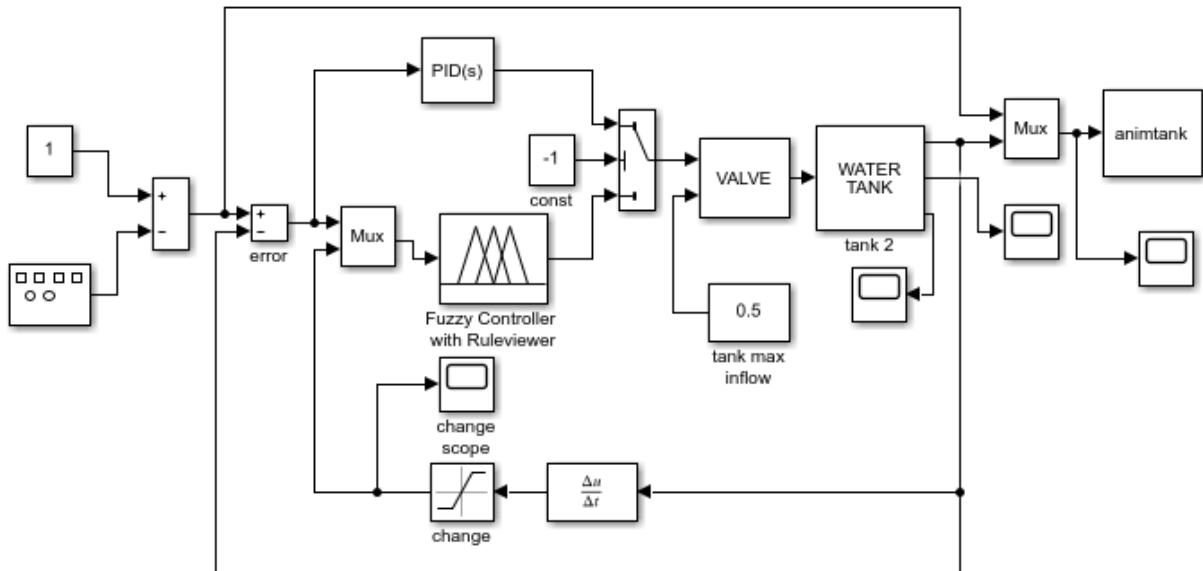
The water level now tracks the setpoint without oscillating.

You can also simulate fuzzy systems using the Fuzzy Logic Controller with Ruleviewer block. The `sltankrule` model is the same as the `sltank` model, except that it uses the Fuzzy Logic Controller with Ruleviewer block.

```
open_system('sltankrule')
```

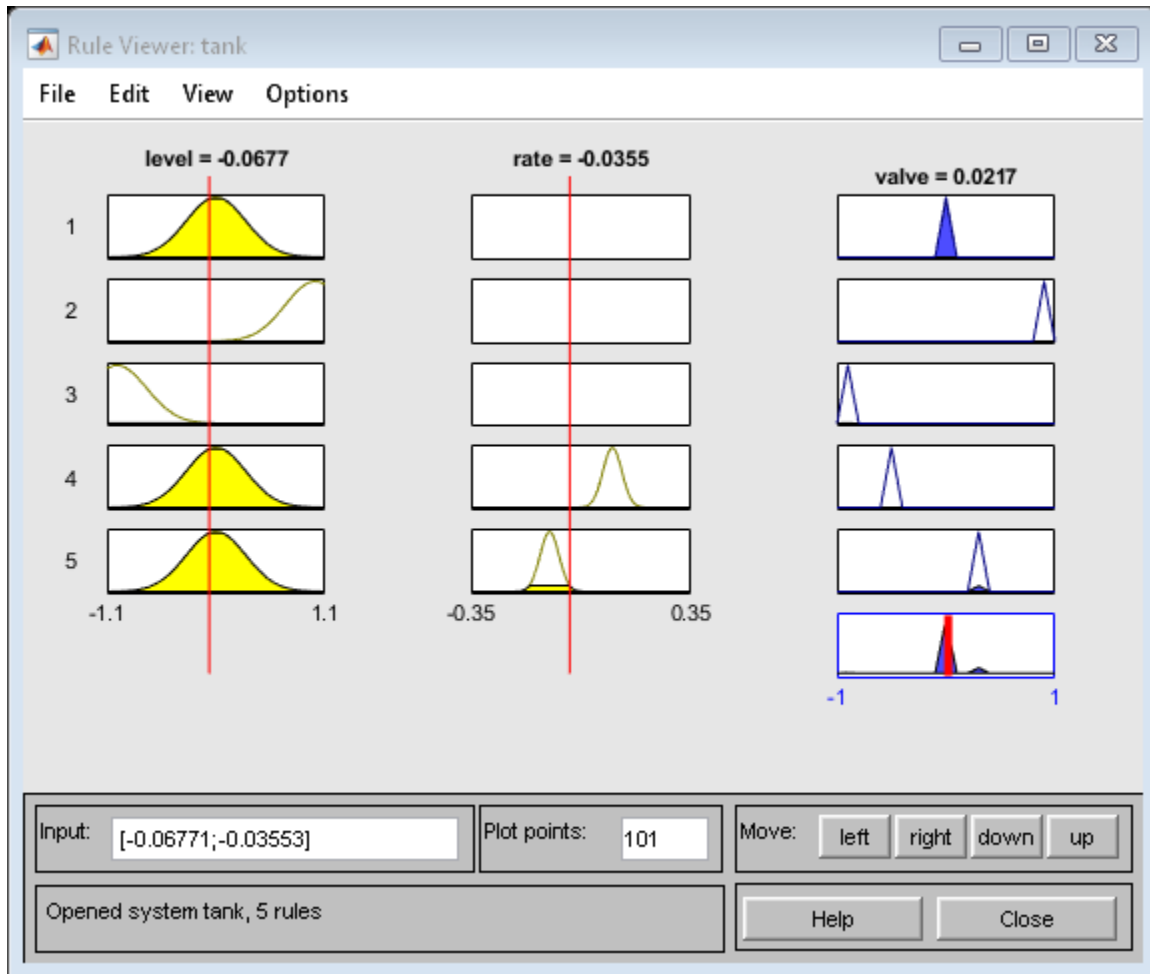
Water Level Control in a Tank

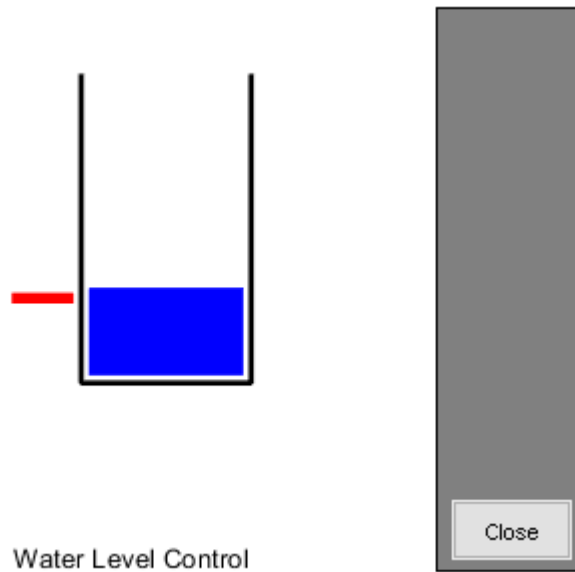
Copyright (c) 2002-2018 The MathWorks, Inc.



During simulation, this block displays the Rule Viewer from the **Fuzzy Logic Designer** app.

```
sim('sltankrule',100)
```





If you pause the simulation, you can examine the FIS behavior by manually adjusting the input variable values in the Rule Viewer, and observing the inference process and output.

You can also access the **Fuzzy Logic Designer** editors from the Rule Viewer. From the Rule Viewer, you can then adjust the parameters of your fuzzy system using these editors, and export the updated system to the MATLAB workspace. To simulate the updated FIS, restart the simulation. For more information on using these editors, see “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14.

Access Intermediate Fuzzy Inference Results

You can access intermediate fuzzy inference results using the Fuzzy Logic Controller block. You can use this data to visualize the fuzzy inference process or troubleshoot the performance of your FIS. To access this data, enable the corresponding parameters in the block, and connect signals to the corresponding output ports.

Block Parameter	Description	Output Port
Fuzzified Inputs	Fuzzified input values, obtained by evaluating the input membership functions of each rule at the current input values.	fi
Rule firing strengths	Rule firing strengths, obtained by evaluating the antecedent of each rule.	rfs
Rule outputs	Rule outputs, obtained by evaluating the consequent of each rule.	ro
Aggregated outputs	Aggregate output for each output variable, obtained by combining the corresponding outputs from all the rules.	ao

For more information, see Fuzzy Logic Controller.

Simulation Modes

The Fuzzy Logic Controller block has the following two simulation modes:

- **Interpreted execution** — Simulate fuzzy systems using precompiled MEX files. Using this option reduces the initial compilation time of the model.
- **Code generation** — Simulate fuzzy system without precompiled MEX files. Use this option when simulating fuzzy systems for code generation applications. Doing so simulates your system using the same code path used for generated code.

To select a simulation mode, set the **Simulate using** parameter of the block. By default, the block uses **Interpreted execution** mode for simulation.

Map Command-Line Functionality to Fuzzy Logic Controller Block

The parameters and ports of the Fuzzy Logic Controller block map to the input and output arguments of `evalfis` or the properties of `evalfisOptions`. The following table shows the block parameters and ports that map to `evalfis` arguments.

evalfis Argument	Description	Block Parameter or Port
fis	Fuzzy inference system	FIS name

evalfis Argument	Description	Block Parameter or Port
input, when a single row	Input variable values	in
output, when a single row	Output variable values	out
fuzzifiedIn	Fuzzified inputs	fi
ruleOut	Rule outputs	ro
aggregateOut	Aggregated outputs	ao
ruleFiring	Rule firing strengths	rfs

The following table shows the block parameters that map to `evalfisOptions` properties.

evalfisOptions Property	Description	Block Parameter or Port
NumSamplePoints	Number of points in output fuzzy sets	Number of samples for output discretization
OutOfRangeInputValueMessage	Diagnostic message behavior when an input is out of range	Out of range input value
NoRuleFiredMessage	Diagnostic message behavior when no rules fire	No rule fired
EmptyOutputFuzzySetMessage	Diagnostic message behavior when an output fuzzy set is empty	Empty output fuzzy set

The remaining parameters of the Fuzzy Logic Controller block do not map to arguments of `evalfis`. Also, unlike the Fuzzy Logic Controller, `evalfis` does not support fixed-point data for simulation or code generation.

See Also

Blocks

Fuzzy Logic Controller | Fuzzy Logic Controller with Ruleviewer

More About

- “Temperature Control in a Shower” on page 5-23

- “Water Level Control in a Tank” on page 5-15

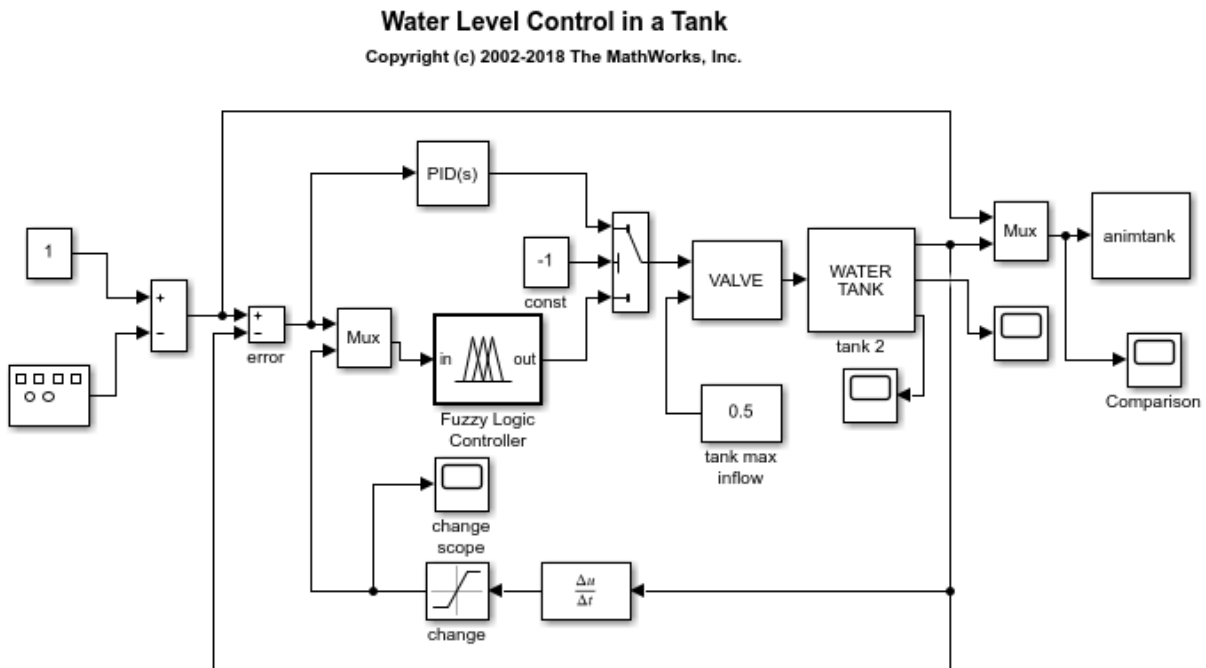
Water Level Control in a Tank

This model shows how to implement a fuzzy inference system (FIS) in a Simulink® model.

Simulink Model

This model controls the level of water in a tank using a fuzzy inference system implemented using a Fuzzy Logic Controller block. Open the `sltank` model.

```
open_system('sltank')
```



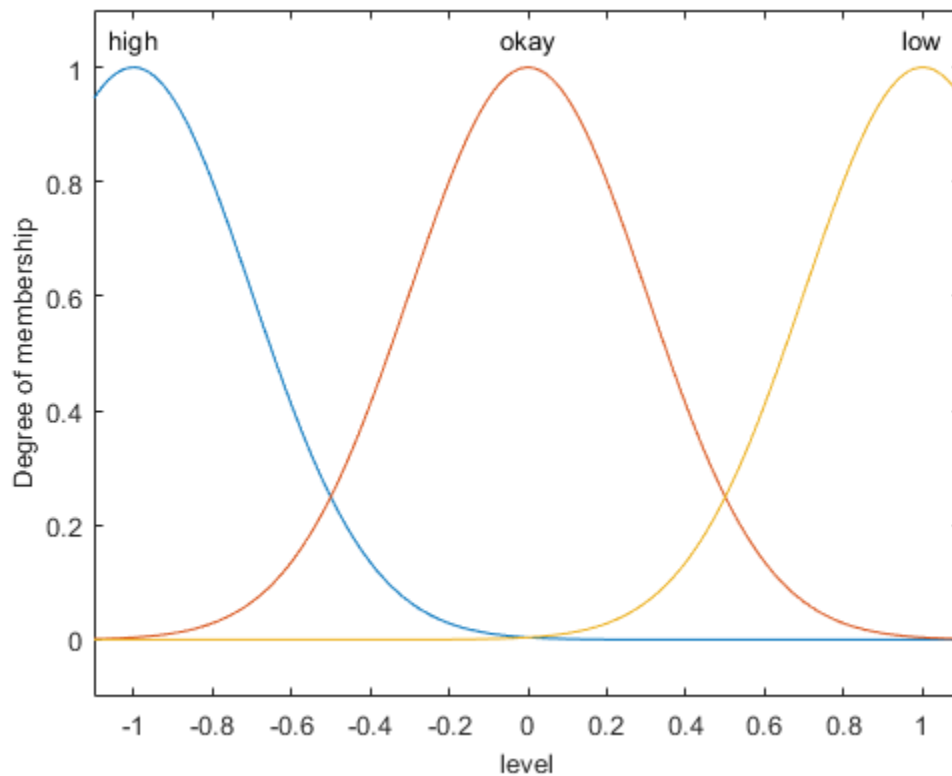
For this system, you control the water that flows into the tank using a valve. The outflow rate depends on the diameter of the output pipe, which is constant, and the pressure in the tank, which varies with water level. Therefore, the system has nonlinear characteristics.

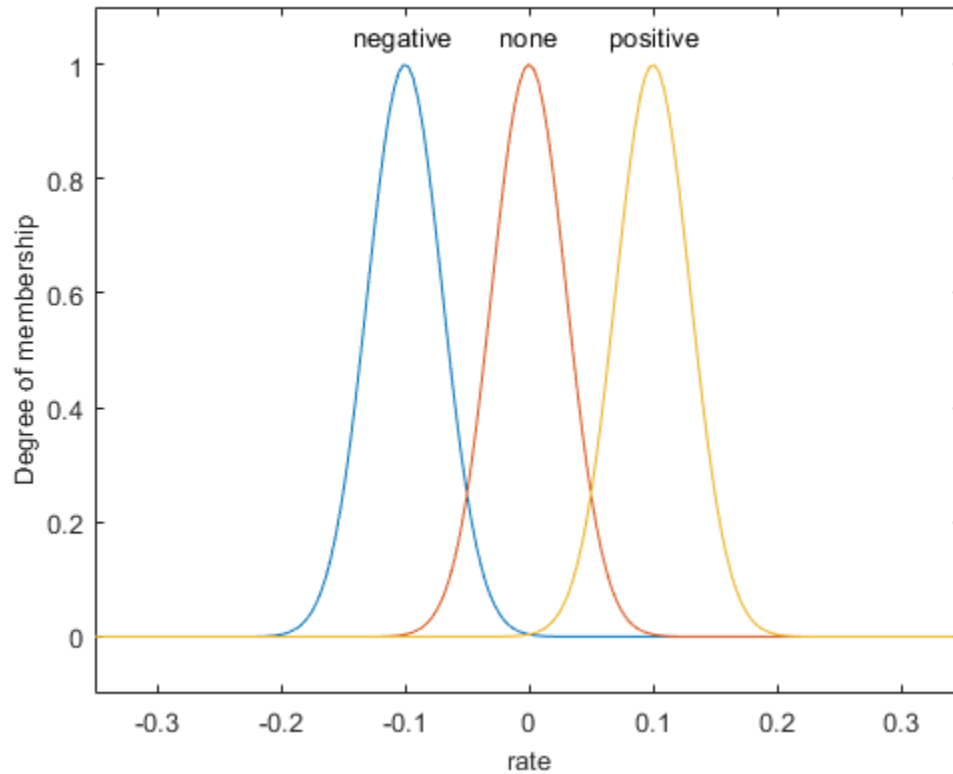
Fuzzy Inference System

The fuzzy system is defined in a FIS object, `tank`, in the MATLAB® workspace. For more information on how to specify a FIS in a Fuzzy Logic Controller block, see Fuzzy Logic Controller.

The two inputs to the fuzzy system are the water level error, `level`, and the rate of change of the water level, `rate`. Each input has three membership functions.

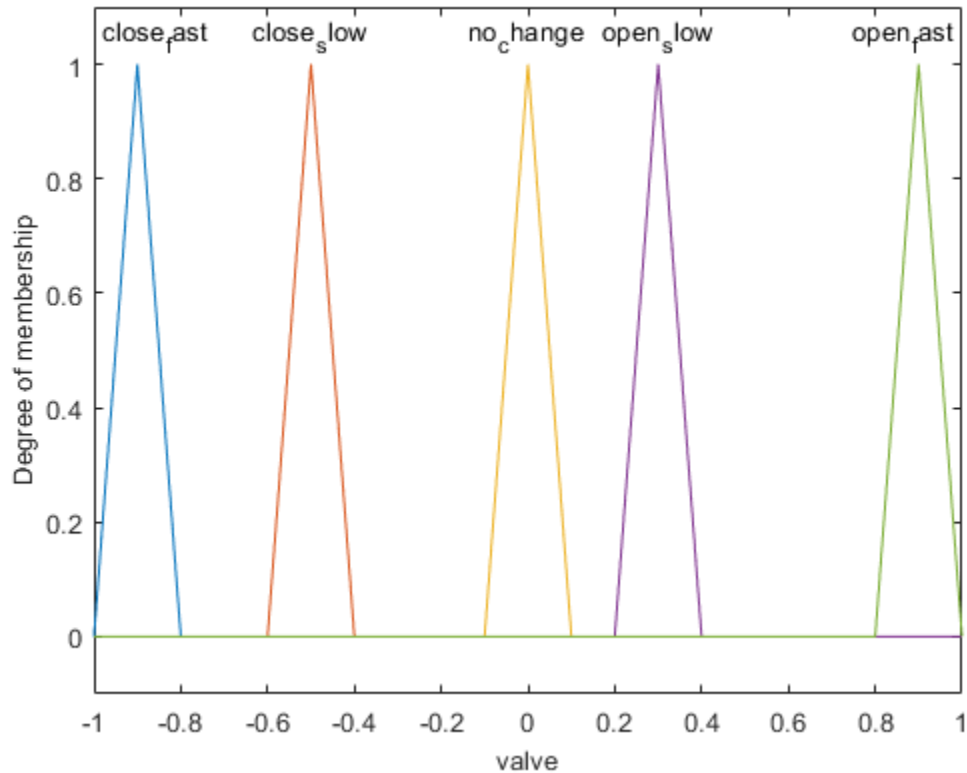
```
figure  
plotmf(tank, 'input', 1)  
figure  
plotmf(tank, 'input', 2)
```





The output of the fuzzy system is the rate at which the control valve is opening or closing, valve, which has five membership functions.

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



Due to the diameter of the outflow pipe, the water tank in this system empties more slowly than it fills up. To compensate for this imbalance, the `close_slow` and `open_slow` valve membership functions are not symmetrical. A PID controller does not support such asymmetry.

The fuzzy system has five rules. The first three rules adjust the valve based on only the water level error.

- If the water level is okay, then do not adjust the valve.
- If the water level is low, then open the valve quickly.
- If the water level is high, then close the valve quickly.

The other two rules adjust the valve based on the rate of change of the water level when the water level is near the setpoint.

- If the water level is okay and increasing, then close the valve slowly.
- If the water level is okay and decreasing, then open the valve slowly.

```
tank.Rules
```

```
ans =
```

```
1x5 fisrule array with properties:
```

```

Description
Antecedent
Consequent
Weight
Connection
```

```
Details:
```

```
Description
```

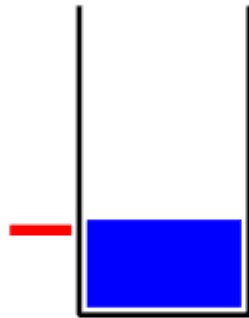
	Description
1	"level==okay => valve=no_change (1)"
2	"level==low => valve=open_fast (1)"
3	"level==high => valve=close_fast (1)"
4	"level==okay & rate==positive => valve=close_slow (1)"
5	"level==okay & rate==negative => valve=open_slow (1)"

In this model, you can also control the water level using a PID controller. To switch to the PID controller, set the const block to a value greater than or equal to zero.

Simulation

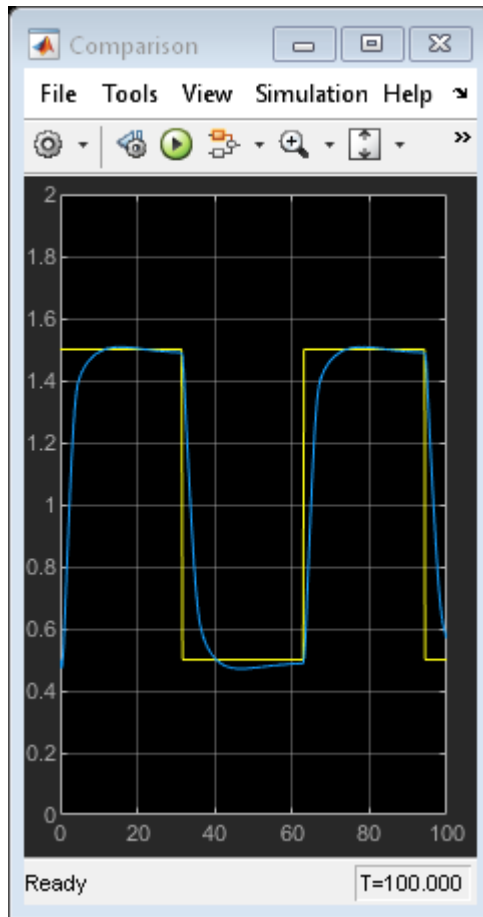
The model simulates the controller with periodic changes in the setpoint of the water level. Run the simulation.

```
sim('sltank',100)
open_system('sltank/Comparison')
```



Water Level Control





The water level tracks the setpoint well. You can adjust the performance of the controller by modifying the rules of the tank FIS. For example, if you remove the last two rules, which are analogous to a derivative control action, the controller performs poorly, with large oscillations in the water level.

See Also

Blocks

Fuzzy Logic Controller | Fuzzy Logic Controller with Ruleviewer

More About

- “Simulate Fuzzy Inference Systems in Simulink” on page 5-2
- “Temperature Control in a Shower” on page 5-23

Temperature Control in a Shower

This model shows how to implement a fuzzy inference system (FIS) in a Simulink® model.

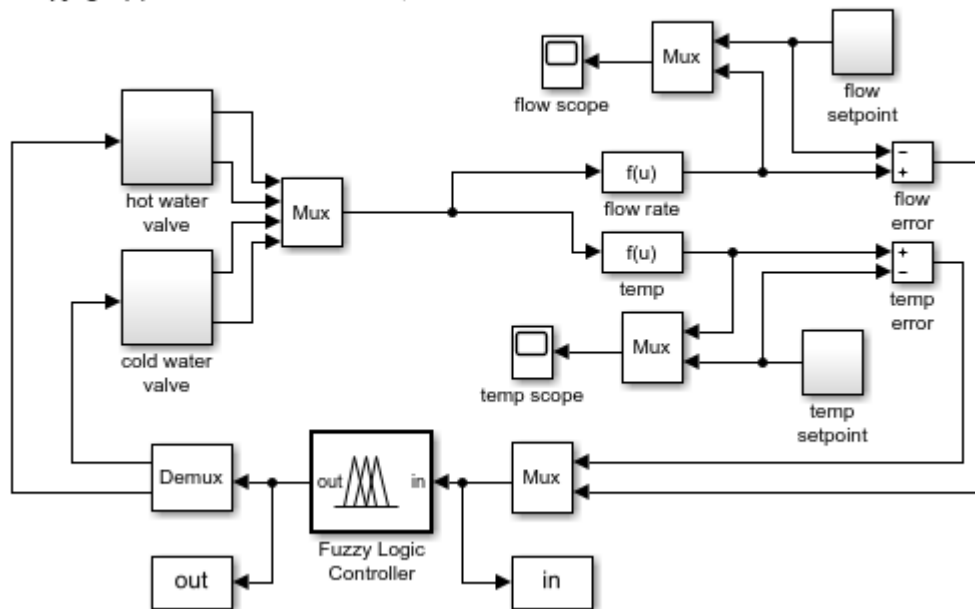
Simulink Model

The model controls the temperature of a shower using a fuzzy inference system implemented using a Fuzzy Logic Controller block. Open the shower model.

```
open_system('shower')
```

Temperature Control in a Shower

Copyright (c) 2002-2018 The MathWorks, Inc.



For this system, you control the flow rate and temperature of a shower by adjusting hot and cold water valves.

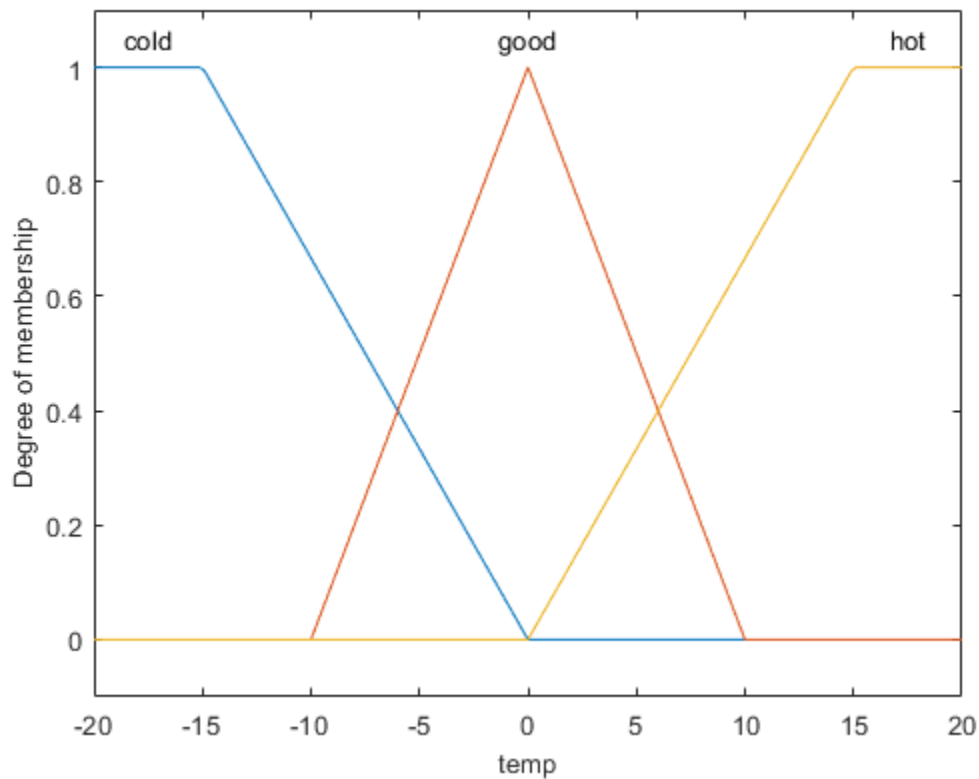
Since there are two inputs for the fuzzy system, the model concatenates the input signals using a Mux block. The output of the Mux block is connected to the input of the Fuzzy Logic Controller block. Similarly, the two output signals are obtained using a Demux block connected to the controller.

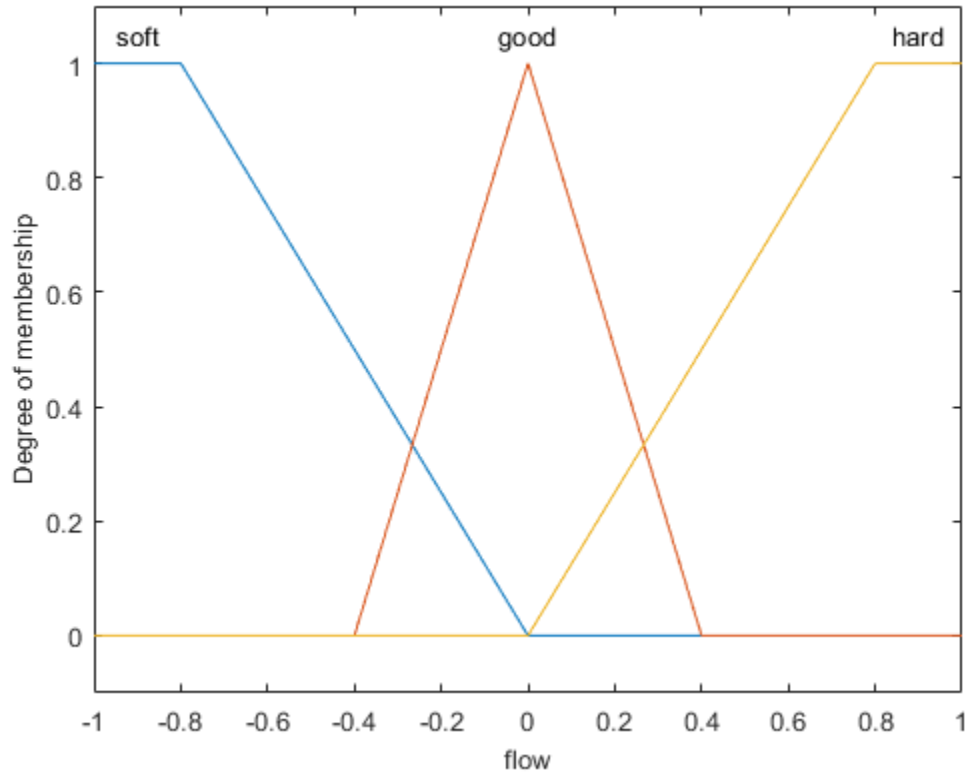
Fuzzy Inference System

The fuzzy system is defined in a FIS object, `fisMatrix`, in the MATLAB® workspace. For more information on how to specify a FIS in a Fuzzy Logic Controller block, see Fuzzy Logic Controller.

The two inputs to the fuzzy system are the temperature error, `temp`, and the flow rate error, `flow`. Each input has three membership functions.

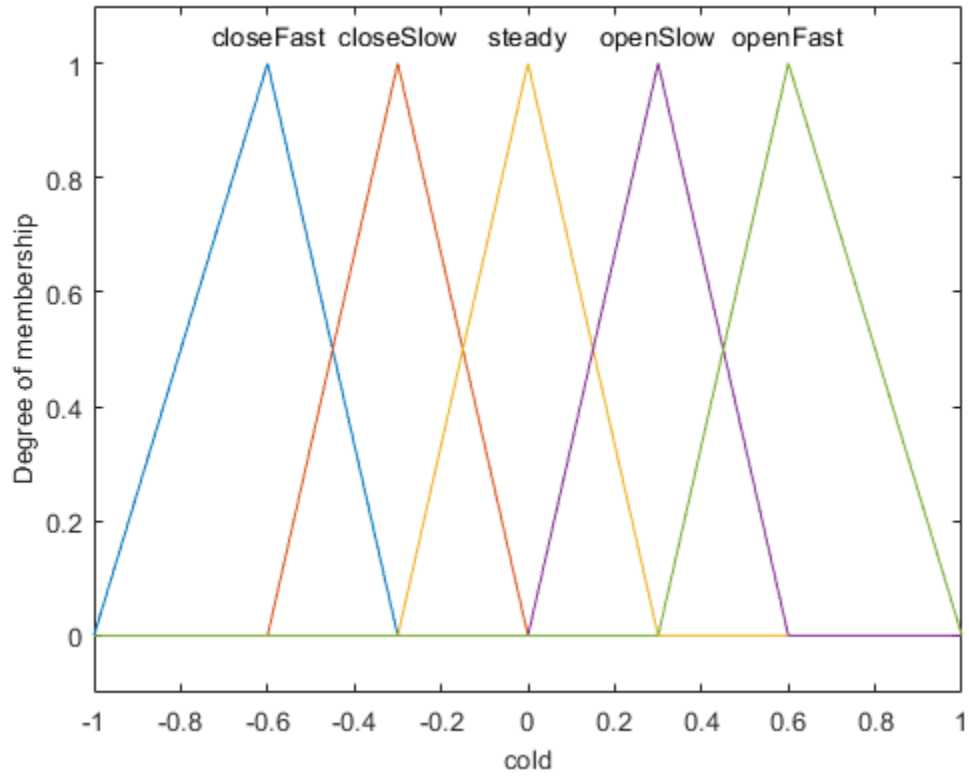
```
figure
plotmf(fisMatrix,'input',1)
figure
plotmf(fisMatrix,'input',2)
```

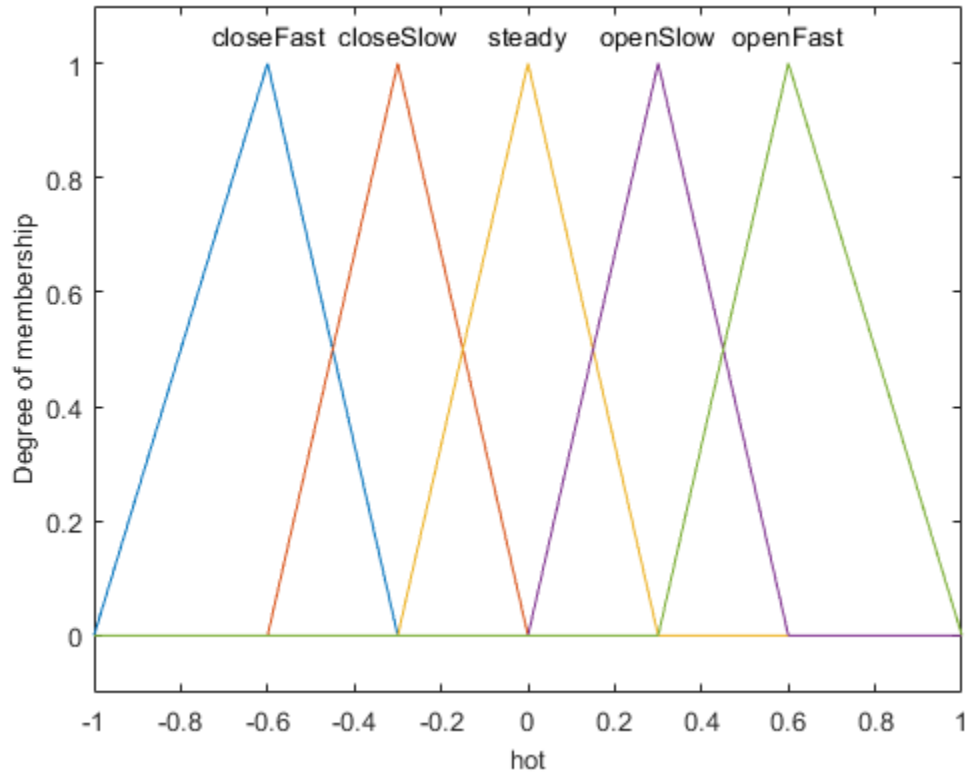




The two outputs of the fuzzy system are the rate at which the cold and hot water valves are opening or closing, cold and hot respectively. Each output has five membership functions.

```
figure
plotmf(fisMatrix, 'output', 1)
figure
plotmf(fisMatrix, 'output', 2)
```





The fuzzy system has nine rules for adjusting the hot and cold water valves based on the flow and temperature errors. The rules adjust the total flow rate based on the flow error, and adjust the relative hot and cold flow rates based on the temperature error.

```
fisMatrix.Rules
```

```
ans =
```

```
1x9 fisrule array with properties:
```

```
Description
Antecedent
Consequent
```

Weight
Connection

Details:

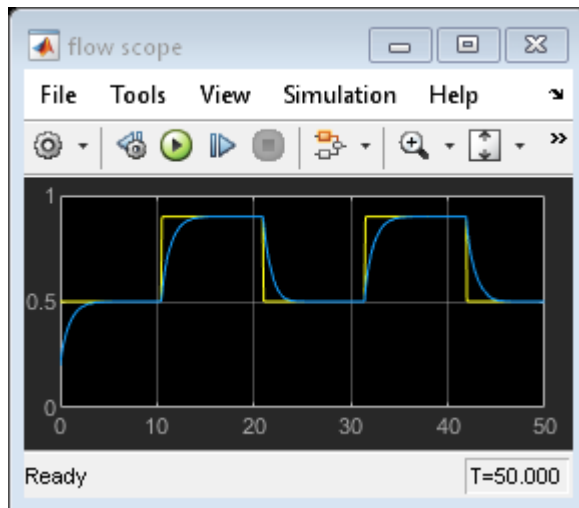
Description

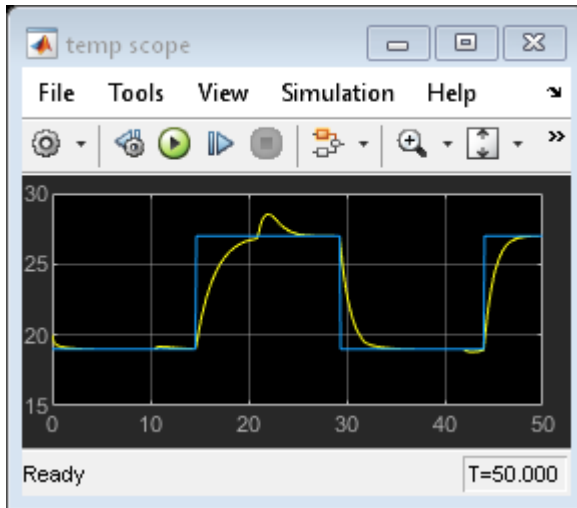
	Description
1	"temp==cold & flow==soft => cold=openSlow, hot=openFast (1)"
2	"temp==cold & flow==good => cold=closeSlow, hot=openSlow (1)"
3	"temp==cold & flow==hard => cold=closeFast, hot=closeSlow (1)"
4	"temp==good & flow==soft => cold=openSlow, hot=openSlow (1)"
5	"temp==good & flow==good => cold=steady, hot=steady (1)"
6	"temp==good & flow==hard => cold=closeSlow, hot=closeSlow (1)"
7	"temp==hot & flow==soft => cold=openFast, hot=openSlow (1)"
8	"temp==hot & flow==good => cold=openSlow, hot=closeSlow (1)"
9	"temp==hot & flow==hard => cold=closeSlow, hot=closeFast (1)"

Simulation

The model simulates the controller with periodic changes in the setpoints of the water temperature and flow rate.

```
set_param('shower/flow scope', 'Open', 'on', 'Ymin', '0', 'Ymax', '1')
set_param('shower/temp scope', 'Open', 'on', 'Ymin', '15', 'Ymax', '30')
sim('shower', 50)
```





The flow rate tracks the setpoint well. The temperature also tracks its setpoint, though there are temperature deviations when the controller adjusts to meet a new flow setpoint.

```
bdclose('shower') % Closing model also clears its workspace variables.
```

See Also

Blocks

Fuzzy Logic Controller

More About

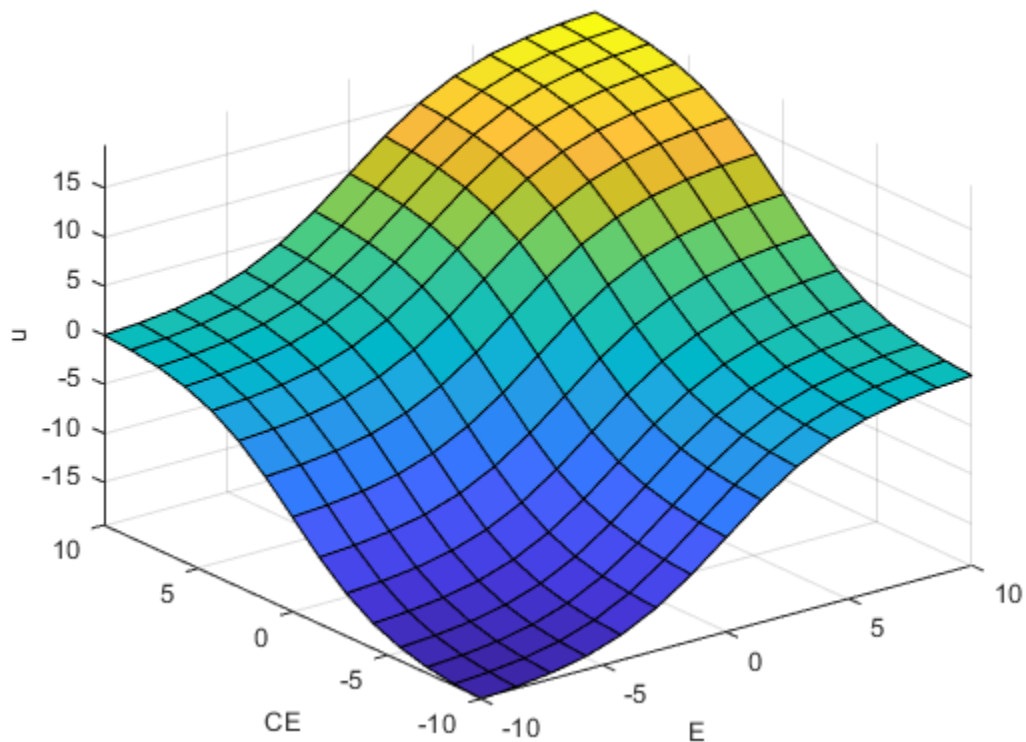
- “Simulate Fuzzy Inference Systems in Simulink” on page 5-2
- “Water Level Control in a Tank” on page 5-15

Implement Fuzzy PID Controller in Simulink Using Lookup Table

This example shows how to implement a fuzzy inference system for nonlinear PID control using a 2-D Lookup Table block.

Overview

A fuzzy inference system (FIS) maps given inputs to outputs using fuzzy logic. For example, a typical mapping of a two-input, one-output fuzzy controller can be depicted in a 3-D plot. The plot is often referred to as a *control surface* plot.



For control applications, typical FIS inputs are the error ($e(k)$) and change of error ($e(k) - e(k-1)$), E and CE respectively in the control surface plot. The FIS output is the control action inferred from the fuzzy rules, u in the surface plot. Fuzzy Logic Toolbox™ provides commands and apps for designing a FIS for a desired control surface. You can then simulate the designed FIS using the Fuzzy Logic Controller block in Simulink®.

You can often approximate nonlinear control surfaces using lookup tables to simplify the generated code and improve execution speed. For example, you can replace a Fuzzy Logic Controller block in Simulink with a set of Lookup Table blocks, one table for each output defined in the FIS. You can compute the data used in the lookup table using the `evalfis` command.

For this example, you design a nonlinear fuzzy PID controller for a plant in Simulink. The plant is a single-input, single-output system in discrete time. The design goal is to achieve good reference tracking performance.

```
Ts = 0.1;  
Plant = c2d(zpk([], [-1 -3 -5], 1), Ts);
```

You also implement the fuzzy inference system using a 2-D lookup table that approximates the control surface and achieves the same control performance.

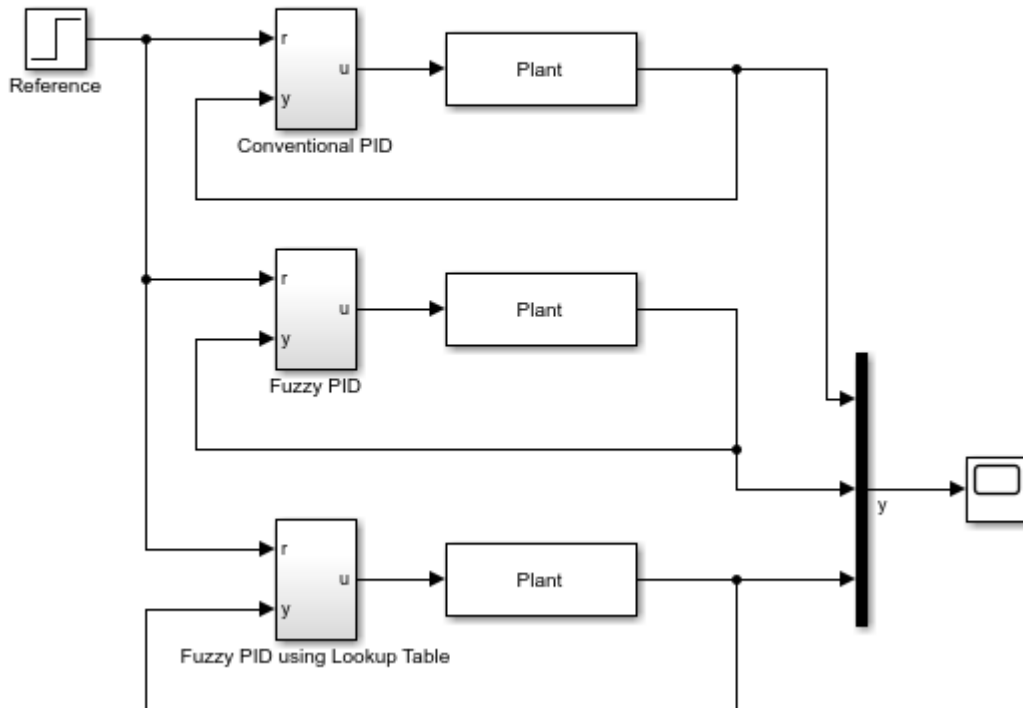
Fuzzy PID Controller Structure

The fuzzy controller in this example is in the feedback loop and computes PID-like actions using fuzzy inference. Open the Simulink model.

```
open_system('sllookuptable')
```

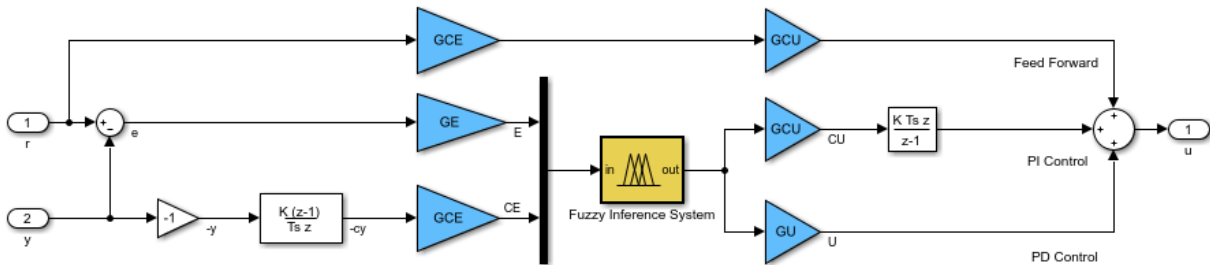
Using Lookup Table to Implement a Fuzzy PID Controller

Copyright (c) 2002-2018 The MathWorks, Inc.



The fuzzy PID controller uses a parallel structure as shown in the Fuzzy PID subsystem. For more information, see [1]. The controller is a combination of fuzzy PI control and fuzzy PD control.

```
open_system('sllookuptable/Fuzzy PID')
```

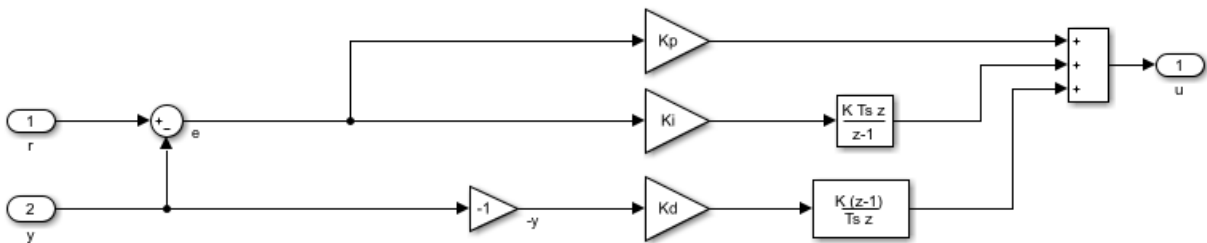


The fuzzy PID controller uses the change of the output $-(y(k) - y(k-1))$, instead of change of error $e(k) - e(k-1)$, as the second input signal to the FIS. Doing so prevents the step change in reference signal from directly triggering the derivative action. The two gain blocks, GCE and GCU, in the feed forward path from r to u , ensure that the error signal e is used in proportional action when the fuzzy PID controller is linear.

Design Conventional PID Controller

The conventional PID controller in this example is a discrete-time PID controller with Backward Euler numerical integration in both the integral and derivative actions. The controller gains are K_p , K_i , and K_d .

```
open_system('sllookuptable/Conventional PID')
```



Similar to the fuzzy PID controller, the input signal to the derivative action is $-y(k)$, instead of $e(k)$.

You can tune the PID controller gains manually or using tuning formulas. In this example, obtain the initial PID design using the `piddtune` command from Control System Toolbox™.

Define the PID structure, tune the controller, and extract the PID gains.

```
C0 = pid(1,1,1,'Ts',Ts,'IF','B','DF','B');
C = pidtune(Plant,C0)
[Kp,Ki,Kd] = piddata(C);
```

C =

$$Kp + Ki * \frac{Ts*z}{z-1} + Kd * \frac{z-1}{Ts*z}$$

with Kp = 30.6, Ki = 25.2, Kd = 9.02, Ts = 0.1

Sample time: 0.1 seconds
Discrete-time PID controller in parallel form.

Design Equivalent Linear Fuzzy PID Controller

By configuring the FIS and selecting the four scaling factors, you can obtain a linear fuzzy PID controller that reproduces the control performance of the conventional PID controller.

First, configure the fuzzy inference system so that it produces a linear control surface from inputs E and CE to output u. The FIS settings are based on design choices described in [2]:

- Use a Sugeno style fuzzy inference system with default inference methods.
- Normalize the ranges of both inputs to [-10 10].
- Use triangular input membership functions that overlap their neighbor functions at a membership value of 0.5.
- Use an output range of [-20 20].
- Use constant output membership functions.

Construct the fuzzy inference system.

```
FIS = sugfis;
```

Define input variable E.

```
FIS = addInput(FIS,[-10 10],'Name','E');
FIS = addMF(FIS,'E','trimf',[-20 -10 0],'Name','Negative');
```



```
FIS = addMF(FIS, 'E', 'trimf', [-10 0 10], 'Name', 'Zero');
FIS = addMF(FIS, 'E', 'trimf', [0 10 20], 'Name', 'Positive');
```

Define input CE.

```
FIS = addInput(FIS, [-10 10], 'Name', 'CE');
FIS = addMF(FIS, 'CE', 'trimf', [-20 -10 0], 'Name', 'Negative');
FIS = addMF(FIS, 'CE', 'trimf', [-10 0 10], 'Name', 'Zero');
FIS = addMF(FIS, 'CE', 'trimf', [0 10 20], 'Name', 'Positive');
```

Define output variable u with constant membership functions.

```
FIS = addOutput(FIS, [-20 20], 'Name', 'u');
FIS = addMF(FIS, 'u', 'constant', -20, 'Name', 'LargeNegative');
FIS = addMF(FIS, 'u', 'constant', -10, 'Name', 'SmallNegative');
FIS = addMF(FIS, 'u', 'constant', 0, 'Name', 'Zero');
FIS = addMF(FIS, 'u', 'constant', 10, 'Name', 'SmallPositive');
FIS = addMF(FIS, 'u', 'constant', 20, 'Name', 'LargePositive');
```

Define the following fuzzy rules:

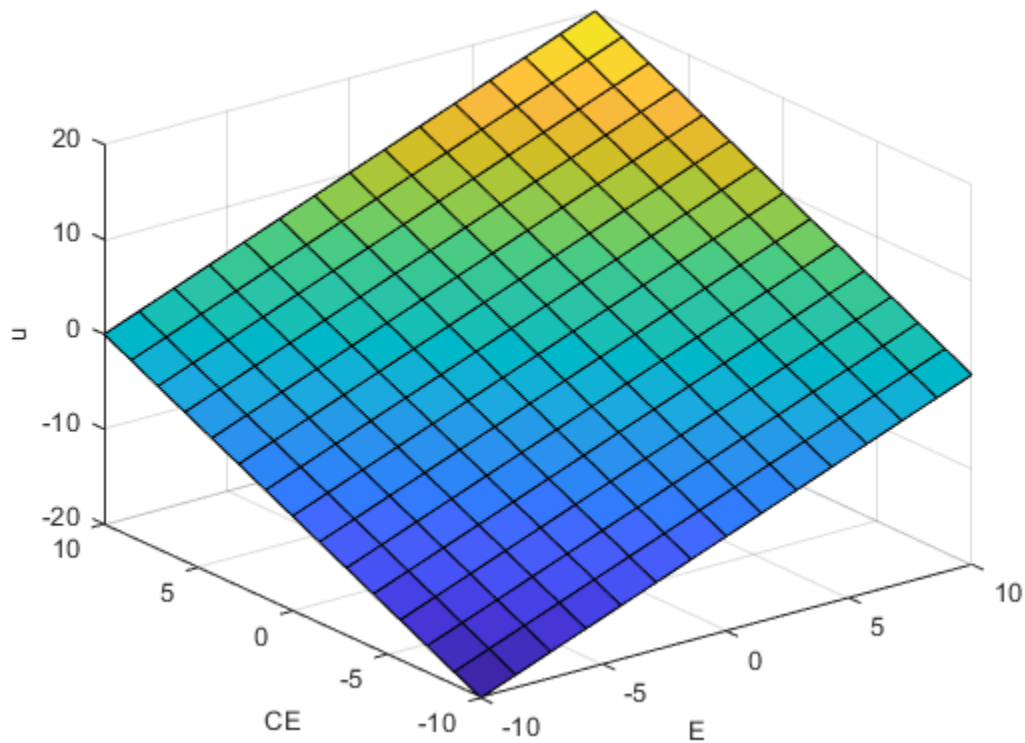
- 1 If E is negative and CE is negative, then u is -20.
- 2 If E is negative and CE is zero, then u is -10.
- 3 If E is negative and CE is positive then u is 0.
- 4 If E is zero and CE is negative, then u is -10.
- 5 If E is zero and CE is zero, then u is 0.
- 6 If E is zero and CE is positive, then u is 10.
- 7 If E is positive and CE is negative, then u is 0.
- 8 If E is positive and CE is zero, then u is 10.
- 9 If E is positive and CE is positive, then u is 20.

```
ruleList = [1 1 1 1 1; % Rule 1
            1 2 2 1 1; % Rule 2
            1 3 3 1 1; % Rule 3
            2 1 2 1 1; % Rule 4
            2 2 3 1 1; % Rule 5
            2 3 4 1 1; % Rule 6
            3 1 3 1 1; % Rule 7
            3 2 4 1 1; % Rule 8
            3 3 5 1 1]; % Rule 9
FIS = addRule(FIS, ruleList);
```

While you implement your FIS from the command line in this example, you can alternatively build your FIS using the **Fuzzy Logic Designer** app.

Plot the linear control surface.

```
gensurf(FIS)
```



Determine scaling factors G_E , G_{CE} , G_{CU} , and G_U from the K_p , K_i , and K_d gains of by the conventional PID controller. Comparing the expressions of the traditional PID and the linear fuzzy PID, the variables are related as follows:

- $K_p = G_{CU} * G_{CE} + G_U * G_E$
- $K_i = G_{CU} * G_E$

- $K_d = G_U * G_{CE}$

Assume that the maximum reference step is 1, and thus the maximum error e is 1. Since the input range of E is $[-10\ 10]$, set G_E to 10. You can then solve for G_{CE} , G_{CU} , and G_U .

```
GE = 10;
GCE = GE*(Kp-sqrt(Kp^2-4*Ki*Kd))/2/Ki;
GCU = Ki/GE;
GU = Kd/GCE;
```

Implement Fuzzy Inference System Using 2-D Lookup Table

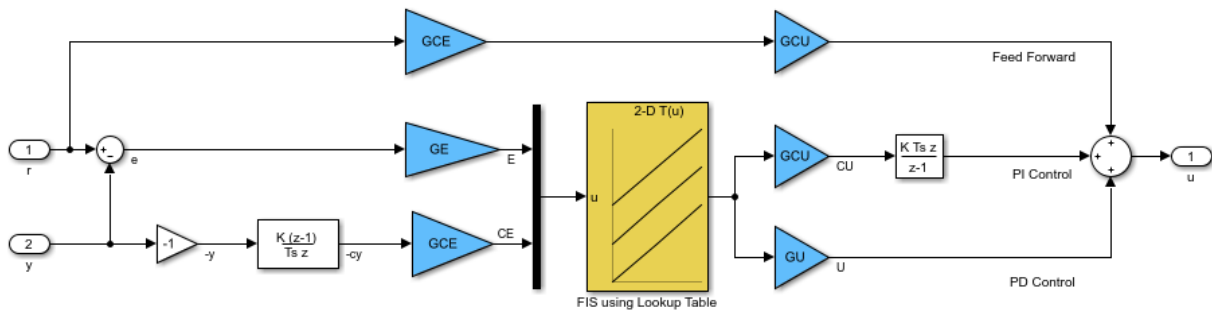
The fuzzy controller block has two inputs (E and CE) and one output (u). Therefore, you can replace the fuzzy system using a 2-D lookup table.

To generate a 2-D lookup table from your FIS, loop through the input universe, and compute the corresponding output values using `evalfis`. Since the control surface is linear, you can use a few sample points for each input variable.

```
Step = 10;
E = -10:Step:10;
CE = -10:Step:10;
N = length(E);
LookupTableData = zeros(N);
for i=1:N
    for j=1:N
        % Compute output u for each combination of sample points.
        LookupTableData(i,j) = evalfis(FIS,[E(i) CE(j)]);
    end
end
```

View the fuzzy PID controller using 2-D lookup table.

```
open_system('sllookuptable/Fuzzy PID using Lookup Table')
```



The only difference compared to the Fuzzy PID controller is that the Fuzzy Logic Controller block is replaced with a 2-D Lookup Table.

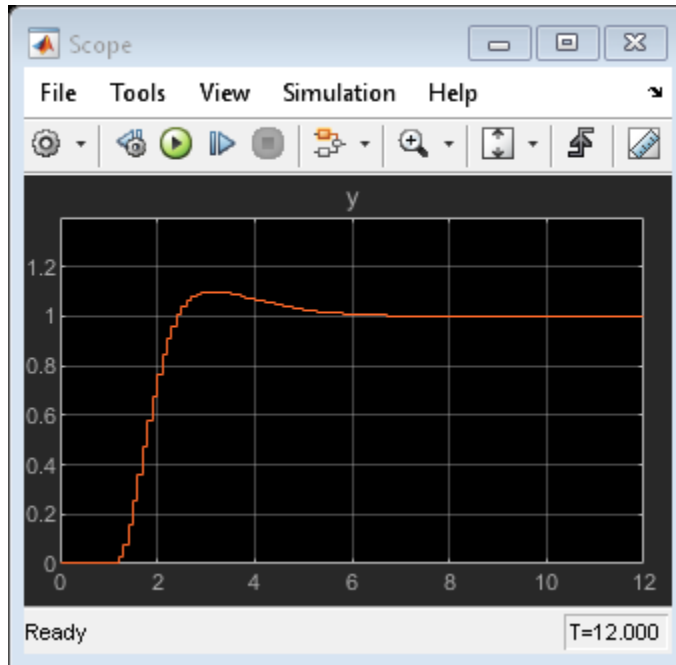
When the control surface is linear, a fuzzy PID controller using the 2-D lookup table produces the same result as one using the Fuzzy Logic Controller block.

Simulate Closed-Loop Response in Simulink

The Simulink model simulates three different controller subsystems, namely Conventional PID, Fuzzy PID, and Fuzzy PID using Lookup Table, to control the same plant.

Run the simulation. To compare the closed-loop responses to a step reference change, open the scope. As expected, all three controllers produce the same result.

```
sim('sllookuptable')
open_system('sllookuptable/Scope')
```



Design Fuzzy PID Controller with Nonlinear Control Surface

Once you have a linear fuzzy PID controller, you can obtain a nonlinear control surface by adjusting your FIS settings, such as its style, membership functions, and rule base.

For this example, design a steep control surface using a Sugeno-type FIS. Each input set has two terms (Positive and Negative), and the number of rules is reduced to four.

Construct the FIS.

```
FIS = sugfis;
```

Define input E.

```
FIS = addInput(FIS, [-10 10], 'Name', 'E');
FIS = addMF(FIS, 'E', 'gaussmf', [7 -10], 'Name', 'Negative');
FIS = addMF(FIS, 'E', 'gaussmf', [7 10], 'Name', 'Positive');
```

Define input CE.

```
FIS = addInput(FIS, [-10 10], 'Name', 'CE');  
FIS = addMF(FIS, 'CE', 'gaussmf', [7 -10], 'Name', 'Negative');  
FIS = addMF(FIS, 'CE', 'gaussmf', [7 10], 'Name', 'Positive');
```

Define output u.

```
FIS = addOutput(FIS, [-20 20], 'Name', 'u');  
FIS = addMF(FIS, 'u', 'constant', -20, 'Name', 'Min');  
FIS = addMF(FIS, 'u', 'constant', 0, 'Name', 'Zero');  
FIS = addMF(FIS, 'u', 'constant', 20, 'Name', 'Max');
```

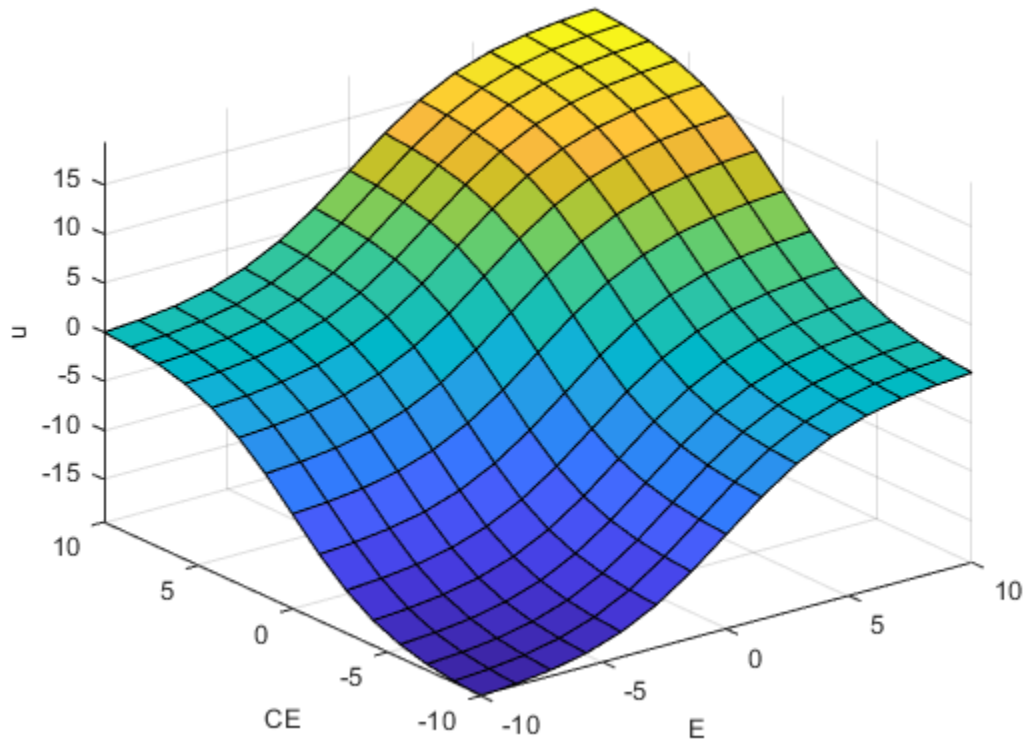
Define the following rules:

- 1 If E is negative and CE is negative, then u is -20.
- 2 If E is negative and CE is positive, then u is 0.
- 3 If E is positive and CE is negative, then u is 0.
- 4 If E is positive and CE is positive, then u is 20.

```
ruleList = [1 1 1 1 1;... % Rule 1  
            1 2 2 1 1;... % Rule 2  
            2 1 2 1 1;... % Rule 3  
            2 2 3 1 1]; % Rule 4  
FIS = addRule(FIS, ruleList);
```

View the 3-D nonlinear control surface. This surface has a higher gain near the center of the E and CE plane than the linear surface has, which helps reduce the error more quickly when the error is small. When the error is large, the controller becomes less aggressive to avoid possible saturation.

```
gensurf(FIS)
```



Before starting the simulation, update the lookup table with the new control surface data. Since the surface is nonlinear, to obtain a sufficient approximation, add more sample points.

```

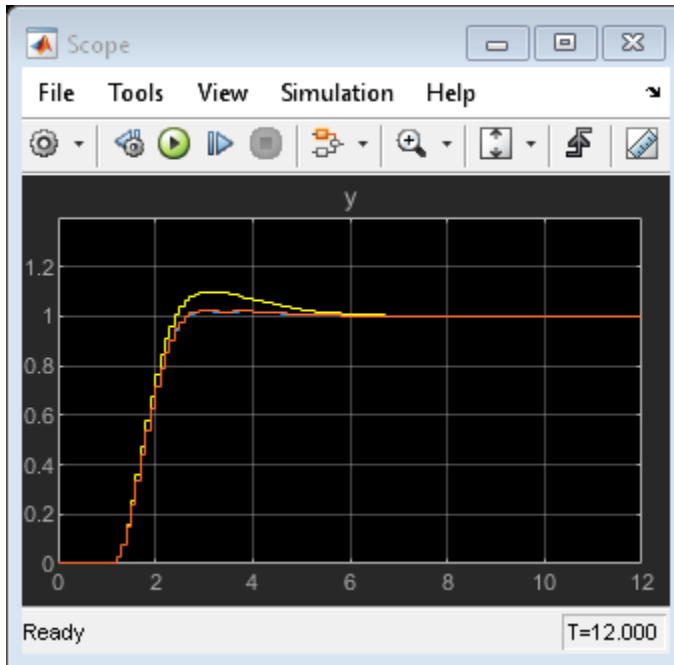
Step = 1;
E = -10:Step:10;
CE = -10:Step:10;
N = length(E);
LookupTableData = zeros(N);
for i=1:N
    for j=1:N
        % Compute output u for each combination of sample points.
        LookupTableData(i,j) = evalfis(FIS,[E(i) CE(j)]);
    end
end

```

```
end  
end
```

Run the simulation.

```
sim('sllookuptable')
```



Compared with the traditional linear PID controller (the response curve with large overshoot), the nonlinear fuzzy PID controller reduces the overshoot by 50%. The two response curves from the nonlinear fuzzy controllers almost overlap, which indicates that the 2-D lookup table approximates the fuzzy system well.

```
bdclose('sllookuptable') % Closing model also clears its workspace variables.
```

Conclusion

You can approximate a nonlinear fuzzy PID controller using a lookup table. By replacing a Fuzzy Logic Controller block with Lookup Table blocks in Simulink, you can deploy a fuzzy controller with simplified generated code and improved execution speed.

References

[1] Xu, J. X., Hang, C. C., Liu, C. "Parallel structure and tuning of a fuzzy PID controller." *Automatica*, Vol. 36, pp. 673-684. 2000.

[2] Jantzen, J. *Tuning of Fuzzy PID Controllers*, Technical Report, Dept. of Automation, Technical University of Denmark. 1999.

See Also

Blocks

2-D Lookup Table | Fuzzy Logic Controller

More About

- "Simulate Fuzzy Inference Systems in Simulink" on page 5-2

Deployment

- “Deploy Fuzzy Inference Systems” on page 6-2
- “Generate Code for Fuzzy System Using Simulink Coder” on page 6-4
- “Generate Structured Text for Fuzzy System Using Simulink PLC Coder” on page 6-9
- “Generate Code for Fuzzy System Using MATLAB Coder” on page 6-12

Deploy Fuzzy Inference Systems

You can deploy a fuzzy inference system (FIS) by generating code in either Simulink or MATLAB. Generating code supports all fuzzy inference system options, including custom inference functions.

Generate Code in Simulink

You can generate code for evaluating fuzzy inference systems in Simulink using the Fuzzy Logic Controller block. You can generate code for double-precision, single-precision, or fixed-point data using Simulink Coder™ or Simulink PLC Coder™.

For more information, see “Generate Code for Fuzzy System Using Simulink Coder” on page 6-4 and “Generate Structured Text for Fuzzy System Using Simulink PLC Coder” on page 6-9.

Generate Code in MATLAB

You can generate code for evaluating fuzzy inference systems in MATLAB. You can generate code for double-precision or single-precision data using MATLAB Coder.

Code generation in MATLAB does not support fuzzy inference system objects. Instead, convert your fuzzy system into a homogeneous structure using `getFISCodeGenerationData`, and pass the resulting structure to `evalfis`.

For more information, see “Generate Code for Fuzzy System Using MATLAB Coder” on page 6-12.

Note Code generation does not support the construction of fuzzy systems at the command line.

See Also

Functions

`evalfis` | `mamfis` | `sugfis`

Blocks

Fuzzy Logic Controller

More About

- “Build Fuzzy Systems at the Command Line” on page 2-38
- “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14

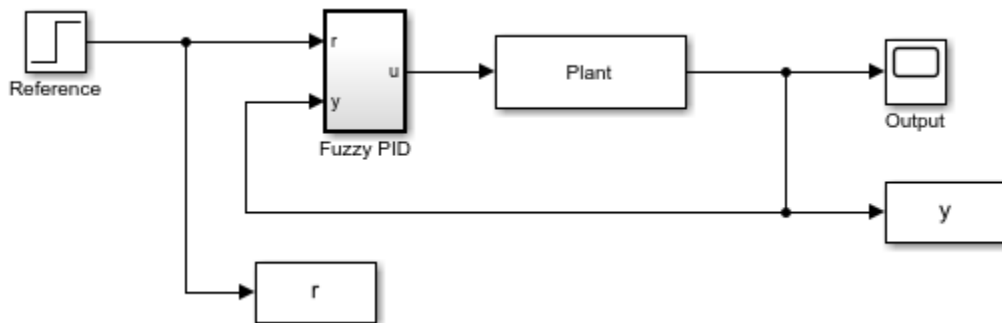
Generate Code for Fuzzy System Using Simulink Coder

You can generate code for a Fuzzy Logic Controller block using Simulink® Coder™. For more information on generating code, see “Code Generation” (Simulink Coder).

Generate Code for Fuzzy Inference System

By default, the Fuzzy Logic Controller block uses double-precision data for simulation and code generation. The `fuzzyPID` model is configured to use double-precision data. For more information on configuring your fuzzy inference system for code generation, see Fuzzy Logic Controller.

```
mdl = 'fuzzyPID';
open_system(mdl)
```



It is good practice to validate the performance of the system in Simulink. Run the simulation. The model saves the output response, `u`, to the MATLAB® workspace.

```
sim(mdl)
```

To generate code for the model, use the `rtwbuild` function. For this example, suppress the Command Window output for the build process.

```
set_param(mdl, 'RTWVerbose', 'off')
rtwbuild(mdl)
```

```
### Starting build procedure for model: fuzzyPID
### Successful completion of build procedure for model: fuzzyPID
```

By default, Simulink Coder generates C code for a generic real-time target. To select a different target file and language, in the Configuration Parameters dialog box, modify the **System target file** and **Language** parameters, respectively.

The generated code is stored in a new `fuzzyPID_grt_rtw` folder in your current working folder. The name of this folder depends on the selected target file.

On a Windows® system, by default, an executable file named `fuzzyPID.exe` is also added to the current working folder. To generate code without compilation, in the Configuration parameters dialog box, select the **Generate code only** parameter before generating code.

Run the executable.

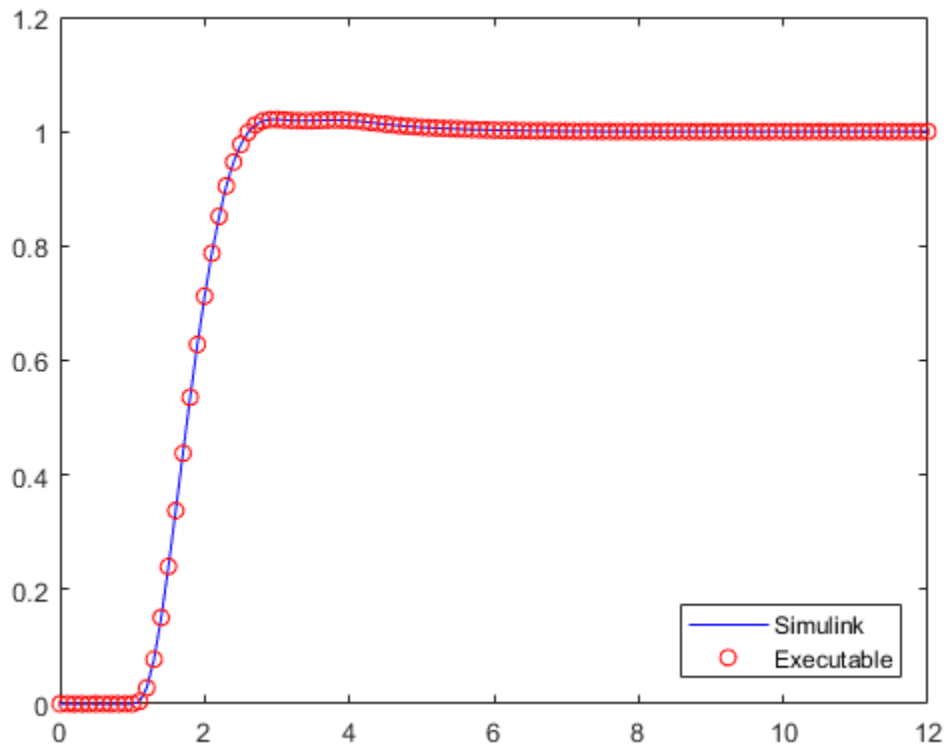
```
if ispc
    status = system mdl;
else
    disp('The example only runs the executable on Windows system. ');
end

** starting the model **
** created fuzzyPID.mat **
```

After the executable completes successfully (`status = 0`), the software creates a `fuzzyPID.mat` data file that contains the simulation results.

You can compare the output response from the generated code, `rtw_y`, with the output from the Simulink simulation, `y`, using the following code:

```
load fuzzyPID.mat
plot(tout,y,'b-',rt_tout,rt_y,'ro')
legend('Simulink','Executable','Location','Southeast')
```



The result from the generated code matches the Simulink simulation.

You can also generate code for just the controller subsystem in this model. To do so, specify the subsystem when calling the `rtwbuild` function.

```
rtwbuild([mdl '/Fuzzy PID'])
```

```
### Starting build procedure for model: Fuzzy0  
### Successful completion of build procedure for model: Fuzzy0
```

You can deploy generated code according to your application needs. For example, you can configure the properties of executable files and create static or dynamic libraries. For more information, see “Deployment” (Simulink Coder).

Generate Code for Other Data Types

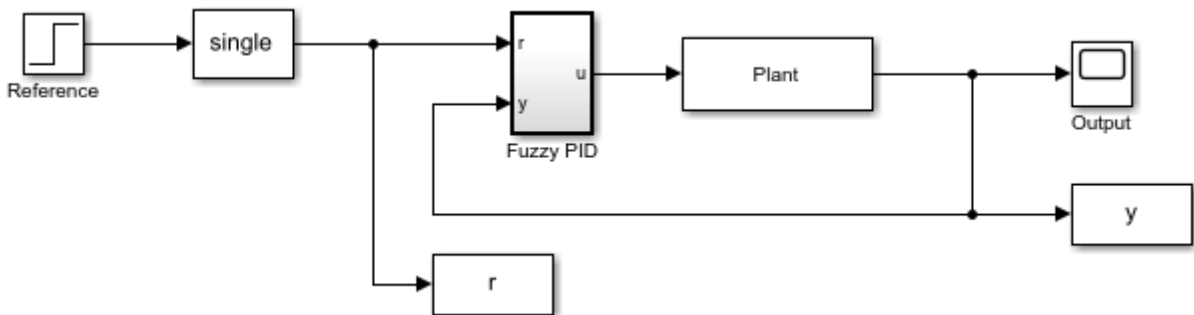
The Fuzzy Logic Controller block also supports single-precision and fixed-point data for simulation and code generation. In both cases, your resulting fuzzy system has decreased accuracy compared to an equivalent double-precision fuzzy system. Use:

- Single-precision data to reduce the memory footprint of your system.
- Fixed-point data if your target platform only supports fixed-point arithmetic.

To use one of these data types, set the **Data type** property of the block, and configure the other components in the model to use the same data type.

The `fuzzyPID_single` model is configured for single-precision data. Open the model.

```
mdl2 = 'fuzzyPID_single';
open_system mdl2)
```



In this model, the **Data type** parameter of the Fuzzy Logic Controller block is set to `single`. The Fuzzy Logic Controller block automatically converts input signals to the specified data type. Also, the **Simulate using** parameter is set to `Code Generation`. The **Simulate using** option does not affect the code generation process. Instead, setting this option simulates your fuzzy system using the same code path used by generated code.

Generate code for this model.

```
set_param mdl2, 'RTWVerbose', 'off')
rtwbuild mdl2)
```

```
### Starting build procedure for model: fuzzyPID_single
### Successful completion of build procedure for model: fuzzyPID_single
```

Setting the **Data type** parameter of a Fuzzy Logic Controller block ensures that all the inference steps use the specified data type. However, depending on the configuration of other blocks in the model, some of the generated code can still use double-precision data.

See Also

Fuzzy Logic Controller

More About

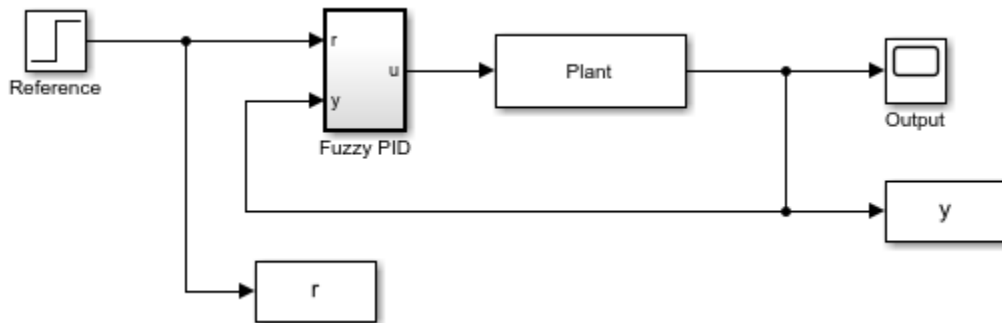
- “Deploy Fuzzy Inference Systems” on page 6-2
- “Generate Structured Text for Fuzzy System Using Simulink PLC Coder” on page 6-9
- “Generate Code for Fuzzy System Using MATLAB Coder” on page 6-12

Generate Structured Text for Fuzzy System Using Simulink PLC Coder

You can generate Structured Text for a Fuzzy Logic Controller block using Simulink® PLC Coder™. For more information on generating Structured Text, see “Code Generation” (Simulink PLC Coder).

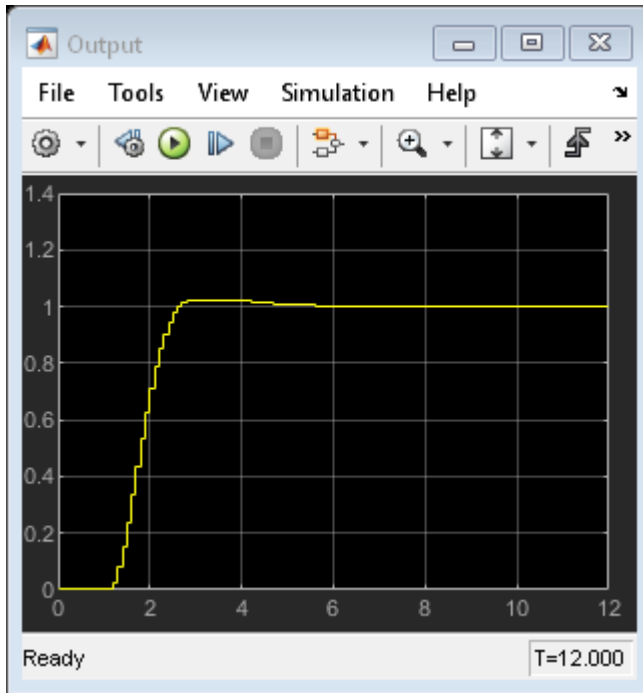
By default, the Fuzzy Logic Controller block uses double-precision data for simulation and code generation. The fuzzyPID model is configured to use double-precision data. You can also use either single-precision or fixed-point data. For more information on configuring your fuzzy inference system for code generation, see Fuzzy Logic Controller.

```
mdl = 'fuzzyPID';  
open_system(mdl)
```



It is good practice to validate the performance of the system in Simulink before generating code. Run the simulation.

```
sim(mdl)  
open_system([mdl '/Output'])
```



To generate Structured Text for the model, use the `plcgeneratecode` command. This command generates code for an *atomic subsystem* in a model. To generate code for the Fuzzy PID controller, configure the subsystem as an atomic subsystem by selecting the **Treat as atomic unit** parameter for the subsystem.

```
subsys = [mdl '/Fuzzy PID'];
set_param(subsys, 'TreatAsAtomicUnit', 'on')
```

When generating code for just a Fuzzy Logic Controller block, place the block inside a subsystem, and set the **Treat as atomic unit** parameter of that subsystem.

Generate Structured Text for the Fuzzy PID subsystem.

```
plcgeneratecode(subsys);
```

```
PLC code generation successful for 'fuzzyPID/Fuzzy PID'.
```

Generated files:

```
<a href="matlab: edit('plcsrc\fuzzyPID.exp')">plcsrc\fuzzyPID.exp</a>
```

By default, the software saves the generated code in the following location:

`plcsrc/fuzzy_PID.exp`

See Also

Fuzzy Logic Controller

More About

- “Deploy Fuzzy Inference Systems” on page 6-2
- “Generate Code for Fuzzy System Using Simulink Coder” on page 6-4

Generate Code for Fuzzy System Using MATLAB Coder

You can generate code for evaluating a fuzzy inference system using MATLAB® Coder™. For more information on generating code, see “Code Generation” (MATLAB Coder).

To generate code for evaluating fuzzy systems, you must first create a fuzzy inference system (FIS). For more information, see “Build Fuzzy Systems at the Command Line” on page 2-38 and “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14.

Generating code using MATLAB Coder does not support `mamfis` or `sugfis` objects. To generate code for evaluating fuzzy systems, you must convert your fuzzy inference system objects into homogeneous structures using the `getFISCodeGenerationData` function.

Embed FIS Data in Generated Code

You can embed the data for your fuzzy inference system within the generated code. Use this option if you do not want to change the FIS data after compilation.

First, create a fuzzy system, or load a fuzzy system from a `.fis` file. For this example, load the fuzzy system from `tipper.fis`.

```
fisObject = readfis("tipper.fis");
```

To use this FIS for code generation, convert it to a homogeneous structure.

```
fis = getFISCodeGenerationData(fisObject);
```

Create a function for evaluating the fuzzy system `fis` for a given input vector `x`. Within this function, you can specify options for the `evalfis` function using `evalfisOptions`.

```
function y = evaluatefis1(fis,x)
    %#codegen
    opt = evalfisOptions('NumSamplePoints',51);
    y = evalfis(fis,x,opt);
end
```

Generate code for `evaluatefis1`, specifying that the `fis` input argument is constant. You can specify different targets for your build, such as a static library, an executable, or a MEX file. For this example, generate a MEX file.

```
codegen('evaluatefis1','-args',{coder.Constant(fis)],[0 0]','-config:mex')
```

To verify the execution of the MEX file:

- 1 Evaluate the MEX file for one or more input values. When you call the MEX file, specify the same FIS structure that you used at compile time.
- 2 Evaluate the original FIS for the same input values using `evalfis`. When evaluating using `evalfis`, use the same homogeneous FIS structure.
- 3 Compare the evaluation results.

```
mexOutput1 = evaluatefis1_mex(fis,[7 9])
```

```
mexOutput1 = 21.0327
```

```
opt = evalfisOptions('NumSamplePoints',51);
evalfisOutput = evalfis(fis,[7 9],opt)
```

```
evalfisOutput = 21.0327
```

The MEX file output matches the `evalfis` output.

Alternatively, you can embed the FIS data in the generated code by reading the FIS data from a file at code generation time. Specify a function for evaluating a fuzzy system for given input vector `x`. Within this function, read the FIS data from the file `tipper.fis`.

```
function y = evaluatefis2(x)
    %#codegen
    fis = getFISCodeGenerationData('tipper.fis');
    opt = evalfisOptions('NumSamplePoints',51);
    y = evalfis(fis,x,opt);
end
```

Generate code for `evaluatefis2`.

```
codegen('evaluatefis2','-args',{[0 0]},'-config:mex')
```

Verify the execution of the MEX file using the same input values for `x`. In this case, you do not have to specify the original FIS structure used at compile time.

```
mexOutput2 = evaluatefis2_mex([7 9])
```

```
mexOutput2 = 21.0327
```

```
evalfisOutput
```

```
evalfisOutput = 21.0327
```

Generate Code for Loading FIS Data at Run Time

You can generate code for evaluating a FIS that is read from a `.fis` file specified at run time. In this case, the FIS data is not embedded in the generated code. Specify a function for evaluating the fuzzy system defined in the specified file `fileName` for a given input vector `x`.

```
function y = evaluatefis3(fileName,x)
    %#codegen
    fis = getFISCodeGenerationData(fileName);
    opt = evalfisOptions('NumSamplePoints',51);
    y = evalfis(fis,x,opt);
end
```

Define input data types for this function.

```
fileName = coder.newtype('char',[1 Inf],[false true]);
x = coder.newtype('double',[1 Inf],[false true]);
```

Generate code for `evaluatefis3`.

```
codegen('evaluatefis3','-args',{fileName,x},'-config:mex')
```

Verify the execution of the MEX file using the same input values for `x`. In this case, you specify the name of the `.fis` file.

```
mexOutput3 = evaluatefis3_mex('tipper.fis',[7 9])
```

```
mexOutput3 = 21.0327
```

```
evalfisOutput
```

```
evalfisOutput = 21.0327
```

Each time you run `evaluatefis3`, it reloads the fuzzy system from the file. For computational efficiency, you can create a function that only loads the FIS when a new file name is specified.

```
function y = evaluatefis4(fileName,x)
    %#codegen
    %#internal

    persistent fisName fis
    if isempty(fisName)
        [fisName,fis] = loadFIS(fileName);
```



```

elseif ~strcmp(fisName,fileName)
    [fisName,fis] = loadFIS(fileName);
end

opt = evalfisOptions('NumSamplePoints',51);
y = evalfis(fis,x,opt);
end

function [fisName,fis] = loadFIS(fileName)
    fisName = fileName;
    fis = getFISCodeGenerationData(fisName);
end

```

Generate code `evaluatefis4`. The input data types for this function are the same as for `evaluatefis3`.

```
codegen('evaluatefis4','-args',{fileName,x},'-config:mex')
```

Verify the execution of the MEX file using the same input values file name.

```
mexOutput4 = evaluatefis4_mex('tipper.fis',[7 9])

mexOutput4 = 21.0327

evalfisOutput

evalfisOutput = 21.0327
```

Generate Code for Single-Precision Data

The preceding examples generated code for double-precision data. To generate code for single-precision data, specify the data type of the input values as `single`. For example, generate code for `evaluatefis2` using single-precision data.

```
codegen('evaluatefis2','-args',{single([0 0])},'-config:mex')
```

Verify the MEX file execution, passing in single-precision input values.

```
mexOutputSingle = evaluatefis2_mex(single([7 9]))

mexOutputSingle = single
    21.0327

evalfisOutput
```

```
evalfisOutput = 21.0327
```

See Also

`evalfis` | `getFISCodeGenerationData`

More About

- “Deploy Fuzzy Inference Systems” on page 6-2
- “Generate Code for Fuzzy System Using Simulink Coder” on page 6-4

Apps — Alphabetical List

Fuzzy Logic Designer

Design and test fuzzy inference systems

Description

The **Fuzzy Logic Designer** app lets you design and test fuzzy inference systems for modeling complex system behaviors.

Using this app, you can:

- Design Mamdani and Sugeno fuzzy inference systems.
- Add or remove input and output variables.
- Specify input and output membership functions.
- Define fuzzy if-then rules.
- Select fuzzy inference functions for:
 - And operations
 - Or operations
 - Implication
 - Aggregation
 - Defuzzification
- Adjust input values and view associated fuzzy inference diagrams.
- View output surface maps for fuzzy inference systems.
- Export fuzzy inference systems to the MATLAB workspace.

Open the Fuzzy Logic Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- MATLAB command prompt: Enter `fuzzyLogicDesigner`.

Examples

- “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14

Programmatic Use

`fuzzyLogicDesigner` opens the **Fuzzy Logic Designer** app.

`fuzzyLogicDesigner(fis)` opens the app and loads the fuzzy inference system `fis`. `fis` can be any `mamfis` or `sugfis` object in the MATLAB workspace.

`fuzzyLogicDesigner(fileName)` opens the app and loads a fuzzy inference system from a file. `fileName` is the name of a `.fis` file on the MATLAB path.

To save a fuzzy inference system to a `.fis` file:

- In **Fuzzy Logic Designer**, select **File > Export > To File**.
- At the command line, use `writefis`.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

Apps

Neuro-Fuzzy Designer

Functions

evalfis | mfedit | newfis | plotfis | ruleedit | ruleview | surfview

Topics

“Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14

“What Is Fuzzy Logic?” on page 1-3

“Foundations of Fuzzy Logic” on page 1-10

“Fuzzy Inference Process” on page 1-28

Introduced in R2014b

Neuro-Fuzzy Designer

Design, train, and test Sugeno-type fuzzy inference systems

Description

The **Neuro-Fuzzy Designer** app lets you design, train, and test adaptive neuro-fuzzy inference systems (ANFIS) using input/output training data.

Using this app, you can:

- Tune membership function parameters of Sugeno-type fuzzy inference systems.
- Automatically generate an initial inference system structure based on your training data.
- Modify the inference system structure before tuning.
- Prevent overfitting to the training data using additional checking data.
- Test the generalization ability of your tuned system using testing data.
- Export your tuned fuzzy inference system to the MATLAB workspace.

You can use the **Neuro-Fuzzy Designer** to train a Sugeno-type fuzzy inference system that:

- Has a single output.
- Uses weighted average defuzzification.
- Has output membership functions all of the same type, for example `linear` or `constant`.
- Has complete rule coverage with no rule sharing; that is, the number of rules must match the number of output membership functions, and every rule must have a different consequent.
- Has unity weight for each rule.
- Does not use custom membership functions.

Open the Neuro-Fuzzy Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- MATLAB command prompt: Enter `neuroFuzzyDesigner`.

Examples

- “Train Adaptive Neuro-Fuzzy Inference Systems” on page 3-13
- “Test Data Against Trained System” on page 3-18

Programmatic Use

`neuroFuzzyDesigner` opens the **Neuro-Fuzzy Designer** app.

`neuroFuzzyDesigner(fis)` opens the app and loads the fuzzy inference system `fis`. `fis` can be any valid `sugfis` object in the MATLAB workspace.

You can create an initial Sugeno-type fuzzy inference system from training data using the `genfis` command.

`neuroFuzzyDesigner(fileName)` opens the app and loads a fuzzy inference system. `fileName` is the name of a `.fis` file on the MATLAB path.

To save a fuzzy inference system to a `.fis` file:

- In the **Fuzzy Logic Designer**, select **File > Export > To File**
- At the command line, use `writefis`.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

Apps

Fuzzy Logic Designer

Functions

`anfis` | `genfis`

Topics

“Train Adaptive Neuro-Fuzzy Inference Systems” on page 3-13

“Test Data Against Trained System” on page 3-18

“Neuro-Adaptive Learning and ANFIS” on page 3-2

“Comparison of `anfis` and Neuro-Fuzzy Designer Functionality” on page 3-7

Introduced in R2014b

Functions — Alphabetical List

addInput

Add input variable to fuzzy inference system

Syntax

```
fisOut = addInput(fisIn)
fisOut = addInput(fisIn, range)
fisOut = addInput( ____, Name, Value)
```

Description

`fisOut = addInput(fisIn)` adds a default input variable to `fisIn` and returns the resulting fuzzy system in `fisOut`. This input variable has a default name, default range, and no membership functions.

`fisOut = addInput(fisIn, range)` adds an input variable with the specified range.

`fisOut = addInput(____, Name, Value)` configures the input variable using one or more name-value pair arguments.

Examples

Add Input Variable to Fuzzy Inference System

Create a Sugeno fuzzy inference system.

```
fis = sugfis('Name', 'tipper');
```

Add an input variable with default specifications.

```
fis = addInput(fis);
```

You can configure the input variable properties using dot notation. For example, specify the name and range for the variable.

```
fis.Inputs(1).Name = "service";  
fis.Inputs(1).Range = [0 10];
```

View the input variable.

```
fis.Inputs(1)  
  
ans =  
    fisvar with properties:  
  
        Name: "service"  
        Range: [0 10]  
    MembershipFunctions: [0x0 fismf]
```

You can also specify a variable name and range when you add it to the fuzzy system.

```
fis2 = sugfis('Name', 'tipper');  
fis2 = addInput(fis2, [0 10], 'Name', "service");
```

Add Input Variable with Membership Functions

Create a fuzzy inference system.

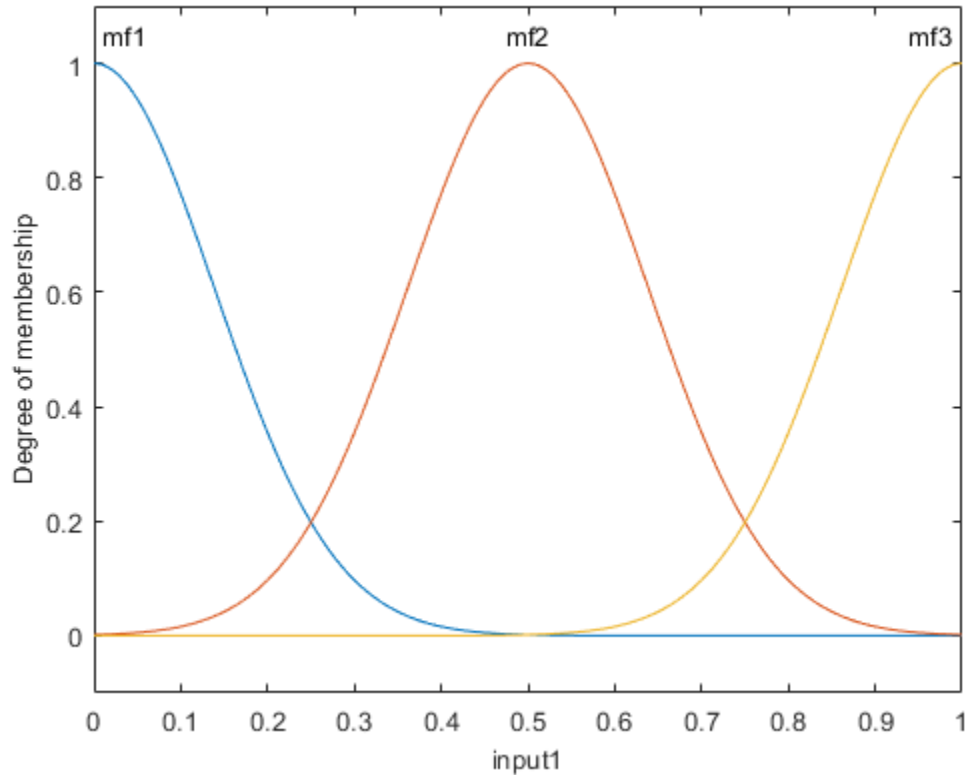
```
fis = mamfis('Name', "tipper");
```

Add an input variable with three Gaussian membership functions distributed over the input range.

```
fis = addInput(fis, 'NumMFs', 3, 'MFType', "gaussmf");
```

View the membership functions.

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



Input Arguments

fisIn — Fuzzy inference system

mamfis object | sugfis object

Fuzzy inference system, specified as a mamfis or sugfis object.

range — Variable range

[0 1] (default) | two-element vector

Variable range, specified as a two-element element vector where the first element is less than the second element. The first element specifies the lower bound of the range, and the second element specifies the upper bound of the range.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'NumMFs', 3` configures the variable to use three membership functions

Name — Variable name

string | character vector

Variable name, specified as the comma-separated pair consisting of `'Name'` and a string or character vector. The default variable name is `"input<uniqueIndex>"`, where `uniqueIndex` is automatically generated based on the current number of inputs in `fisIn`.

NumMFs — Number of membership functions

0 (default) | nonnegative integer

Number of membership functions, specified as the comma-separated pair consisting of `'NumMFs'` and a nonnegative integer.

MFType — Membership function type

"trimf" (default) | "gaussmf"

Membership function type, specified as the comma-separated pair consisting of `'MFType'` and one of the following:

- `"trimf"` — Triangular membership functions
- `"gaussmf"` — Gaussian membership functions

The membership functions are uniformly distributed over the input variable range with approximately 80% overlap in the membership function supports.

Output Arguments

fisOut — Fuzzy inference system

mamfis object | sugfis object

Fuzzy inference system, returned as a mamfis or sugfis object. `fisOut` contains the added input variable, with all other properties matching the properties of `fisIn`.

See Also

`addOutput` | `fisvar` | `mamfis` | `removeInput` | `sugfis`

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

addMF

Add membership function to fuzzy variable

Syntax

```
fisOut = addMF(fisIn,varName)
fisOut = addMF(fisIn,varName,type,parameters)
fisOut = addMF( ____,Name,Value)
```

```
varOut = addMF(varIn)
varOut = addMF(varIn,type,parameters)
varOut = addMF( ____,Name,Value)
```

Description

`fisOut = addMF(fisIn,varName)` adds a default membership function to the input or output variable `varName` in the fuzzy inference system `fisIn` and returns the resulting fuzzy system in `fisOut`.

`fisOut = addMF(fisIn,varName,type,parameters)` adds a membership function with the specified type and parameters.

`fisOut = addMF(____,Name,Value)` configures the membership function using one or more name-value pair arguments.

`varOut = addMF(varIn)` adds a default membership function to fuzzy variable `varIn` and returns the resulting fuzzy variable in `varOut`.

`varOut = addMF(varIn,type,parameters)` adds a membership function with the specified type and parameters.

`varOut = addMF(____,Name,Value)` specifies the name of the membership function using the `Name` name-value pair argument.

Examples

Add Membership Function to Fuzzy Inference System

Create a Mamdani fuzzy system, and add three input variables and one output variable. For this example, give the second input variable and the output variable the same name.

```
fis = mamfis;
fis = addInput(fis,[0 80],"Name","speed");
fis = addInput(fis,[0 100],"Name","throttle");
fis = addInput(fis,[0 10],"Name","distance");
fis = addOutput(fis,[0 100],"Name","throttle");
```

Add a membership function to the first input variable, specifying a trapezoidal membership function, and set the membership function parameters.

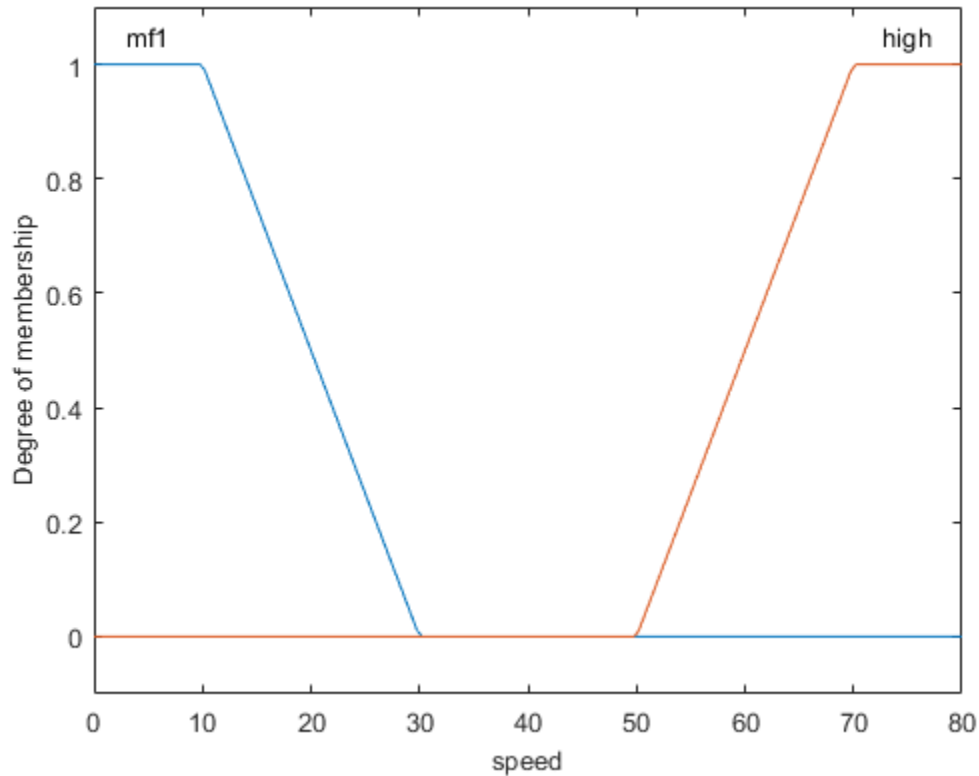
```
fis = addMF(fis,"speed","trapmf",[-5 0 10 30]);
```

You can also specify the name of your membership when you add it to a fuzzy system. Add a membership function called "high" to the first input variable.

```
fis = addMF(fis,"speed","trapmf",[50 70 80 85],'Name','high');
```

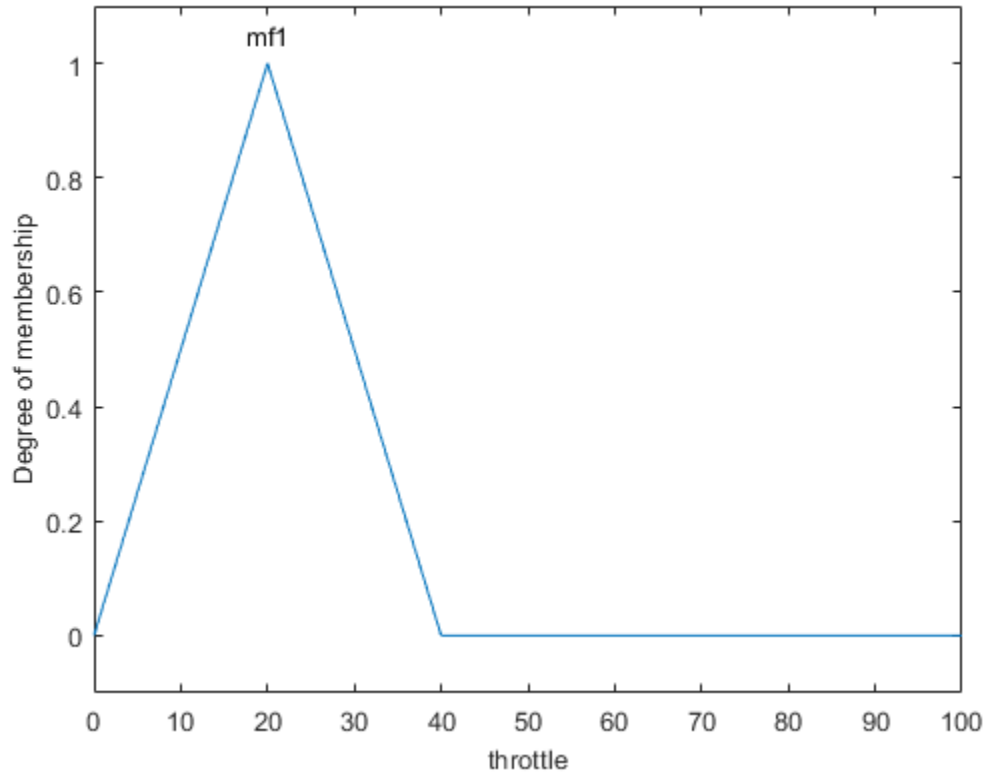
View the membership functions for the first input variable.

```
plotmf(fis,"input",1)
```



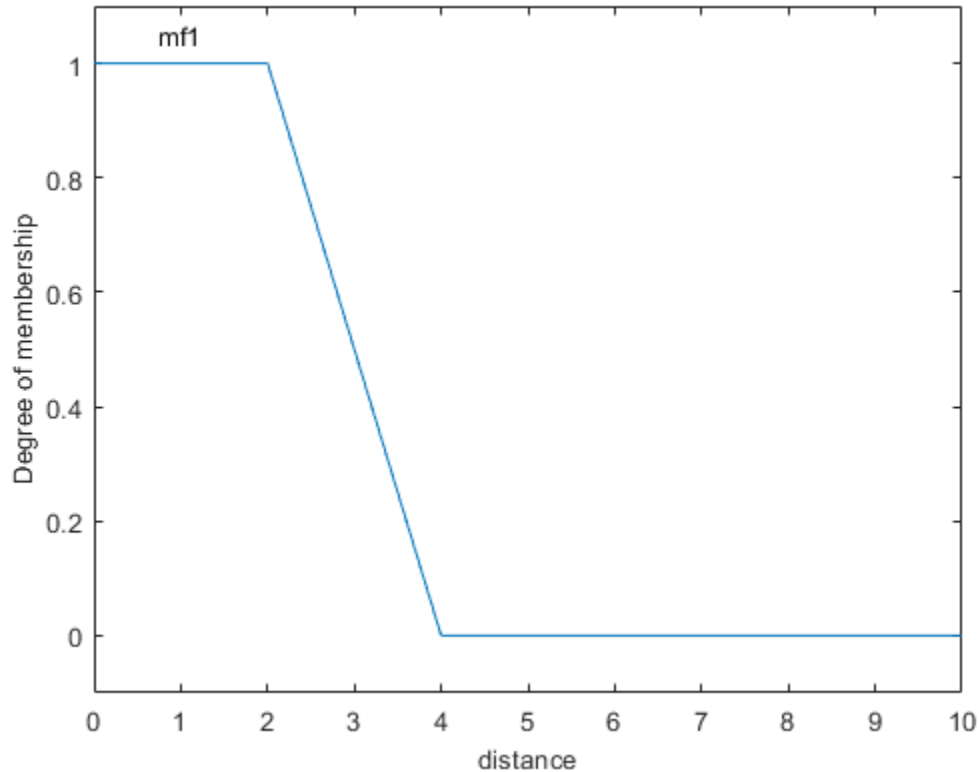
If your system has an input variable with the same name as an output variable, you must specify the variable type when adding a membership function. For example, add a membership function to the output variable.

```
fis = addMF(fis,"throttle","trimf",[0 20 40],'VariableType','output');  
plotmf(fis,"output",1)
```



Alternatively, you can add a default membership function to a fuzzy system and set its parameters using dot notation. For example, add and configure a membership function for the third input variable.

```
fis = addMF(fis,"distance");  
fis.Inputs(3).MembershipFunctions(1).Type = "trapmf";  
fis.Inputs(3).MembershipFunctions(1).Parameters = [-1 0 2 4];  
plotmf(fis,"input",3)
```



Add Membership Function to Fuzzy Variable

Create a fuzzy variable with a specified range.

```
var = fisvar([0 1]);
```

Add a membership function to the variable, specifying a trapezoidal membership function, and set the membership function parameters.

```
var = addMF(var, "trapmf", [-0.5 0 0.2 0.4]);
```

You can also specify the name of your membership when you add it to a fuzzy variable. For example, add a membership function called "large".

```
var = addMF(var, "trapmf", [0.6 0.8 1 1.5], 'Name', "large");
```

View the membership functions.

```
var.MembershipFunctions
```

```
ans =
```

```
1x2 fsmf array with properties:
```

```
    Name  
    Type  
Parameters
```

```
Details:
```

	Name	Type	Parameters			
1	"mf1"	"trapmf"	-0.5	0	0.2	0.4
2	"large"	"trapmf"	0.6	0.8	1	1.5

Alternatively, you can add a default membership function to a fuzzy variable and set its parameters using dot notation.

```
var = fisvar([0 1]);  
var = addMF(var);  
var.MembershipFunctions(1).Type = "trapmf";  
var.MembershipFunctions(1).Parameters = [-0.5 0 0.2 0.4];
```

Input Arguments

fisIn — Fuzzy inference system

mamfis object | sugfis object

Fuzzy inference system, specified as a mamfis or sugfis object.

varName — Variable name

string | character vector

Variable name, specified as a string or character vector. You can specify the name of either an input or output variable in your FIS. If your system has an input variable with the same name as an output variable, specify the type of the variable you want to add a membership function to using the `VariableType` name-value pair.

type — Membership function type

"trimf" (default) | string | character vector | function handle

Membership function type, specified as a string or character vector that contains the name of a function in the current working folder or on the MATLAB path. You can also specify a handle to such a function. When you specify `type`, you must also specify parameters.

This table describes the values that you can specify for `type`.

Membership Function Type	Description	For More Information
"gbellmf"	Generalized bell-shaped membership function	gbellmf
"gaussmf"	Gaussian membership function	gaussmf
"gauss2mf"	Gaussian combination membership function	gauss2mf
"trimf"	Triangular membership function	trimf
"trapmf"	Trapezoidal membership function	trapmf
"sigmf"	Sigmoidal membership function	sigmf
"dsigmf"	Difference between two sigmoidal membership functions	dsigmf
"psigmf"	Product of two sigmoidal membership functions	psigmf
"zmf"	Z-shaped membership function	zmf
"pimf"	Pi-shaped membership function	pimf
"smf"	S-shaped membership function	smf
"constant"	Constant membership function (not supported for output variables of Mamdani systems or for any input variables)	"What Is Sugeno-Type Fuzzy Inference?" on page 2-5

Membership Function Type	Description	For More Information
"linear"	Linear membership function (not supported for output variables of Mamdani systems or for any input variables)	
String or character vector	Name of a custom membership function in the current working folder or on the MATLAB path. Custom functions are not supported for output variables of Sugeno systems.	"Build Fuzzy Systems Using Custom Functions" on page 2-50
Function handle	Handle to a custom membership function in the current working folder or on the MATLAB path. Custom functions are not supported for output variables of Sugeno systems.	

parameters — Membership function parameters

[0 0.5 1] (default) | vector

Membership function parameters, specified as a vector. The length of the parameter vector depends on the membership function type. When you specify `parameters`, you must also specify `type`.

varIn — Fuzzy variable

`fisvar` object

Fuzzy variable, specified as a `fisvar` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Name', "large"` specifies the membership function name "large"

Name — Membership function name`string | character vector`

Membership function name, specified as the comma-separated pair consisting of 'Name' and a string or character vector. The default membership function name is "mf<uniqueIndex>", where `uniqueIndex` is automatically generated based on the current number of membership functions in the associated variable.

VariableType — Variable type`"input" | "output"`

Variable type, specified as the comma-separated pair 'VariableType' and one of the following:

- "input" — Input variable
- "output" — Output variable

If your system has an input variable with the same name as an output variable, specify which variable to add the membership function to `VariableType`.

This name-value pair applies only when adding a membership function to a `mamfis` or `sugfis` object.

Output Arguments

fisOut — Fuzzy inference system`mamfis object | sugfis object`

Fuzzy inference system, returned as a `mamfis` or `sugfis` object. `fisOut` contains the added membership function, with all other properties matching the properties of `fisIn`.

varOut — Fuzzy variable`fisvar object`

Fuzzy variable, returned as a `fisvar` object. `varOut` contains the added membership function, with all other properties matching the properties of `varIn`.

Compatibility Considerations

addmf is now addMF and its function syntax has changed

Behavior changed in R2018b

The name and behavior of the `addmf` function has changed. Now:

- `addmf` is `addMF`
- You specify the variable to which you want to add the membership function by name rather than by index.
- You specify the name of the membership function using a name-value pair argument.

These changes require updates to your code.

The following table shows some typical usages of `addmf` for adding membership functions to fuzzy variables and how to update your code. In this table, `fis` is a fuzzy inference system with two inputs, `service` and `food`, and one output, `tip`.

If your code has this form:	Use this code instead:
<code>fis = addmf(fis,'input',1,... 'poor',... 'gaussmf',[1.5 0])</code>	<code>fis = addMF(fis,"service",... "gaussmf",[1.5 0], 'Name',"poor")</code>
<code>fis = addmf(fis,'input',2,... 'rancid',... 'trapmf',[-2 0 1 3])</code>	<code>fis = addMF(fis,"food",... "trapmf",[-2 0 1 3],... 'Name',"rancid")</code>
<code>fis = addmf(fis,'output',1,... 'cheap',... 'trimf',[0 5 10])</code>	<code>fis = addMF(fis,"tip",... "trimf",[0 5 10],... 'Name',"cheap")</code>

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

`addInput` | `addOutput` | `addRule` | `fisvar` | `mamfis` | `removeMF` | `sugfis`

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

addOutput

Add output variable to fuzzy inference system

Syntax

```
fisOut = addOutput(fisIn)
fisOut = addOutput(fisIn, range)
fisOut = addOutput( ____, Name, Value)
```

Description

`fisOut = addOutput(fisIn)` adds a default output variable to `fisIn`, and returns the resulting fuzzy system in `fisOut`. This output variable has a default name, default range, and no membership functions.

`fisOut = addOutput(fisIn, range)` adds an output variable with the specified range.

`fisOut = addOutput(____, Name, Value)` configures the output variable using one or more name-value pair arguments.

Examples

Add Output Variable to Fuzzy Inference System

Create a Mamdani fuzzy inference system.

```
fis = mamfis('Name', 'tipper');
```

Add an output variable with default specifications.

```
fis = addOutput(fis);
```

You can configure the output variable properties using dot notation. For example, specify the name and range for the variable.

```

fis.Outputs(1).Name = "tip";
fis.Outputs(1).Range = [10 30];

```

View the output variable.

```

fis.Outputs(1)

ans =
    fisvar with properties:
        Name: "tip"
        Range: [10 30]
        MembershipFunctions: [0x0 fismf]

```

You can also specify the variable name and range when you add it to the fuzzy system.

```

fis2 = mamfis('Name', 'tipper');
fis2 = addOutput(fis2, [10 30], 'Name', "tip");

```

Add Output Variable with Membership Functions

Create a Sugeno fuzzy inference system.

```

fis = sugfis('Name', "tipper");

```

Add an output variable with three constant membership functions distributed over the output range.

```

fis = addOutput(fis, 'NumMFs', 3, 'MFTType', "constant");

```

View the membership functions.

```

fis.Outputs(1).MembershipFunctions

ans =
    1x3 fismf array with properties:

```

```

    Name
    Type
    Parameters

```

```

Details:

```

	Name	Type	Parameters
1	"mf1"	"constant"	0
2	"mf2"	"constant"	0.5
3	"mf3"	"constant"	1

Input Arguments

fisIn — Fuzzy inference system

mamfis object | sugfis object

Fuzzy inference system, specified as a mamfis or sugfis object.

range — Variable range

[0 1] (default) | two-element vector

Variable range, specified as a two-element element vector where the first element is less than the second element. The first element specifies the lower bound of the range, and the second element specifies the upper bound of the range.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: 'NumMFs', 3 configures the variable to use three membership functions

Name — Variable name

string | character vector

Variable name, specified as the comma-separated pair consisting of 'Name' and a string or character vector.

NumMFs — Number of membership functions

0 (default) | nonnegative integer

Number of membership functions, specified as the comma-separated pair consisting of 'NumMFs' and a nonnegative integer.

MFType — Membership function type

"trimf" (default) | "gaussmf"

Membership function type, specified as the comma-separated pair consisting of 'MFType' and one of the following:

- "trimf" — Triangular membership functions for the outputs of Mamdani system
- "gaussmf" — Gaussian membership functions for the outputs of Mamdani systems
- "constant" — Constant membership functions for the outputs of Sugeno systems
- "linear" — Linear membership functions for the outputs of Sugeno systems. To add an output variable with linear membership functions, your FIS must have at least one input variable.

The membership functions are uniformly distributed over the variable range with approximately 80% overlap in the membership function supports.

Output Arguments

fisOut — Fuzzy inference system

mamfis object | sugfis object

Fuzzy inference system, returned as a mamfis or sugfis object. fisOut contains the added output variable, with all other properties matching the properties of fisIn.

See Also

addInput | fisvar | mamfis | removeOutput | sugfis

Topics

"Build Fuzzy Systems at the Command Line" on page 2-38

Introduced in R2018b

addRule

Add rule to fuzzy inference system

Syntax

```
fisOut = addRule(fisIn)
fisOut = addRule(fisIn,ruleDescription)
```

Description

`fisOut = addRule(fisIn)` adds a single fuzzy rule to fuzzy inference system `fisIn` with the default description "input1==mf1 => output1=mf1" and returns the resulting fuzzy system in `fisOut`.

`fisOut = addRule(fisIn,ruleDescription)` adds one or more fuzzy rules using the rule descriptions in `ruleDescription`.

Examples

Add Single Rule to Fuzzy Inference System

Load a fuzzy inference system (FIS), and clear the existing rules.

```
fis = readfis('tipper');
fis.Rules = [];
```

Add a rule to the FIS.

```
ruleTxt = 'If service is poor then tip is cheap';
fis2 = addRule(fis,ruleTxt);
```

`fis2` is equivalent to `fis`, except that the specified rule is added to the rule base.

```
fis2.Rules
```



```
ans =
  fisrule with properties:

    Description: "service==poor => tip=cheap (1)"
    Antecedent: [1 0]
    Consequent: 1
    Weight: 1
    Connection: 1
```

Add Rules Using Symbolic Expressions

Load a fuzzy inference system (FIS), and clear the existing rules.

```
fis = readfis('tipper');
fis.Rules = [];
```

Specify the following rules using symbolic expressions:

- If service is poor or food is rancid then tip is cheap.
- If service is excellent and food is not rancid then tip is generous.

```
rule1 = "service==poor | food==rancid => tip=cheap";
rule2 = "service==excellent & food~=rancid => tip=generous";
rules = [rule1 rule2];
```

Add the rules to the FIS.

```
fis2 = addRule(fis,rules);
```

fis2 is equivalent to fis, except that the specified rules are added to the rule base.

```
fis2.Rules
```

```
ans =
  1x2 fisrule array with properties:

    Description
    Antecedent
    Consequent
    Weight
    Connection
```

Details:

Description

```
1 "service==poor | food==rancid => tip=cheap (1)"
2 "service==excellent & food~=rancid => tip=generous (1)"
```

Add Rules Using Membership Function Indices

Load fuzzy inference system (FIS) and clear the existing rules.

```
fis = readfis('mam22.fis');
fis.Rules = [];
```

Specify the following rules using membership function indices:

- If angle is small and velocity is big, then force is negBig and force2 is posBig2.
- If angle is not small and velocity is small, then force is posSmall and force2 is negSmall2.

```
rule1 = [1 2 1 4 1 1];
rule2 = [-1 1 3 2 1 1];
rules = [rule1; rule2];
```

Add the rules to the FIS.

```
fis2 = addRule(fis,rules);
```

fis2 is equivalent to fis, except that the specified rules are added to the rule base.

```
fis2.Rules
```

```
ans =
    1x2 fisrule array with properties:
```

```
    Description
    Antecedent
    Consequent
    Weight
```

Connection

Details:

Description

```

1  "angle==small & velocity==big => force=negBig, force2=posBig2 (1)"
2  "angle~=small & velocity==small => force=posSmall, force2=negSmall2 (1)"

```

Input Arguments

fisIn — Fuzzy inference system

mamfis object | sugfis object

Fuzzy inference system, specified as a mamfis or sugfis object.

ruleDescription — Rule description

string | character vector | numeric row vector | string array | character array | numeric array

Rule description, specified using either a text or numeric rule definition

Text Rule Description

For a text rule description, specify ruleDescription as one of the following:

- String or character vector specifying a single rule


```
rule = "If service is poor or food is rancid then tip is cheap";
```
- String array, where each element corresponds to a rule. For example:


```
ruleList = ["If service is poor or food is rancid then tip is cheap";
            "If service is good then tip is average";
            "If service is excellent or food is delicious then tip is generous"];

```
- Character array where each row corresponds to a rule. For example:

```
rule1 = 'If service is poor or food is rancid then tip is cheap';
rule2 = 'If service is good then tip is average';
rule3 = 'If service is excellent or food is delicious then tip is generous';
ruleList = char(rule1,rule2,rule3);

```

For each rule, use one of the following rule text formats:

- Verbose — Linguistic expression in the following format, using the IF and THEN keywords:

"IF <antecedent> THEN <consequent> (<weight>)"

In <antecedent>, specify the membership function for each input variable using the IS or IS NOT keyword. Connect these conditions using the AND or OR keywords. If a rule does not use a given input variable, omit it from the antecedent.

In <consequent>, specify the condition for each output variable using the IS or IS NOT keyword, and separate these conditions using commas. The IS NOT keyword is not supported for Sugeno outputs. If a rule does not use a given output variable, omit it from the consequent.

Specify the weight using a positive numerical value.

For example:

"IF A IS a AND B IS NOT b THEN X IS x, Y IS NOT y (1)"

- Symbolic — Expression that uses the symbols in the following table instead of keywords. There is no symbol for the IF keyword.

Symbol	Keyword
==	IS (in rule antecedent)
~=	IS NOT
&	AND
	OR
=>	THEN
=	IS (in rule consequent)

For example, the following symbolic rule is equivalent to the previous verbose rule.

"A==a & B~=b => X=x, Y~=y (1)"

Numeric Rule Description

For a numeric rule description, specify ruleDescription as one of the following:

- Row vector to specify a single fuzzy rule

- Array, where each row of `ruleValues` specifies one rule

For each row, the numeric rule description has $M+N+2$ columns, where M is the number of input variables and N is the number of output variables. Each column contains the following information:

- The first M columns specify input membership function indices and correspond to the **Antecedent** property of the rule. To indicate a **NOT** condition, specify a negative value. If a rule does not use a given input, set the corresponding index to 0 . For each rule, at least one input membership function index must be nonzero.
- The next N columns specify output membership function indices and correspond to the **Consequent** property of the rule. To indicate a **NOT** condition for Mamdani systems, specify a negative value. **NOT** conditions are not supported for Sugeno outputs. If a rule does not use a given output, set the corresponding index to 0 . For each rule, at least one output membership function index must be nonzero.
- Column $M+N+1$ specifies the rule weight and corresponds to the **Weight** property of the rule.
- The final column specifies the antecedent fuzzy operator and corresponds to the **Connection** property of the rule.

Output Arguments

fisOut — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, returned as a `mamfis` or `sugfis` object. `fisOut` contains the added rules, with all other properties matching those of `fisIn`.

Compatibility Considerations

addrule is now **addRule**

Behavior changed in R2018b

`addrule` is now `addRule`. To update your code, change the function name from `addrule` to `addRule`. The syntaxes are equivalent.

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

`addInput` | `addMF` | `addOutput` | `mamfis` | `sugfis`

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

addvar

(To be removed) Add variable to fuzzy inference system

Note addvar will be removed in a future release. Use `addInput` or `addOutput` instead. For more information, see “Compatibility Considerations”.

Syntax

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

Description

addvar has four input arguments:

- `fis` — Fuzzy inference system in the MATLAB workspace, specified as a FIS structure.
- `varType` — Type of variable to add, specified as 'input' or 'output'.
- `varName` — Name of the variable to add, specified as a character vector or string.
- `varBounds` — Variable range, specified as a two-element vector, where the first element is the minimum value and the second element is the maximum value for the variable.

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

Examples

Add Variable to Fuzzy Inference System

Create new FIS.

```
fis = newfis('tipper');
```

Add new input variable.

```
fis = addvar(fis, 'input', 'service', [0 10]);
```

View new variable properties.

```
getfis(fis, 'input', 1)
ans = struct with fields:
    Name: 'service'
    NumMFs: 0
    range: [0 10]
```

Compatibility Considerations

addvar will be removed

Not recommended starting in R2018b

`addvar` will be removed in a future release. Use `addInput` or `addOutput` instead. There are differences between these functions that require updates to your code.

To add input or output variables to a fuzzy system, use `addInput` or `addOutput`, respectively.

This table shows some typical usages of `addvar` and how to update your code to use `addInput` or `addOutput` instead.

If your code has this form:	Use this code instead:
<pre>fis = addvar(fis, 'input', ... 'service', [0 10])</pre>	<pre>fis = addInput(fis, [0 10], ... 'Name', "service")</pre>
<pre>fis = addvar(fis, 'output', ... 'tip', [0 30])</pre>	<pre>fis = addOutput(fis, [0 30], ... 'Name', "tip")</pre>

See Also

`addInput` | `addOutput` | `addmf` | `addrule` | `rmmf` | `rmvar`

Introduced before R2006a

anfis

Tune Sugeno-type fuzzy inference system using training data

Syntax

```
fis = anfis(trainingData)
fis = anfis(trainingData,options)

[fis,trainError] = anfis( ___ )
[fis,trainError,stepSize] = anfis( ___ )

[fis,trainError,stepSize,chkFIS,chkError] = anfis(trainingData,
options)
```

Description

`fis = anfis(trainingData)` generates a single-output Sugeno fuzzy inference system (FIS) and tunes the system parameters using the specified input/output training data. The FIS object is automatically generated using grid partitioning.

The training algorithm uses a combination of the least-squares and backpropagation gradient descent methods to model the training data set.

`fis = anfis(trainingData,options)` tunes an FIS using the specified training data and options. Using this syntax, you can specify:

- An initial FIS object to tune.
- Validation data for preventing overfitting to training data.
- Training algorithm options.
- Whether to display training progress information.

`[fis,trainError] = anfis(___)` returns the root mean square training error for each training epoch.

`[fis,trainError,stepSize] = anfis(___)` returns the training step size at each training epoch.

`[fis,trainError,stepSize,chkFIS,chkError] = anfis(trainingData, options)` returns the validation data error for each training epoch, `chkError`, and the tuned FIS object for which the validation error is minimum, `chkFIS`. To use this syntax, you must specify validation data using `options.ValidationData`.

Examples

Train Fuzzy Inference System Using ANFIS

Load training data. This data has a single input and a single output.

```
load fuzex1trnData.dat
```

Generate and train a fuzzy inference system. By default, the FIS structure is created using a grid partition of the input variable range with two membership functions.

```
fis = anfis(fuzex1trnData);
```

ANFIS info:

```
Number of nodes: 12
Number of linear parameters: 4
Number of nonlinear parameters: 6
Total number of parameters: 10
Number of training data pairs: 25
Number of checking data pairs: 0
Number of fuzzy rules: 2
```

Start training ANFIS ...

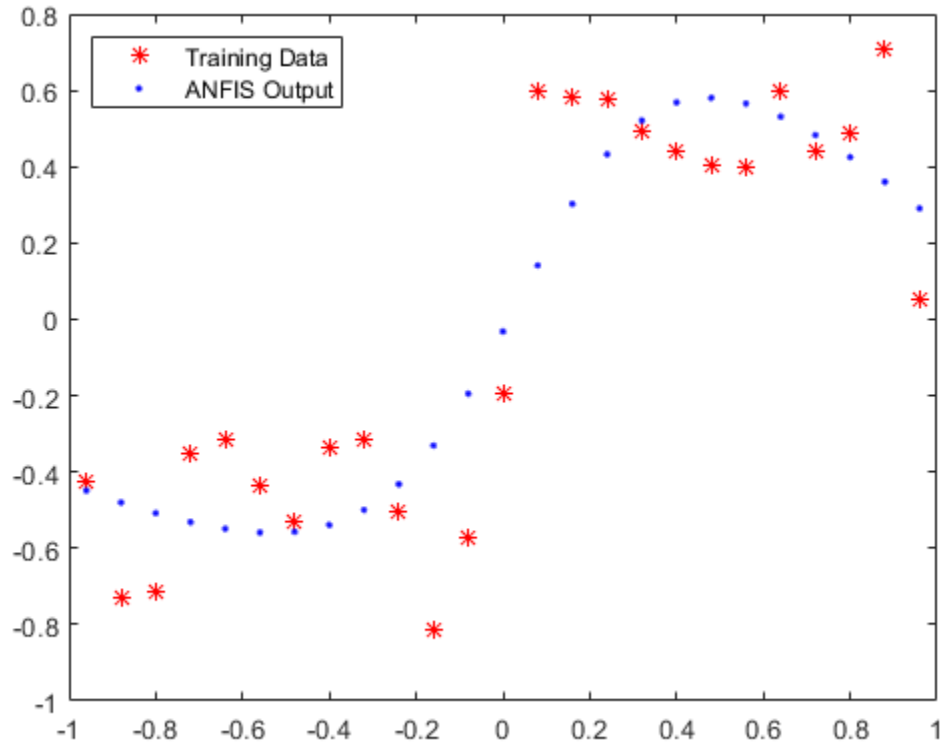
```
1      0.229709
2      0.22896
3      0.228265
4      0.227624
5      0.227036
Step size increases to 0.011000 after epoch 5.
6      0.2265
7      0.225968
8      0.225488
9      0.225052
Step size increases to 0.012100 after epoch 9.
10     0.22465
```

Designated epoch number reached --> ANFIS training completed at epoch 10.

Minimal training RMSE = 0.224650

Plot the ANFIS output and training data.

```
x = fuzex1trnData(:,1);
anfisOutput = evalfis(fis,x);
plot(x,fuzex1trnData(:,2),'*r',x,anfisOutput,'.b')
legend('Training Data','ANFIS Output','Location','NorthWest')
```



The ANFIS data does not match the training data well. To improve the match:

- Increase the number of membership functions in the FIS structure to 4. Doing so adds fuzzy rules and tunable parameters to the system.
- Increase the number of training epochs.

```
opt = anfisOptions('InitialFIS',4,'EpochNumber',40);
```

Suppress the error and step size Command Window display.

```
opt.DisplayErrorValues = 0;  
opt.DisplayStepSize = 0;
```

Train the FIS.

```
fis = anfis(fuzex1trnData,opt);
```

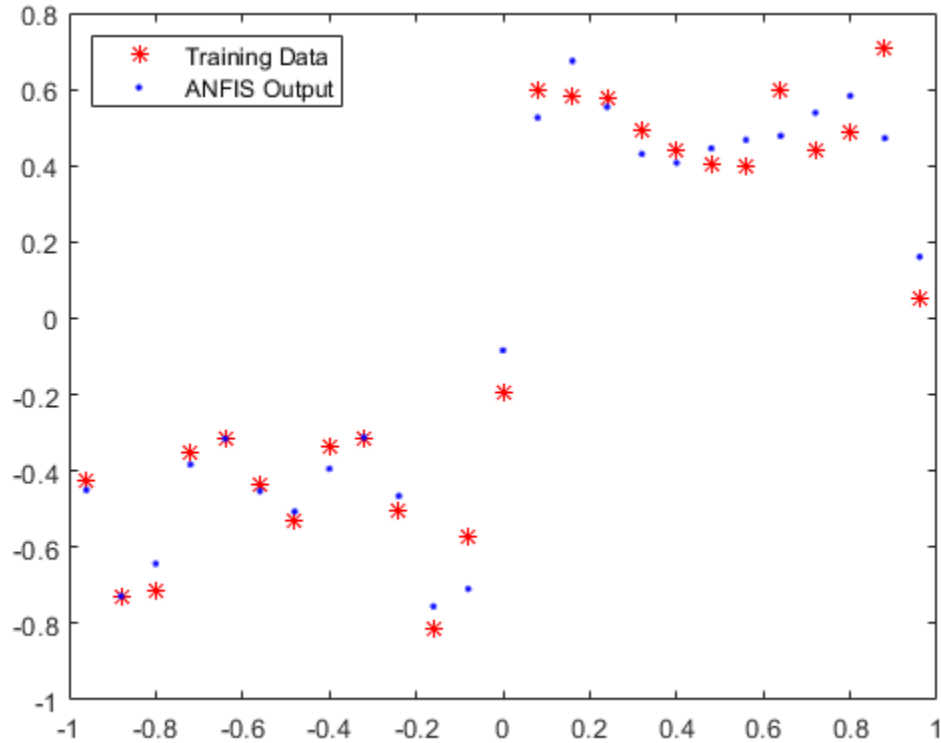
ANFIS info:

```
Number of nodes: 20  
Number of linear parameters: 8  
Number of nonlinear parameters: 12  
Total number of parameters: 20  
Number of training data pairs: 25  
Number of checking data pairs: 0  
Number of fuzzy rules: 4
```

Minimal training RMSE = 0.083385

Plot the ANFIS output and training data.

```
figure  
anfisOutput = evalfis(fis,x);  
plot(x,fuzex1trnData(:,2),'*r',x,anfisOutput,'.b')  
legend('Training Data','ANFIS Output','Location','NorthWest')
```



The match between the training data and ANFIS output has improved.

Create Initial FIS for ANFIS Training

Create single-input, single-output training data.

```
x = (0:0.1:10)';  
y = sin(2*x)./exp(x/5);
```

Define an initial FIS structure with five Gaussian input membership functions.

```
genOpt = genfisOptions('GridPartition');  
genOpt.NumMembershipFunctions = 5;  
genOpt.InputMembershipFunctionType = 'gaussmf';  
inFIS = genfis(x,y,genOpt);
```

Configure the ANFIS training options. Set the initial FIS, and suppress the training progress display.

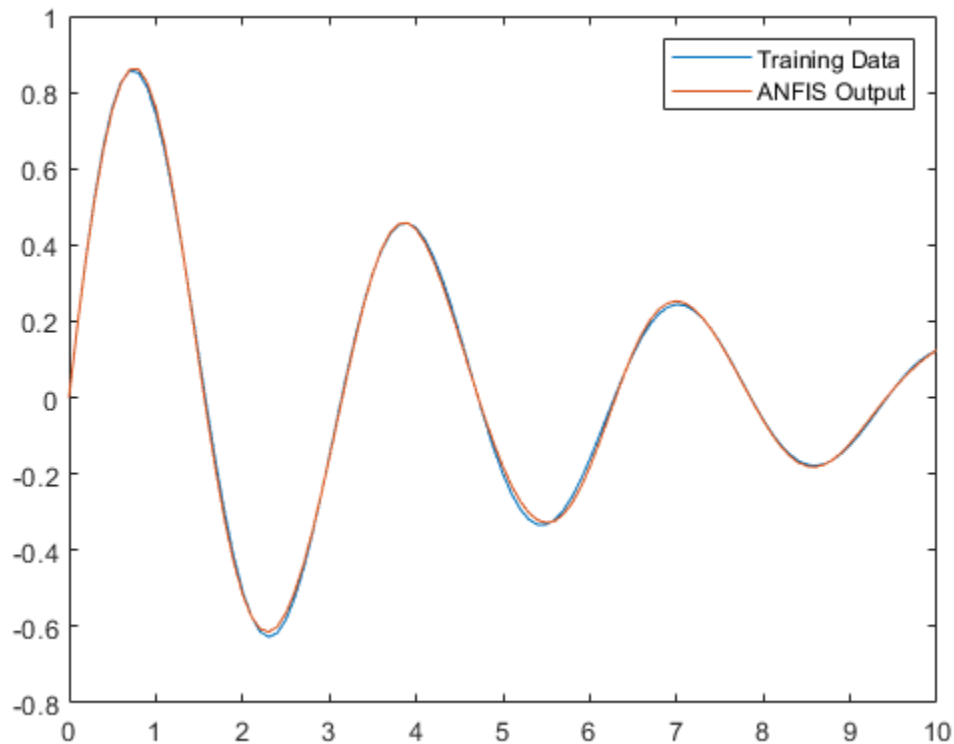
```
opt = anfisOptions('InitialFIS',inFIS);  
opt.DisplayANFISInformation = 0;  
opt.DisplayErrorValues = 0;  
opt.DisplayStepSize = 0;  
opt.DisplayFinalResults = 0;
```

Train the FIS using the specified options.

```
outFIS = anfis([x y],opt);
```

Compare the ANFIS output with the training data.

```
plot(x,y,x,evalfis(outFIS,x))  
legend('Training Data','ANFIS Output')
```



Obtain ANFIS Training Error

Load training data. This data has a single input and a single output.

```
load fuzex2trnData.dat
```

Specify the training options.

```
opt = anfisOptions('InitialFIS',4,'EpochNumber',40);  
opt.DisplayANFISInformation = 0;  
opt.DisplayErrorValues = 0;
```



```
opt.DisplayStepSize = 0;
opt.DisplayFinalResults = 0;
```

Train the FIS, and return the training error.

```
[fis,trainError] = anfis(fuzex2trnData,opt);
```

`trainError` contains the root mean squared error for the training data at each training epoch. The training error for `fis` is the minimum value in `trainError`.

```
fisRMSE = min(trainError)
```

```
fisRMSE = 0.2572
```

Obtain ANFIS Step Size Profile

Create single-input, single-output training data.

```
x = (0:0.1:10)';
y = sin(2*x)./exp(x/5);
```

Configure the ANFIS training options. Set the initial FIS, and suppress the training progress display.

```
opt = anfisOptions('InitialFIS',4,'EpochNumber',60);
opt.DisplayANFISInformation = 0;
opt.DisplayErrorValues = 0;
opt.DisplayStepSize = 0;
opt.DisplayFinalResults = 0;
```

A larger step size increase rate can make the training converge faster. However, increasing the step size increase rate too much can lead to poor convergence. For this example, try doubling the step size increase rate.

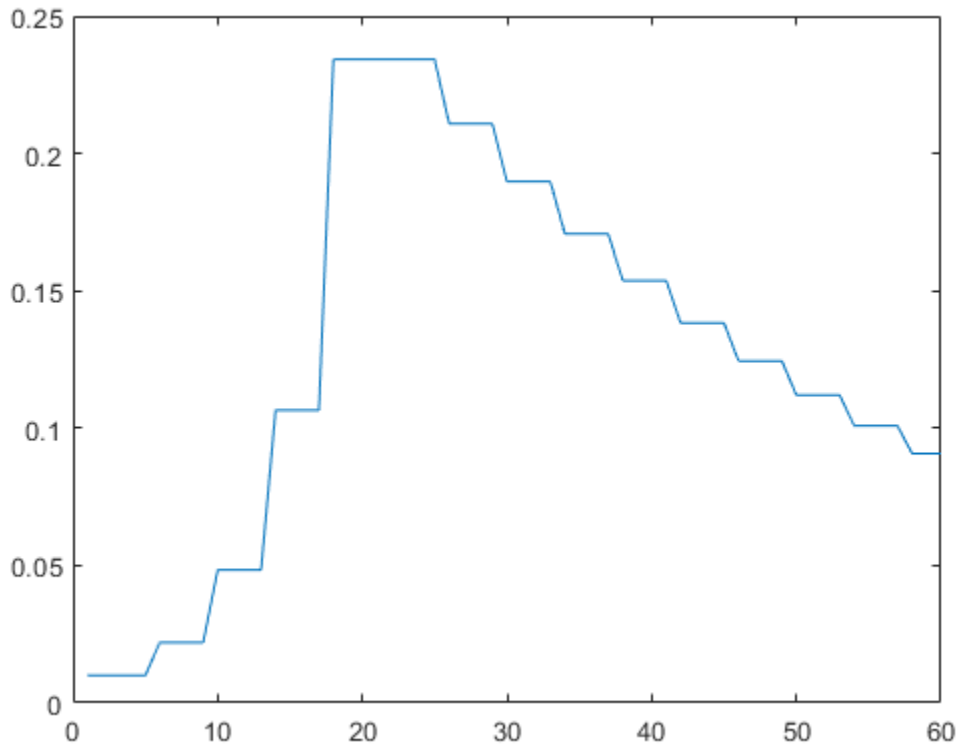
```
opt.StepSizeIncreaseRate = 2*opt.StepSizeIncreaseRate;
```

Train the FIS, and return the step size array.

```
[fis,~,stepSize] = anfis([x y],opt);
```

Plot the step size profile. An optimal step size profile should increase initially, reach a maximum, and then decrease for the rest of the training.

```
figure  
plot(stepSize)
```



Validate ANFIS Training

Load training data.

```
load fuzex1trnData.dat
```

Load validation data.

```
load fuzex1chkData.dat
```

Specify the following training options:

- 4 input membership functions
- 30 training epochs
- Suppress training progress display

```
opt = anfisOptions('InitialFIS',4,'EpochNumber',30);  
opt.DisplayANFISInformation = 0;  
opt.DisplayErrorValues = 0;  
opt.DisplayStepSize = 0;  
opt.DisplayFinalResults = 0;
```

Add the validation data to the training options.

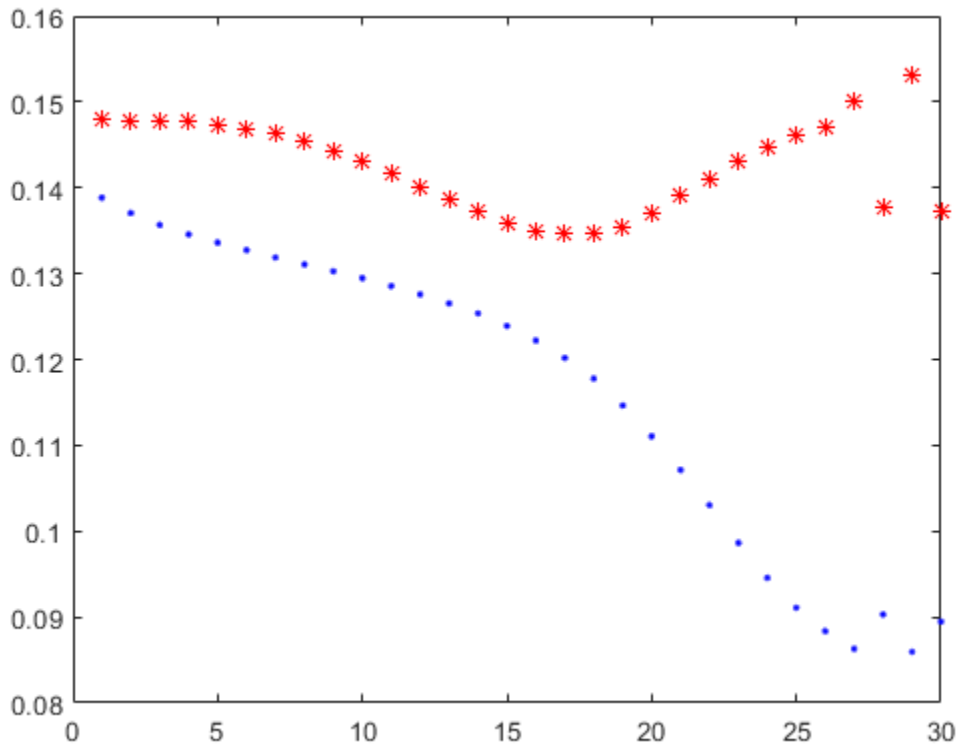
```
opt.ValidationData = fuzexlchkData;
```

Train the FIS, and return the validation results.

```
[fis,trainError,stepSize,chkFIS,chkError] = anfis(fuzexltrnData,opt);
```

The training error, `trainError`, and validation error, `chkError`, arrays each contain one error value per training epoch. Plot the training error and the validation error.

```
x = [1:30];  
plot(x,trainError,'.b',x,chkError,'*r')
```



The minimum validation error occurs at epoch 17. The increase in validation error after this point indicates overfitting of the model parameters to the training data. Therefore, the tuned FIS at epoch 17, `chkFIS`, exhibits the best generalization performance.

Input Arguments

trainingData — Training data

array

Training data, specified as an array. For a fuzzy system with N inputs, specify `trainingData` as an array with $N+1$ columns. The first N columns contain input data,

and the final column contains output data. Each row of `trainingData` contains one data point.

Generally, training data should fully represent the features of the data the FIS is intended to model.

options — Training options

`anfisOptions` option set

Training options, specified as an `anfisOptions` option set. Using `options`, you can specify:

- An initial FIS structure to tune, `options.InitialFIS`.
- Validation data for preventing overfitting to training data, `options.ValidationData`.
- Training algorithm options, such as the maximum number of training epochs, `options.EpochNumber`, or the training error goal, `options.ErrorGoal`.
- Whether to display training progress information, such as the training error values for each training epoch, `options.DisplayErrorValues`.

Output Arguments

fis — Trained fuzzy inference system

`mamfis` object | `sugfis` object

Trained fuzzy inference system with membership function parameters tuned using the training data, returned as a `mamfis` or `sugfis` object. This fuzzy system corresponds to the epoch for which the training error is smallest. If two epochs have the same minimum training error, the FIS from the earlier epoch is returned.

trainError — Root mean square training error

array

Root mean square training error for each training epoch, returned as an array. The minimum value in `trainError` is the training error for fuzzy system `fis`.

stepSize — Training step size

array

Training step size for each epoch, returned as an array. The `anfis` training algorithm tunes the FIS parameters using gradient descent optimization methods. The training step size is the magnitude of the gradient transitions in the parameter space.

Ideally, the step size increases at the start of training, reaches a maximum, and then decreases for the remainder of the training. To achieve this step size profile, adjust the initial step size (`options.InitialStepSize`), step size increase rate (`options.StepSizeIncreaseRate`), and step size decrease rate `options.StepSizeDecreaseRate`.

chkFIS — Tuned FIS for which the validation error is minimum

`mamfis` object | `sugfis` object

Tuned FIS for which the validation error is minimum, returned as a `mamfis` or `sugfis` object. If two epochs have the same minimum validation error, the FIS from the earlier epoch is returned.

`chkFIS` is returned only when you specify validation data using `options.ValidationData`.

chkError — Root mean square validation error

array

Root mean square training error, returned as an array with length equal to the number of training epochs. The minimum value in `chkError` is the training error for fuzzy system `chkFIS`.

`chkError` is returned only when you specify validation data using `options.ValidationData`.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

References

- [1] 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)*. July 1991, pp. 762-767.
- [2] Jang, J.-S. R., "ANFIS: Adaptive-Network-based Fuzzy Inference Systems," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 23, No. 3, May 1993, pp. 665-685.

See Also

Apps

Neuro-Fuzzy Designer

Functions

`anfisOptions` | `genfis`

Topics

“Neuro-Adaptive Learning and ANFIS” on page 3-2

“Comparison of `anfis` and Neuro-Fuzzy Designer Functionality” on page 3-7

“Predict Chaotic Time-Series” on page 3-43

“Modeling Inverse Kinematics in a Robotic Arm” on page 3-61

Introduced before R2006a

anfisOptions

Option set for `anfis` command

Syntax

```
opt = anfisOptions
opt = anfisOptions(Name,Value)
```

Description

`opt = anfisOptions` creates a default option set for tuning a Sugeno fuzzy inference system using `anfis`. Use dot notation to modify this option set for your specific application. Any options that you do not modify retain their default values.

`opt = anfisOptions(Name,Value)` creates an option set with options specified by one or more `Name,Value` pair arguments.

Examples

Create Option Set for ANFIS Training

Create a default option set.

```
opt = anfisOptions;
```

Specify training options using dot notation. For example, specify the following options:

- Initial FIS with 4 membership functions for each input variable
- Maximum number of training epochs equal to 30.

```
opt.InitialFIS = 4;
opt.EpochNumber = 30;
```

You can also specify options when creating the option set using one or more `Name,Value` pair arguments.

```
opt2 = anfisOptions('InitialFIS',4,'EpochNumber',30);
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'EpochNumber',50` sets the maximum number of training epochs to 50.

InitialFIS — Initial FIS structure

2 (default) | positive integer greater than 1 | vector of positive integers | FIS structure

Initial FIS structure to tune, specified as the comma-separated pair consisting of `'InitialFIS'` and one of the following:

- Positive integer greater than 1 specifying the number of membership functions for all input variables. `anfis` generates an initial FIS structure with the specified number of membership functions using `genfis` with grid partitioning.
- Vector of positive integers with length equal to the number of input variables specifying the number of membership functions for each input variable. `anfis` generates an initial FIS structure with the specified numbers of membership functions using `genfis` with grid partitioning.
- FIS structure generated using `genfis` command with grid partitioning or subtractive clustering. The specified system must have the following properties:
 - Single output, obtained using weighted average defuzzification.
 - First or zeroth order Sugeno-type system; that is, all output membership functions must be the same type and be either `'linear'` or `'constant'`.
 - No rule sharing. Different rules cannot use the same output membership function; that is, the number of output membership functions must equal the number of rules.
 - Unity weight for each rule.

- No custom membership functions or defuzzification methods.

EpochNumber — Maximum number of training epochs

10 (default) | positive integer

Maximum number of training epochs, specified as the comma-separated pair consisting of 'EpochNumber' and a positive integer. The training process stops when it reaches the maximum number of training epochs.

ErrorGoal — Training error goal

0 (default) | scalar

Training error goal, specified as the comma-separated pair consisting of 'ErrorGoal' and a scalar. The training process stops when the training error is less than or equal to ErrorGoal.

InitialStepSize — Initial training step size

0.01 (default) | positive scalar

Initial training step size, specified as the comma-separated pair consisting of 'InitialStepSize' and a positive scalar.

The `anfis` training algorithm tunes the FIS parameters using gradient descent optimization methods. The training step size is the magnitude of each gradient transition in the parameter space. Typically, you can increase the rate of convergence of the training algorithm by increasing the step size. During optimization, `anfis` automatically updates the step size using `StepSizeIncreaseRate` and `StepSizeDecreaseRate`.

StepSizeDecreaseRate — Step size decrease rate

0.9 (default) | positive scalar less than 1

Step size decrease rate, specified as the comma-separated pair consisting of 'StepSizeDecreaseRate' and a positive scalar less than 1. If the training error undergoes two consecutive combinations of an increase followed by a decrease, then `anfis` scales the step size by the decrease rate.

StepSizeIncreaseRate — Step size increase rate

1.1 (default) | scalar greater than 1

Step size increase rate, specified as the comma-separated pair consisting of 'StepSizeIncreaseRate' and a scalar greater than 1. If the training error decreases for four consecutive epochs, then `anfis` scales the step size by the increase rate.

DisplayANFISInformation — Flag for showing ANFIS information

1 (default) | 0

Flag for showing ANFIS information at the start of the training process, specified as the comma-separated pair consisting of 'DisplayANFISInformation' and one of the following:

- 1 — Display the following information about the ANFIS system and training data:
 - Number of nodes in the ANFIS system
 - Number of linear parameters to tune
 - Number of nonlinear parameters to tune
 - Total number of parameters to tune
 - Number of training data pairs
 - Number of checking data pairs
 - Number of fuzzy rules
- 0 — Do not display the information.

DisplayErrorValues — Flag for showing training error values

1 (default) | 0

Flag for showing training error values after each training epoch, specified as the comma-separated pair consisting of 'DisplayErrorValues' and one of the following:

- 1 — Display the training error.
- 0 — Do not display the training error.

DisplayStepSize — Flag for showing step size

1 (default) | 0

Flag for showing step size whenever the step size changes, specified as the comma-separated pair consisting of 'DisplayStepSize' and one of the following:

- 1 — Display the step size.
- 0 — Do not display the step size.

DisplayFinalResults — Flag for displaying final results

1 (default) | 0

Flag for displaying final results after training, specified as the comma-separated pair consisting of 'DisplayFinalResults' and one of the following:

- 1 — Display the results.
- 0 — Do not display the results.

ValidationData — Validation data

[] (default) | array

Validation data for preventing overfitting to the training data, specified as the comma-separated pair consisting of 'ValidationData' and an array. For a fuzzy system with N inputs, specify `ValidationData` as an array with $N+1$ columns. The first N columns contain input data and the final column contains output data. Each row of `ValidationData` contains one data point.

At each training epoch, the training algorithm validates the FIS using the validation data.

Generally, validation data should fully represent the features of the data the FIS is intended to model, while also being sufficiently different from the training data to test training generalization.

OptimizationMethod — Optimization method

1 (default) | 0

Optimization method used in membership function parameter training, specified as the comma-separated pair consisting of 'OptimizationMethod' and one of the following:

- 1 — Use a hybrid method, which uses a combination of backpropagation to compute input membership function parameters, and least squares estimation to compute output membership function parameters.
- 0 — Use backpropagation gradient descent to compute all parameters.

Output Arguments

opt — Training options for `anfis` command

`anfisOptions` option set

Training options for `anfis` command, returned as an `anfisOptions` option set.

See Also

anfis | genfis

Introduced in R2017a

convertfis

Convert previous versions of fuzzy inference data in current format

In R2018b, the format of fuzzy inference systems changed from a structure format to an object format. To convert fuzzy systems in an old format to the new format, use `convertfis`.

Syntax

```
fisNew = convertfis(fisOld)
```

Description

`fisNew = convertfis(fisOld)` converts the old-format fuzzy inference system `fisOld` into the current object format.

Examples

Convert Old-Format Fuzzy Inference System

Load a fuzzy inference system created using an old format. For example, load a FIS structure from a MAT-file.

```
load fisStructure
```

View the fields of the structure.

```
fisStructure
```

```
fisStructure = struct with fields:
    name: 'tipper'
    type: 'mamdani'
    andMethod: 'min'
    orMethod: 'max'
    defuzzMethod: 'centroid'
```

```
impMethod: 'min'  
aggMethod: 'max'  
  input: [1x2 struct]  
  output: [1x1 struct]  
  rule: [1x3 struct]
```

Convert the structure to a `mamfis` object and view the object properties.

```
fisObject = convertfis(fisStructure)
```

```
fisObject =  
  mamfis with properties:  
  
          Name: "tipper"  
    AndMethod: "min"  
      OrMethod: "max"  
  ImplicationMethod: "min"  
  AggregationMethod: "max"  
  DefuzzificationMethod: "centroid"  
        Inputs: [1x2 fisvar]  
        Outputs: [1x1 fisvar]  
          Rules: [1x3 fisrule]  
  DisableStructuralChecks: 0
```

Input Arguments

fisOld — Old-format fuzzy inference system

structure | matrix

Old-format fuzzy inference system, specified as a structure or a matrix.

Output Arguments

fisNew — New-format fuzzy inference system

mamfis object | sugfis object

New-format fuzzy inference system, returned as a `mamfis` object or a `sugfis` object.

See Also

mamfis | sugfis

Introduced in R2018b

convertToStruct

Convert fuzzy inference system object into a structure

Syntax

```
fisStructure = convertToStruct(fisObject)
```

Description

`fisStructure = convertToStruct(fisObject)` converts a fuzzy inference system object into a structure.

Examples

Convert FIS Object into Structure

Load a fuzzy inference system.

```
fisObject = readfis('tipper')
```

```
fisObject =  
    mamfis with properties:  
  
                Name: "tipper"  
        AndMethod: "min"  
        OrMethod: "max"  
    ImplicationMethod: "min"  
    AggregationMethod: "max"  
    DefuzzificationMethod: "centroid"  
                Inputs: [1x2 fisvar]  
                Outputs: [1x1 fisvar]  
                Rules: [1x3 fisrule]  
    DisableStructuralChecks: 0
```

Convert the fuzzy inference system object into a structure.

```
fisStructure = convertToStruct(fisObject)
```

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

Input Arguments

fisObject — Fuzzy inference system object

mamfis object | sugfis object

Fuzzy inference system object, specified as a mamfis or sugfis object.

Output Arguments

fisStructure — Fuzzy inference system structure

structure

Fuzzy inference system structure, returned as a structure. The fields of the structure correspond to the properties of the FIS object. For object properties that are themselves objects, the corresponding structure field is a structure.

See Also

mamfis | sugfis

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

convertToSugeno

Transform Mamdani fuzzy inference system into Sugeno fuzzy inference system

Syntax

```
sugenoFIS = convertToSugeno(mamdaniFIS)
```

Description

`sugenoFIS = convertToSugeno(mamdaniFIS)` transforms a Mamdani fuzzy inference system `mamdaniFIS` into a Sugeno fuzzy inference system `sugenoFIS`.

Examples

Transform Mamdani FIS into Sugeno FIS

Load a Mamdani fuzzy inference system.

```
mam_fismat = readfis('mam22.fis');
```

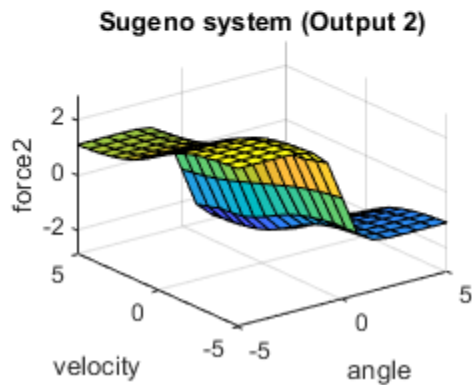
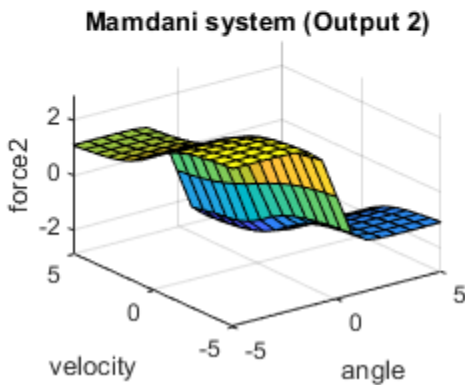
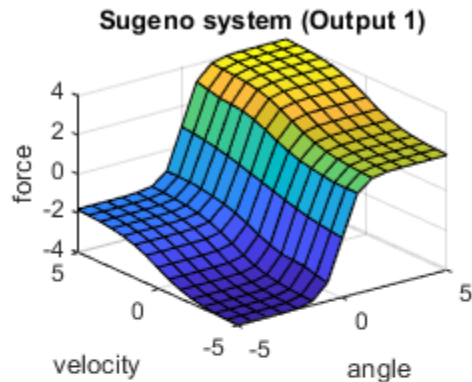
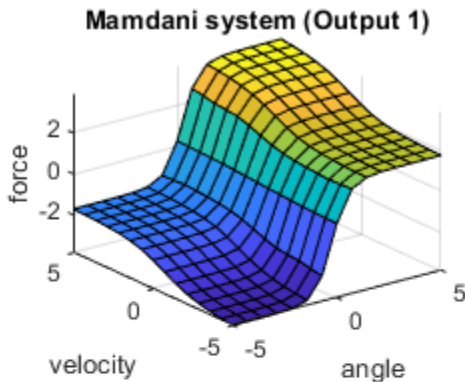
Convert this system to a Sugeno fuzzy inference system.

```
sug_fismat = convertToSugeno(mam_fismat);
```

Plot the output surfaces for both fuzzy systems.

```
subplot(2,2,1)
gensurf(mam_fismat)
title('Mamdani system (Output 1)')
subplot(2,2,2)
gensurf(sug_fismat)
title('Sugeno system (Output 1)')
subplot(2,2,3)
gensurf(mam_fismat,gensurfOptions('OutputIndex',2))
title('Mamdani system (Output 2)')
```

```
subplot(2,2,4)
gensurf(sug_fismat,gensurfOptions('OutputIndex',2))
title('Sugeno system (Output 2)')
```



The output surfaces for both systems are similar.

Input Arguments

mamdaniFIS — Mamdani fuzzy inference system

mamfis object

Mamdani fuzzy inference system, specified as a `mamfis` object. Construct `mamdaniFIS` at the command line or using **Fuzzy Logic Designer**. For more information, see “Build Fuzzy Systems at the Command Line” on page 2-38 and “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14.

Output Arguments

sugenoFIS — Sugeno fuzzy inference system

`sugfis` object

Sugeno fuzzy inference system, returned as a `sugfis` object. `sugenoFIS`:

- Has constant output membership functions, whose values correspond to the centroids of the output membership functions in `mamdaniFIS`
- Uses the weighted-average defuzzification method
- Uses the product implication method
- Uses the sum aggregation method

The remaining properties of `sugenoFIS`, including the input membership functions and rule definitions remain unchanged from `mamdaniFIS`.

Tips

- If you have a functioning Mamdani fuzzy inference system, consider using `convertToSugeno` to convert to a more computationally efficient Sugeno structure to improve performance.
- If `sugFIS` has a single output variable and you have appropriate measured input/output training data, you can tune the membership function parameters of `sugFIS` using `anfis`.

See Also

Functions

`mamfis` | `sugfis`

Apps

Fuzzy Logic Designer

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

“Comparison of Sugeno and Mamdani Systems” on page 2-12

Introduced in R2018b

defuzz

Defuzzify membership function

Syntax

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

Description

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

- 'centroid' — Centroid of the area under the output fuzzy set. This method is the default for Mamdani systems.
- 'bisector' — Bisector of the area under the output fuzzy set
- 'mom' — Mean of the values for which the output fuzzy set is maximum
- 'lom' — Largest value for which the output fuzzy set is maximum
- 'som' — Smallest value for which the output fuzzy set is maximum

You can also specify `type` using a character vector or string that contains the name of a custom function in the current working folder or on the MATLAB path. For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

Examples

Obtain Defuzzified Value

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

out = -3.2857

See Also

Fuzzy Logic Designer

Topics

“Fuzzy Inference Process” on page 1-28

Introduced before R2006a

dsigmf

Difference between two sigmoidal membership functions

Syntax

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

Description

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

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

The membership function `dsigmf` depends on four parameters, a_1 , c_1 , a_2 , and c_2 , and is the difference between two of these sigmoidal functions.

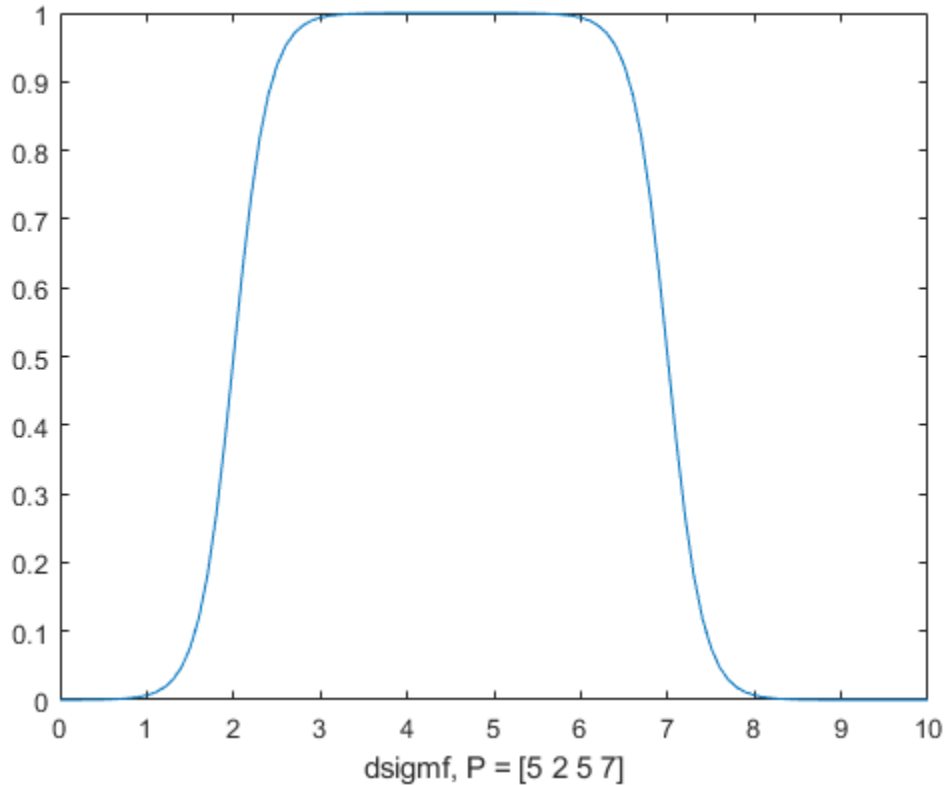
$$f_1(x; a_1, c_1) - f_2(x; a_2, c_2)$$

The parameters are listed in the order: $[a_1 c_1 a_2 c_2]$.

Examples

Obtain Difference of Two Sigmoidal Functions

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



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

dsigmf | evalmf | gauss2mf | gaussmf | gbellmf | mf2mf | pimf | psigmf | sigmf | smf | trapmf | trapmf | trimf | trimf | zmf

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

evalfis

Evaluate fuzzy inference system

Syntax

```
output = evalfis(fis,input)
output = evalfis(fis,input,options)
[output,fuzzifiedIn,ruleOut,aggregatedOut,ruleFiring] = evalfis( ___ )
```

Description

`output = evalfis(fis,input)` evaluates the fuzzy inference system `fis` for the input values in `input` and returns the resulting output values in `output`.

`output = evalfis(fis,input,options)` evaluates the fuzzy inference system using specified evaluation options.

`[output,fuzzifiedIn,ruleOut,aggregatedOut,ruleFiring] = evalfis(___)` returns intermediate results from the fuzzy inference process.

Examples

Evaluate Fuzzy Inference System

Load FIS.

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

Evaluate the FIS when the first input is 2 and the second input is 1.

```
output = evalfis(fis,[2 1])
```

```
output = 7.0169
```

Evaluate FIS for Multiple Input Combinations

Load FIS.

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

Specify the input combinations to evaluate using an array with one row per input combination.

```
input = [2 1;  
         4 5;  
         7 8];
```

Evaluate the FIS for the specified input combinations.

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

```
output = 3×1
```

```
    7.0169  
   14.4585  
   20.3414
```

Each row of `output` is the defuzzified output value for the corresponding row of `input`.

Specify Number of Output Samples for FIS Evaluation

Load FIS.

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

Create an `evalfisOptions` option set, specifying the number of samples in the output fuzzy sets.

```
options = evalfisOptions('NumSamplePoints',50);
```

Evaluate the FIS using this option set.

```
output = evalfis(fis,[2 1],options);
```

Obtain Intermediate Fuzzy Inference Results

Load FIS.

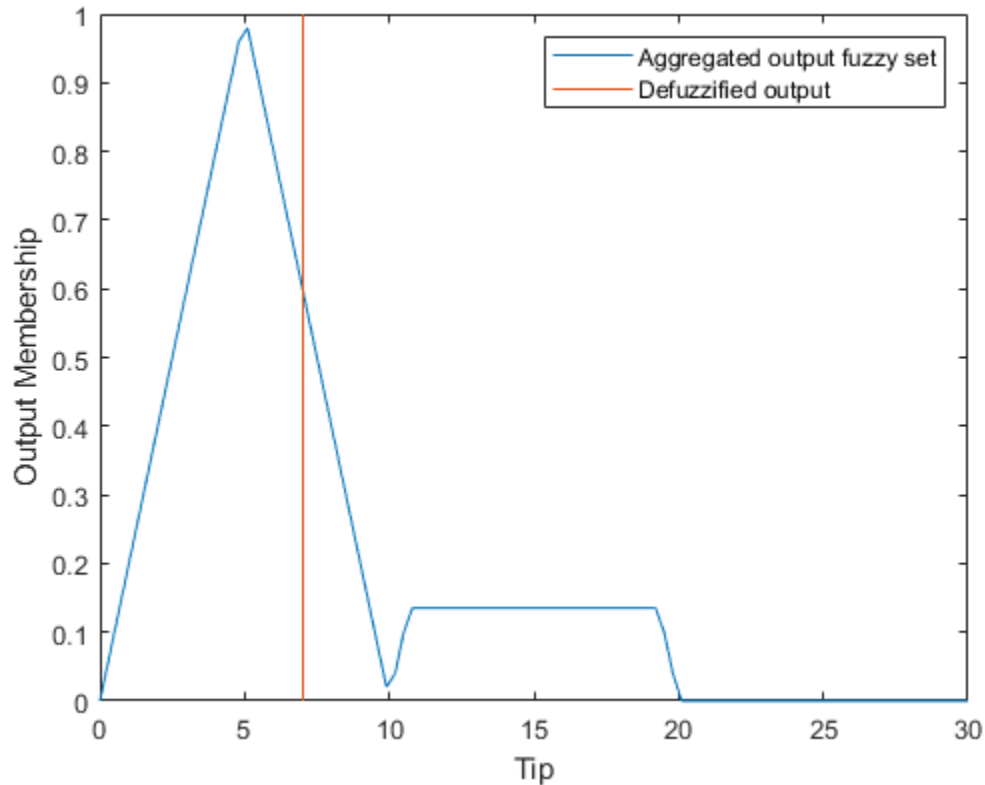
```
fis = readfis('tipper');
```

Evaluate the FIS, and return the intermediate inference results.

```
[output, fuzzifiedIn, ruleOut, aggregatedOut, ruleFiring] = evalfis(fis,[2 1]);
```

You can examine the intermediate results to understand or visualize the fuzzy inference process. For example, view the aggregated output fuzzy set, which is the fuzzy set that `evalfis` defuzzifies to find the output value. Also, plot the defuzzified output value.

```
outputRange = linspace(fis.output.range(1),fis.output.range(2),length(aggregatedOut))';  
plot(outputRange,aggregatedOut,[output output],[0 1])  
xlabel('Tip')  
ylabel('Output Membership')  
legend('Aggregated output fuzzy set','Defuzzified output')
```

The length of `aggregatedOutput` corresponds to the number of sample points used to discretize output fuzzy sets.

Input Arguments

fis — Fuzzy inference system

`mamfis` object | `sugfis` object | homogeneous structure

Fuzzy inference system to be evaluated, specified as a `mamfis` object or `sugfis` object.

To create a fuzzy inference system, you can:

- Use the **Fuzzy Logic Designer** app. For an example, see “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14.
- Automatically generate the FIS object using the `genfis` command.
- Build the system at the command line. For an example, see “Build Fuzzy Systems at the Command Line” on page 2-38.

For code generation applications, you can also specify `fis` as a homogeneous structure create using `getFISCodeGenerationData`. For an example, see “Generate Code for Fuzzy System Using MATLAB Coder” on page 6-12.

input — Input values

M-by-*N_U* array

Input values, specified as an *M*-by-*N_U* array, where *N_U* is the number of input variables in `fis` and *M* is the number of input combinations to evaluate.

`evalfis` supports double-precision or single-precision input values.

options — Evaluation options

`evalfisOptions` object

Evaluation options, specified as an `evalfisOptions` object.

Output Arguments

output — Output values

M-by-*N_Y* array

Output values, returned as an *M*-by-*N_Y* array, where *N_Y* is the number of output variables in `fis`. `evalfis` evaluates `fis` for each row of `input` and returns the resulting defuzzified outputs in the corresponding row of `output`.

fuzzifiedIn — Fuzzified input values

N_R-by-*N_U* array

Fuzzified input values, returned as an *N_R*-by-*N_U* array, where *N_R* is the number of rules in `fis`. Element (*i,j*) of `fuzzifiedIn` is the value of the input membership function for the *j*th input in the *i*th rule.

If `input` specifies multiple input combinations, then `fuzzifiedIn` corresponds to the combination in the last row of `input`.

For more information on fuzzifying input values, see “Fuzzify Inputs” on page 1-29.

ruleOut — Rule outputs

N_S -by- $(N_R N_Y)$ array | N_R -by- N_Y array

Rule outputs, returned as an N_S -by- $(N_R N_Y)$ array or an N_R -by- N_Y array, where N_S is the number of sample points used for evaluating output variable ranges. To obtain the output for each rule, `evalfis` applies the firing strength from the rule antecedent to the output membership function using the implication method specified in `fis`.

For a Mamdani system, each rule output is a fuzzy set. In this case, `ruleOut` is an N_S -by- $(N_R N_Y)$ array. Each column of `ruleOut` contains the output fuzzy set for one rule. The first N_R columns contain the rule outputs for the first output variable, the next N_R columns correspond to the second output variable, and so on.

For a Sugeno system, each rule output is a scalar value. In this case, `ruleOut` is an N_R -by- N_Y array. Element (j,k) of `ruleOut` is the value of the k th output variable for the j th rule.

If `input` specifies multiple input combinations, then `ruleOut` corresponds to the combination in the last row of `input`.

For more information on fuzzy implication, see “Apply Implication Method” on page 1-31 and “What Is Sugeno-Type Fuzzy Inference?” on page 2-5

aggregatedOut — Aggregated output

N_S -by- N_Y array | row vector of length N_Y

Aggregated output for each output variable, returned as an N_S -by- N_Y array or a row vector of length N_Y . For each output variable, `evalfis` combines the corresponding outputs from all the rules using the aggregation method specified in `fis`.

For a Mamdani system, the aggregate result for each output variable is a fuzzy set. In this case, `aggregatedOut` is as an N_S -by- N_Y array. Each column of `aggregatedOut` contains the aggregate fuzzy set for one output variable.

For a Sugeno system, the aggregate result for each output variable is a scalar value. In this case, `aggregatedOut` is a row vector of length N_Y , where element k is the aggregate result for the k th output variable.

If `input` specifies multiple input combinations, then `aggregatedOut` corresponds to the combination in the last row of `input`.

For more information on fuzzy aggregation, see “Aggregate All Outputs” on page 1-31 and “What Is Sugeno-Type Fuzzy Inference?” on page 2-5

ruleFiring — Rule firing strengths

column vector of length N_R

Rule firing strength, returned as a column vector of length N_R . To obtain the firing strength for each rule, `evalfis` evaluates the rule antecedents; that is, it applies fuzzy operator to the values of the fuzzified inputs.

If `input` specifies multiple input combinations, then `ruleFiring` corresponds to the combination in the last row of `input`.

For more information on applying the fuzzy operator, see “Apply Fuzzy Operator” on page 1-30.

Alternative Functionality

App

You can evaluate fuzzy inference systems using the **Rule Viewer** in the **Fuzzy Logic Designer** app.

Simulink Block

You can evaluate fuzzy inference systems using the Fuzzy Logic Controller block. For more information on mapping the arguments of `evalfis` to the Fuzzy Logic Controller block, see “Simulate Fuzzy Inference Systems in Simulink” on page 5-2.

Compatibility Considerations

evalfis input argument order has changed

Behavior changed in R2018b

The order of input arguments for `evalfis` has changed, which requires updates to your code.

Previously, to evaluate a fuzzy inference system, `fis`, you specified the input variable values, `input`, as the first input argument. For example:

```
output = evalfis(input,fis);  
output = evalfis(input,fis,options);
```

Update your code to specify the fuzzy inference system as the first input argument. For example:

```
output = evalfis(fis,input);  
output = evalfis(fis,input,options);
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All `evalfis` syntaxes are supported for code generation. However, `mamfis` and `sugfis` objects are not supported. To use `evalfis` for code generation, you must convert your FIS objects into homogenous structures using `getFISCodeGenerationData`.
- Unlike the Fuzzy Logic Controller, `evalfis` does not support fixed-point data for simulation or code generation.
- When evaluating a fuzzy inference system in Simulink, it is recommended to not use `evalfis` or `evalfisOptions` within a MATLAB Function block. Instead, evaluate your fuzzy inference system using the Fuzzy Logic Controller block.

See Also

Functions

`evalfisOptions` | `mamfis` | `sugfis`

Topics

“Fuzzy Inference Process” on page 1-28

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced before R2006a

evalmf

Evaluate fuzzy membership function

Syntax

```
y = evalmf(mf,x)
```

Description

`y = evalmf(mf,x)` evaluates one or more membership functions in `mf` based on the input values in `x`.

Examples

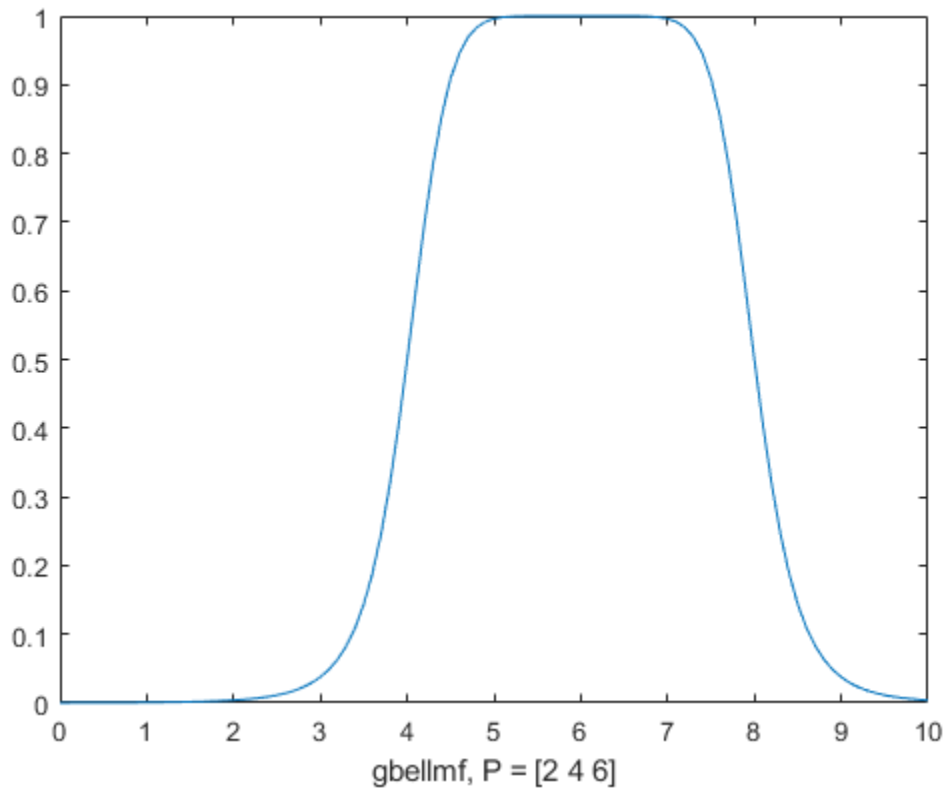
Evaluate Membership Function

Evaluate a generalized bell-shaped membership function across a range of input values from 0 through 10.

```
x = 0:0.1:10;  
mf = fismf("gbellmf",[2 4 6]);  
y = evalmf(mf,x);
```

Plot the evaluation.

```
plot(x,y)  
xlabel('gbellmf, P = [2 4 6]')
```



Evaluate Multiple Membership Functions

Create a vector of three Gaussian membership functions.

```
mf = [fismf("gaussmf",[0.9 2.5],'Name',"low");  
      fismf("gaussmf",[0.9 5],'Name',"medium");  
      fismf("gaussmf",[0.9 7.55],'Name',"high")];
```

Specify the input range over which to evaluate the membership functions.

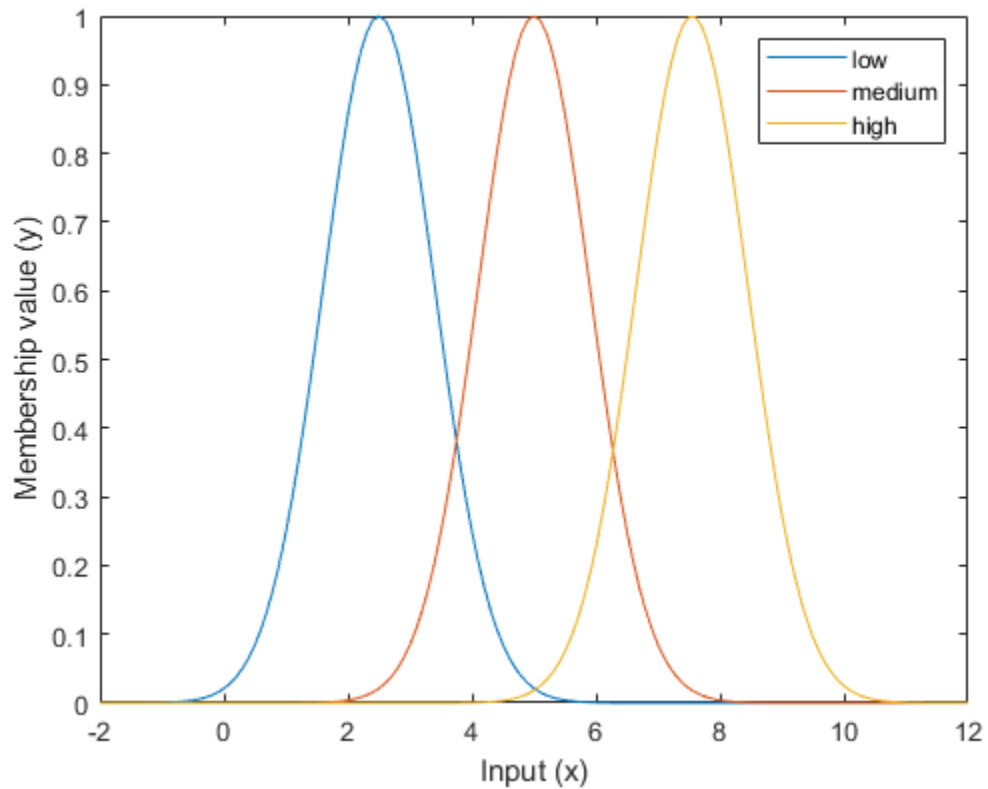
```
x = (-2:0.1:12)';
```


Evaluate the membership functions.

```
y = evalmf(mf,x);
```

Plot the evaluation results.

```
plot(x,y)  
xlabel('Input (x)')  
ylabel('Membership value (y)')  
legend("low","medium","high")
```



Input Arguments

mf — Membership function

`fismf` object | vector of `fismf` objects

Membership function, specified as a `fismf` object or a vector of `fismf` objects.

x — Input value

scalar | vector | 2-D matrix

Input value, specified as a scalar, vector, or 2-D matrix. If `mf` is a:

- Single `fismf` object, then you can specify `x` as a scalar, vector, or matrix.
- Vector of `fismf` objects, then you can specify `x` as a scalar or vector

Output Arguments

y — Output membership value

scalar | vector | 2-D matrix

Output membership value, returned as a scalar, vector, or 2-D matrix. If `mf` is a:

- Single `fismf` object, then `y` is a scalar, vector, or matrix with the same dimensions as `x`. Each element of `y` is the evaluated membership value for the corresponding element of `x`.
- Vector of `fismf` objects, then `y` is an M -by- N matrix, where M and N are the lengths of `mf` and `x`, respectively. $y(i,j)$ is the evaluated value of membership function `mf(i)` for input value `x(j)`.

Compatibility Considerations

`evalmf` now takes a `fismf` object as an input argument

Behavior changed in R2018b

`evalmf` now takes a `fismf` object as an input argument rather than the type and parameters of the membership function. Also, you can now evaluate multiple membership functions by passing an array of `fismf` objects to `evalmf`. There are differences between these approaches that require updates to your code.

Previously, you evaluated a membership function for given input values, `x`, by specifying the type of membership function, `type`, and the membership functions parameters, `params`.

```
y = evalmf(x,params,type);
```

Update your code to first create a `fismf` object, `mf`. Then, pass this object to `evalmf`.

```
mf = fismf(type,params);  
y = evalmf(mf,x);
```

Also, previously, to evaluate multiple membership functions you called `evalmf` once for each membership function.

```
y1 = evalmf(x,params1,type1);  
y2 = evalmf(x,params2,type2);  
y3 = evalmf(x,params3,type3);
```

Now, you can evaluate multiple membership functions by passing an array of `fismf` objects to `evalmf`.

```
mf1 = fismf(type1,params1);  
mf2 = fismf(type2,params2);  
mf3 = fismf(type3,params3);  
y = evalmf([mf1 mf2 mf3],x);
```

Here, `y = [y1 y2 y3]'`;

See Also

`fismf`

Topics

“Foundations of Fuzzy Logic” on page 1-10

Introduced before R2006a

fcm

Fuzzy c-means clustering

Syntax

```
[centers,U] = fcm(data,Nc)
[centers,U] = fcm(data,Nc,options)

[centers,U,objFunc] = fcm( ___ )
```

Description

`[centers,U] = fcm(data,Nc)` performs fuzzy c-means clustering on the given data and returns `Nc` cluster centers.

`[centers,U] = fcm(data,Nc,options)` specifies additional clustering options.

`[centers,U,objFunc] = fcm(___)` also returns the objective function values at each optimization iteration for all of the previous syntaxes.

Examples

Cluster Data Using Fuzzy C-Means Clustering

Load data.

```
load fcmdata.dat
```

Find 2 clusters using fuzzy c-means clustering.

```
[centers,U] = fcm(fcmdata,2);

Iteration count = 1, obj. fcn = 8.970479
Iteration count = 2, obj. fcn = 7.197402
Iteration count = 3, obj. fcn = 6.325579
```

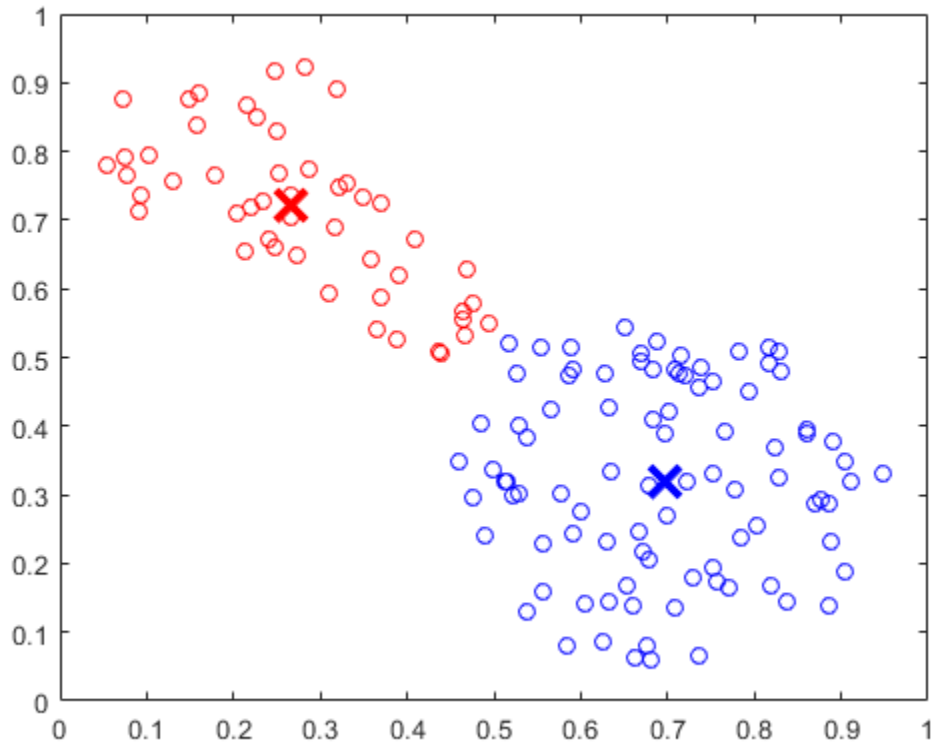
```
Iteration count = 4, obj. fcn = 4.586142
Iteration count = 5, obj. fcn = 3.893114
Iteration count = 6, obj. fcn = 3.810804
Iteration count = 7, obj. fcn = 3.799801
Iteration count = 8, obj. fcn = 3.797862
Iteration count = 9, obj. fcn = 3.797508
Iteration count = 10, obj. fcn = 3.797444
Iteration count = 11, obj. fcn = 3.797432
Iteration count = 12, obj. fcn = 3.797430
```

Classify each data point into the cluster with the largest membership value.

```
maxU = max(U);
index1 = find(U(1,:) == maxU);
index2 = find(U(2,:) == maxU);
```

Plot the clustered data and cluster centers.

```
plot(fcndata(index1,1),fcndata(index1,2),'ob')
hold on
plot(fcndata(index2,1),fcndata(index2,2),'or')
plot(centers(1,1),centers(1,2),'xb','MarkerSize',15,'LineWidth',3)
plot(centers(2,1),centers(2,2),'xr','MarkerSize',15,'LineWidth',3)
hold off
```



Specify Fuzzy Overlap Between Clusters

Create a random data set.

```
data = rand(100,2);
```

To increase the amount of fuzzy overlap between the clusters, specify a large fuzzy partition matrix exponent.

```
options = [3.0 NaN NaN 0];
```

Cluster the data.

```
[centers,U] = fcm(data,2,options);
```

Configure Clustering Termination Conditions

Load the clustering data.

```
load clusterdemo.dat
```

Set the clustering termination conditions such that the optimization stops when either of the following occurs:

- The number of iterations reaches a maximum of 25.
- The objective function improves by less than 0.001 between two consecutive iterations.

```
options = [NaN 25 0.001 0];
```

The first option is NaN, which sets the fuzzy partition matrix exponent to its default value of 2. Setting the fourth option to 0 suppresses the objective function display.

Cluster the data.

```
[centers,U,objFun] = fcm(clusterdemo,3,options);
```

To determine which termination condition stopped the clustering, view the objective function vector.

```
objFun
```

```
objFun = 13×1
```

```
54.7257  
42.9867  
42.8554  
42.1857  
39.0857  
31.6814  
28.5736  
27.1806  
20.7359  
15.7147
```

⋮

The optimization stopped because the objective function improved by less than 0.001 between the final two iterations.

Input Arguments

data — Data set to be clustered

matrix

Data set to be clustered, specified as a matrix with N_d rows, where N_d is the number of data points. The number of columns in **data** is equal to the data dimensionality.

Nc — Number of clusters

integer greater than 1

Number of clusters to create, specified as an integer greater than 1.

options — Clustering options

vector

Clustering options, specified as a vector with the following elements:

Option	Description	Default
options(1)	<p>Exponent for the fuzzy partition matrix, U, specified as a scalar greater than 1.0. This option controls the amount of fuzzy overlap between clusters, with larger values indicating a greater degree of overlap.</p> <p>If your data set is wide with a lot of overlap between potential clusters, then the calculated cluster centers might be very close to each other. In this case, each data point has approximately the same degree of membership in all clusters. To improve your clustering results, decrease this value, which limits the amount of fuzzy overlap during clustering.</p> <p>For an example of fuzzy overlap adjustment, see “Adjust Fuzzy Overlap in Fuzzy C-Means Clustering” on page 4-8.</p>	2.0

Option	Description	Default
options(2)	Maximum number of iterations, specified as a positive integer.	100
options(3)	Minimum improvement in objective function between two consecutive iterations, specified as a positive scalar.	1e-5
options(4)	Information display flag indicating whether to display the objective function value after each iteration, specified as one of the following: <ul style="list-style-type: none"> • true — Display objective function. • false — Do not display objective function. 	true

If any element of `options` is NaN, the default value for that option is used.

The clustering process stops when the maximum number of iterations is reached or when the objective function improvement between two consecutive iterations is less than the specified minimum.

Output Arguments

centers — Cluster centers

matrix

Final cluster centers, returned as a matrix with N_c rows containing the coordinates of each cluster center. The number of columns in `centers` is equal to the dimensionality of the data being clustered.

U — Fuzzy partition matrix

matrix

Fuzzy partition matrix, returned as a matrix with N_c rows and N_d columns. Element $U(i, j)$ indicates the degree of membership of the j th data point in the i th cluster. For a given data point, the sum of the membership values for all clusters is one.

objFunc — Objective function values

vector

Objective function values for each iteration, returned as a vector.

Tips

- To generate a fuzzy inference system using FCM clustering, use the `genfis` command. For example, suppose you cluster your data using the following syntax:

```
[centers,U] = fcm(data,Nc,options);
```

where the first `M` columns of `data` correspond to input variables, and the remaining columns correspond to output variables.

You can generate a fuzzy system using the same training data and FCM clustering configuration. To do so:

- 1 Configure clustering options.

```
opt = genfisOptions('FCMclustering');  
opt.NumClusters = Nc;  
opt.Exponent = options(1);  
opt.MaxNumIteration = options(2);  
opt.MinImprovement = options(3);  
opt.Verbose = options(4);
```

- 2 Extract the input and output variable data.

```
inputData = data(:,1:M);  
outputData = data(:,M+1:end);
```

- 3 Generate the FIS structure.

```
fis = genfis(inputData,outputData,opt);
```

The fuzzy system, `fis`, contains one fuzzy rule for each cluster, and each input and output variable has one membership function per cluster. For more information, see `genfis` and `genfisOptions`.

Algorithms

Fuzzy *c*-means (FCM) is a clustering method that allows each data point to belong to multiple clusters with varying degrees of membership.

FCM is based on the minimization of the following objective function

$$J_m = \sum_{i=1}^D \sum_{j=1}^N \mu_{ij}^m \|x_i - c_j\|^2,$$

where

- D is the number of data points.
- N is the number of clusters.
- m is fuzzy partition matrix exponent for controlling the degree of fuzzy overlap, with $m > 1$. Fuzzy overlap refers to how fuzzy the boundaries between clusters are, that is the number of data points that have significant membership in more than one cluster.
- x_i is the i th data point.
- c_j is the center of the j th cluster.
- μ_{ij} is the degree of membership of x_i in the j th cluster. For a given data point, x_i , the sum of the membership values for all clusters is one.

fcm performs the following steps during clustering:

- 1 Randomly initialize the cluster membership values, μ_{ij} .
- 2 Calculate the cluster centers:

$$c_j = \frac{\sum_{i=1}^D \mu_{ij}^m x_i}{\sum_{i=1}^D \mu_{ij}^m}.$$

- 3 Update μ_{ij} according to the following:

$$\mu_{ij} = \frac{1}{\sum_{k=1}^N \left(\frac{\|x_i - c_j\|}{\|x_i - c_k\|} \right)^{\frac{2}{m-1}}}.$$

- 4 Calculate the objective function, J_m .
- 5 Repeat steps 2-4 until J_m improves by less than a specified minimum threshold or until after a specified maximum number of iterations.

References

- [1] Bezdec, J.C., *Pattern Recognition with Fuzzy Objective Function Algorithms*, Plenum Press, New York, 1981.

See Also

`findcluster` | `genfis`

Topics

“Fuzzy Clustering” on page 4-2

“Cluster Quasi-Random Data Using Fuzzy C-Means Clustering” on page 4-4

“Adjust Fuzzy Overlap in Fuzzy C-Means Clustering” on page 4-8

Introduced before R2006a

findcluster

Open Clustering tool

Syntax

```
findcluster
```

```
findcluster(fileName)
```

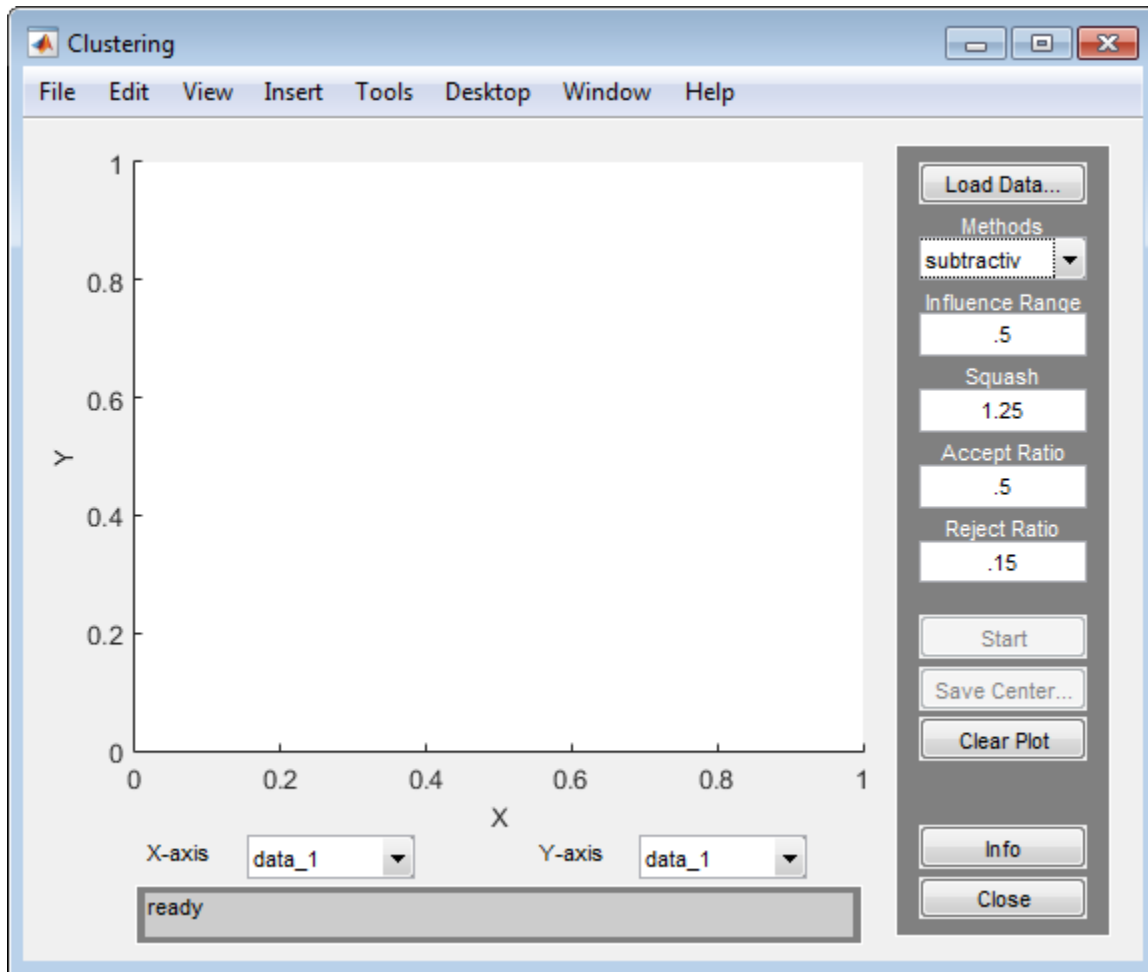
Description

`findcluster` opens a UI to implement either fuzzy c-means or fuzzy subtractive clustering. For more information on:

- Clustering methods, see “Fuzzy Clustering” on page 4-2.
- Using the Clustering tool, see “Data Clustering Using Clustering Tool” on page 4-47.

`findcluster(fileName)` opens the UI, loads the data set in the file `fileName`, specified as a character vector or string, and plots the first two dimensions of the data.

The data set file must have the extension `.dat`. Each line of the data set file contains one data point. For example, if you have 5-dimensional data with 100 data points, the file contains 100 lines, and each line contains five values.



Examples

Open Clustering Tool and Load Data Set

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

Tips

- Using the Clustering tool, you can obtain only the computed cluster centers. To obtain additional information for:
 - Fuzzy c-means clustering, such as the fuzzy partition matrix, cluster the data using `fcm`.
 - Subtractive clustering, such as the range of influence in each data dimension, cluster the data using `subclust`.
- To use the same clustering data with either `fcm` or `subclust`, first load the data file into the MATLAB workspace. For example, at the MATLAB command line, type:

```
load clusterdemo.dat
```

See Also

`fcm` | `subclust`

Topics

“Data Clustering Using Clustering Tool” on page 4-47

Introduced before R2006a

fuzarith

Perform fuzzy arithmetic

Syntax

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

Description

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

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

Note Fuzzy addition might generate the message "divide by zero" but this does not affect the accuracy of this function.

Examples

Perform Fuzzy Arithmetic

Specify Gaussian and Trapezoidal membership functions.

```
N = 101;  
minx = -20;  
maxx = 20;  
x = linspace(minx,maxx,N);
```



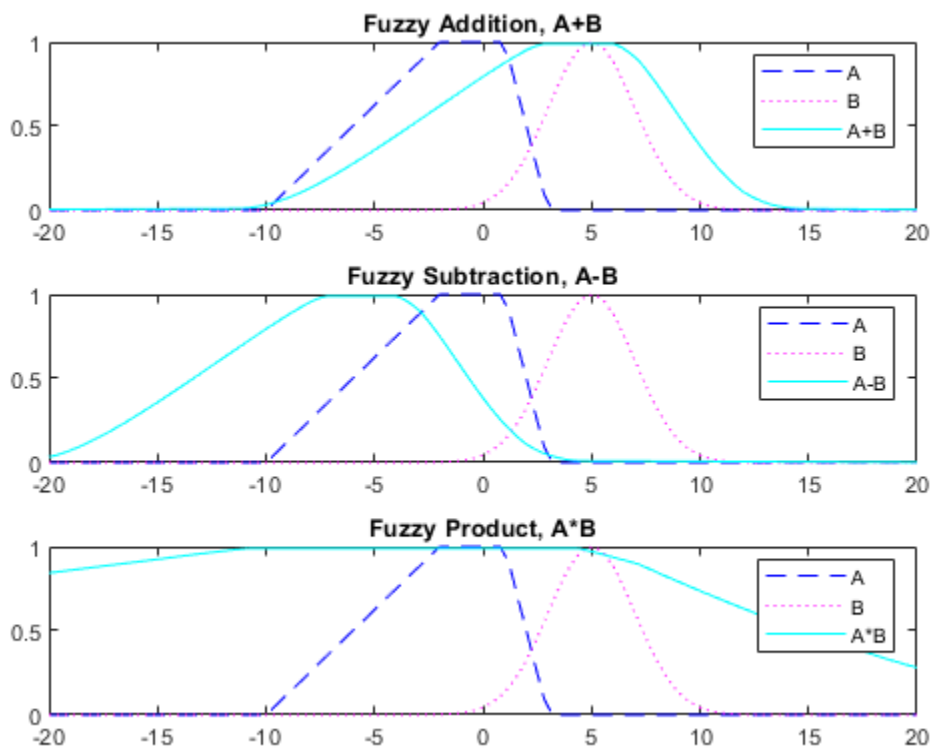
```
A = trapmf(x,[-10 -2 1 3]);  
B = gaussmf(x,[2 5]);
```

Evaluate the sum, difference, and product of A and B.

```
Csum = fuzarith(x,A,B,'sum');  
Csub = fuzarith(x,A,B,'sub');  
Cprod = fuzarith(x,A,B,'prod');
```

Plot the results.

```
figure  
subplot(3,1,1)  
plot(x,A,'b--',x,B,'m:',x,Csum,'c')  
title('Fuzzy Addition, A+B')  
legend('A','B','A+B')  
subplot(3,1,2)  
plot(x,A,'b--',x,B,'m:',x,Csub,'c')  
title('Fuzzy Subtraction, A-B')  
legend('A','B','A-B')  
subplot(3,1,3)  
plot(x,A,'b--',x,B,'m:',x,Cprod,'c')  
title('Fuzzy Product, A*B')  
legend('A','B','A*B')
```



Introduced before R2006a

gauss2mf

Gaussian combination membership function

Syntax

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

Description

The Gaussian function depends on two parameters *sig* and *c* as given by

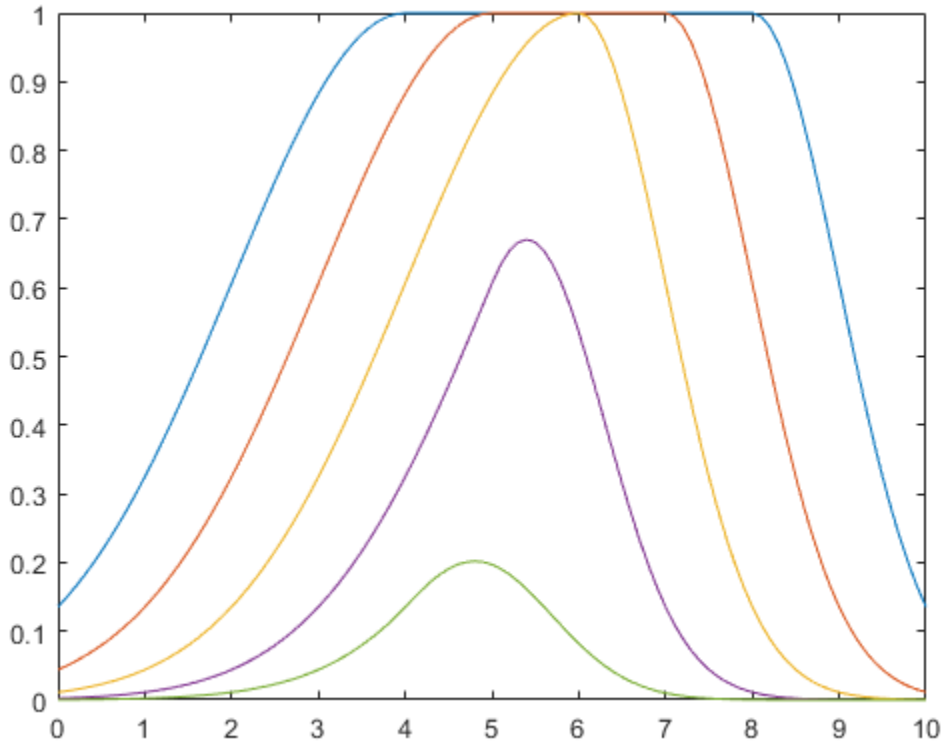
$$f(x; \sigma, c) = e^{-\frac{(x-c)^2}{2\sigma^2}}$$

The function `gauss2mf` is a combination of two of these two parameters. The first function, specified by *sig1* and *c1*, determines the shape of the left-most curve. The second function specified by *sig2* and *c2* determines the shape of the right-most curve. Whenever $c1 < c2$, the `gauss2mf` function reaches a maximum value of 1. Otherwise, the maximum value is less than one. The parameters are listed in the order: `[sig1, c1, sig2, c2]`.

Examples

Gaussian Combination Membership Functions

```
x = [0:0.1:10]';
y1 = gauss2mf(x,[2 4 1 8]);
y2 = gauss2mf(x,[2 5 1 7]);
y3 = gauss2mf(x,[2 6 1 6]);
y4 = gauss2mf(x,[2 7 1 5]);
y5 = gauss2mf(x,[2 8 1 4]);
plot(x,[y1 y2 y3 y4 y5])
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

dsigmf | evalmf | gaussmf | gbellmf | mf2mf | pimf | psigmf | sigmf | smf | trapmf
| trapmf | trimf | trimf | zmf

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

gaussmf

Gaussian curve membership function

The `gaussmf` function lets you define Gaussian membership functions for fuzzy logic variables.

A Gaussian membership function is not the same as a Gaussian probability distribution. For example, a Gaussian membership function always has a maximum value of 1. For more information on Gaussian probability distributions, see “Normal Distribution” (Statistics and Machine Learning Toolbox).

Syntax

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

Description

The symmetric Gaussian function depends on two parameters σ and c as given by

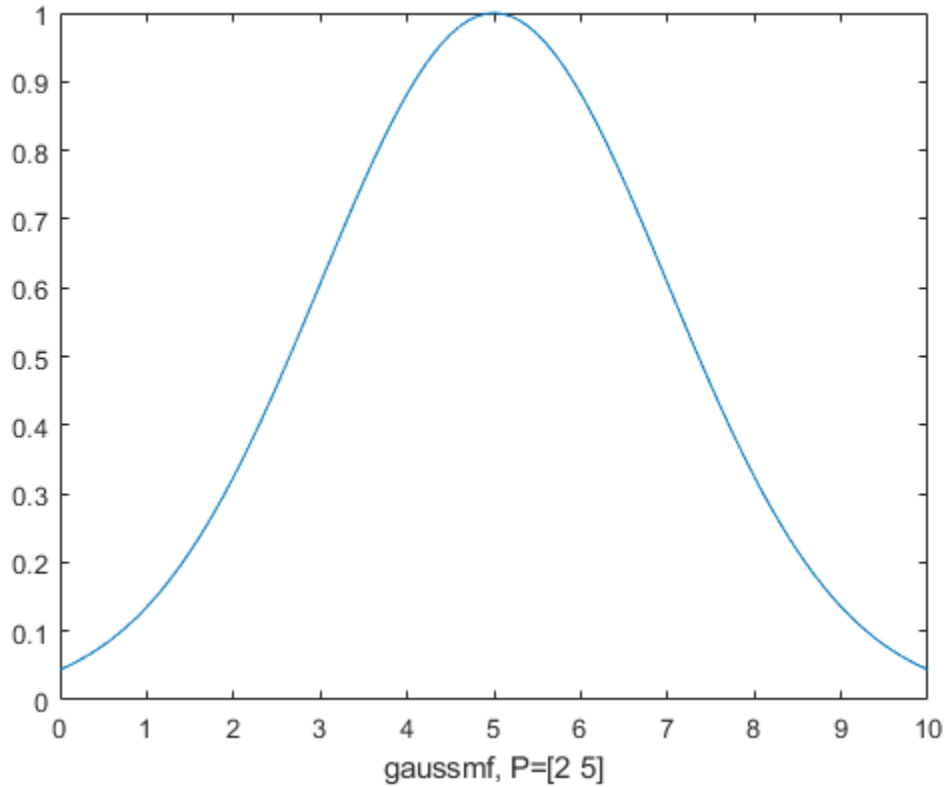
$$f(x;\sigma,c) = e^{-\frac{(x-c)^2}{2\sigma^2}}$$

The parameters for `gaussmf` represent the parameters σ and c listed in order in the vector `[sig c]`.

Examples

Gaussian Membership Function

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



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

gbellmf

Generalized bell-shaped membership function

Syntax

```
y = gbellmf(x,params)
```

Description

The generalized bell function depends on three parameters a , b , and c as given by

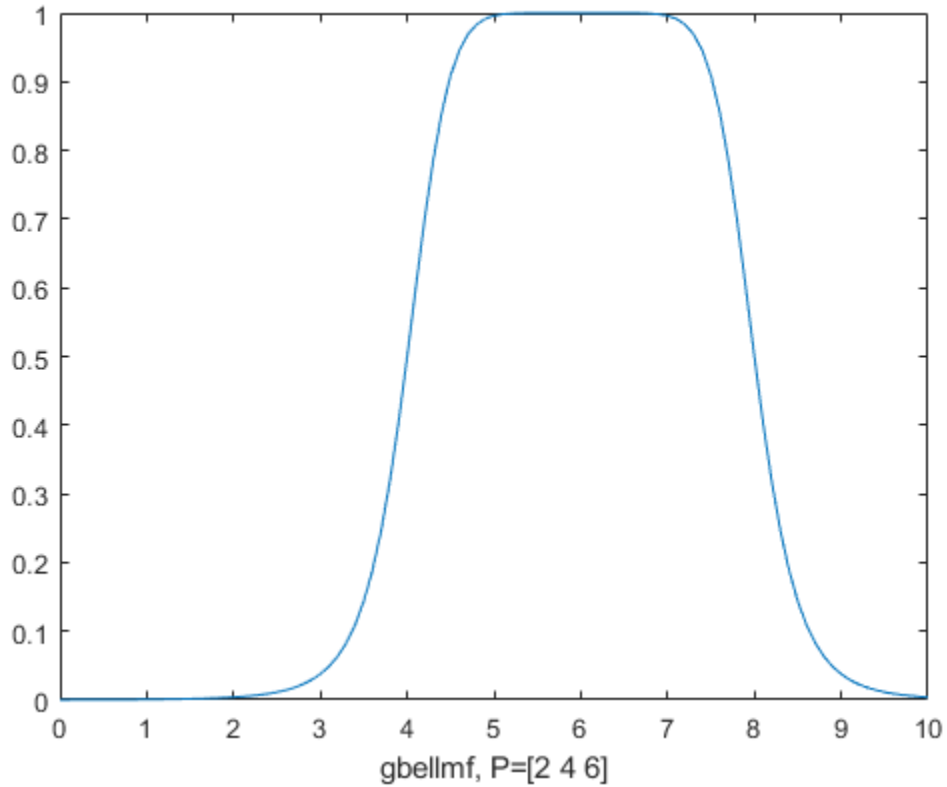
$$f(x;a,b,c) = \frac{1}{1 + \left| \frac{x-c}{a} \right|^{2b}}$$

where the parameter b is usually positive. The parameter c locates the center of the curve. Enter the parameter vector `params`, the second argument for `gbellmf`, as the vector whose entries are a , b , and c , respectively.

Examples

Generalized Bell-Shaped Membership Function

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



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

dsigmf | evalmf | gauss2mf | gaussmf | mf2mf | pimf | psigmf | sigmf | smf |
trapmf | trapmf | trimf | trimf | zmf

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

genfis

Generate fuzzy inference system object from data

Syntax

```
fis = genfis(inputData,outputData)
fis = genfis(inputData,outputData,options)
```

Description

`fis = genfis(inputData,outputData)` returns a single-output Sugeno fuzzy inference system (FIS) using a grid partition of the given input and output data.

`fis = genfis(inputData,outputData,options)` returns an FIS generated using the specified input/output data and options. You can generate fuzzy systems using grid partitioning, subtractive clustering, or fuzzy c-means (FCM) clustering.

Examples

Generate Fuzzy Inference System Using Default Options

Define training data.

```
inputData = [rand(10,1) 10*rand(10,1)-5];
outputData = rand(10,1);
```

Generate a fuzzy inference system.

```
fis = genfis(inputData,outputData);
```

The generated system, `fis`, is created using grid partitioning with default options.

Generate FIS Using Grid Partitioning

Define training data.

```
inputData = [rand(10,1) 10*rand(10,1)-5];  
outputData = rand(10,1);
```

Create a default `genfisOptions` option set for grid partitioning.

```
opt = genfisOptions('GridPartition');
```

Specify the following input membership functions for the generated FIS:

- 3 Gaussian membership functions for the first input variable
- 5 triangular membership functions for the second input variable

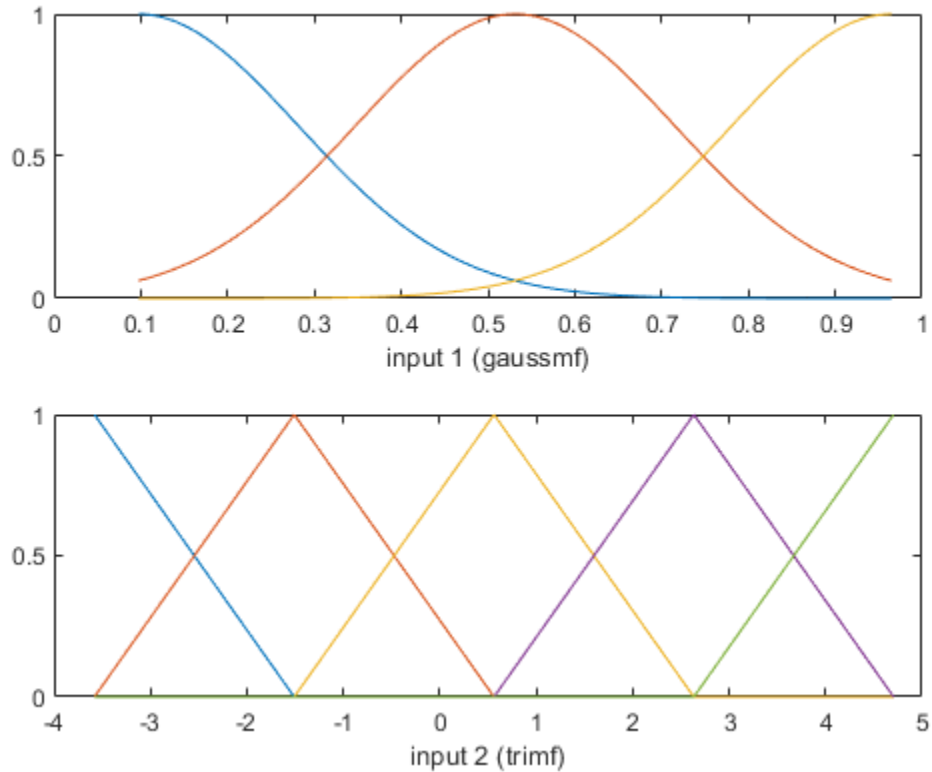
```
opt.NumMembershipFunctions = [3 5];  
opt.InputMembershipFunctionType = ["gaussmf" "trimf"];
```

Generate the FIS.

```
fis = genfis(inputData,outputData,opt);
```

Plot the input membership functions. Each input variable has the specified number and type of input membership functions, evenly distributed over their input range.

```
[x,mf] = plotmf(fis,'input',1);  
subplot(2,1,1)  
plot(x,mf)  
xlabel('input 1 (gaussmf)')  
[x,mf] = plotmf(fis,'input',2);  
subplot(2,1,2)  
plot(x,mf)  
xlabel('input 2 (trimf)')
```



Generate FIS Using Subtractive Clustering

Obtain input and output training data.

```
load clusterdemo.dat
inputData = clusterdemo(:,1:2);
outputData = clusterdemo(:,3);
```

Create a `genfisOptions` option set and specify the range of influence for each data dimension. Specify `0.5` and `0.25` as the range of influence for the first and second input variables. Specify `0.3` as the range of influence for the output data.

```
opt = genfisOptions('SubtractiveClustering',...
                  'ClusterInfluenceRange',[0.5 0.25 0.3]);
```

Generate the FIS.

```
fis = genfis(inputData,outputData,opt);
```

The generated FIS contains one rule for each cluster.

```
showrule(fis)
```

```
ans = 3x83 char array
```

```
'1. If (in1 is in1cluster1) and (in2 is in2cluster1) then (out1 is out1cluster1) ('
```

```
'2. If (in1 is in1cluster2) and (in2 is in2cluster2) then (out1 is out1cluster2) ('
```

```
'3. If (in1 is in1cluster3) and (in2 is in2cluster3) then (out1 is out1cluster3) ('
```

Generate FIS Using FCM Clustering

Obtain the input and output data.

```
load clusterdemo.dat
inputData = clusterdemo(:,1:2);
outputData = clusterdemo(:,3);
```

Create a `genfisOptions` option set for FCM Clustering, specifying a Mamdani FIS type.

```
opt = genfisOptions('FCMClustering','FISType','mamdani');
```

Specify the number of clusters.

```
opt.NumClusters = 3;
```

Suppress the display of iteration information to the Command Window.

```
opt.Verbose = 0;
```

Generate the FIS.

```
fis = genfis(inputData,outputData,opt);
```

The generated FIS contains one rule for each cluster.

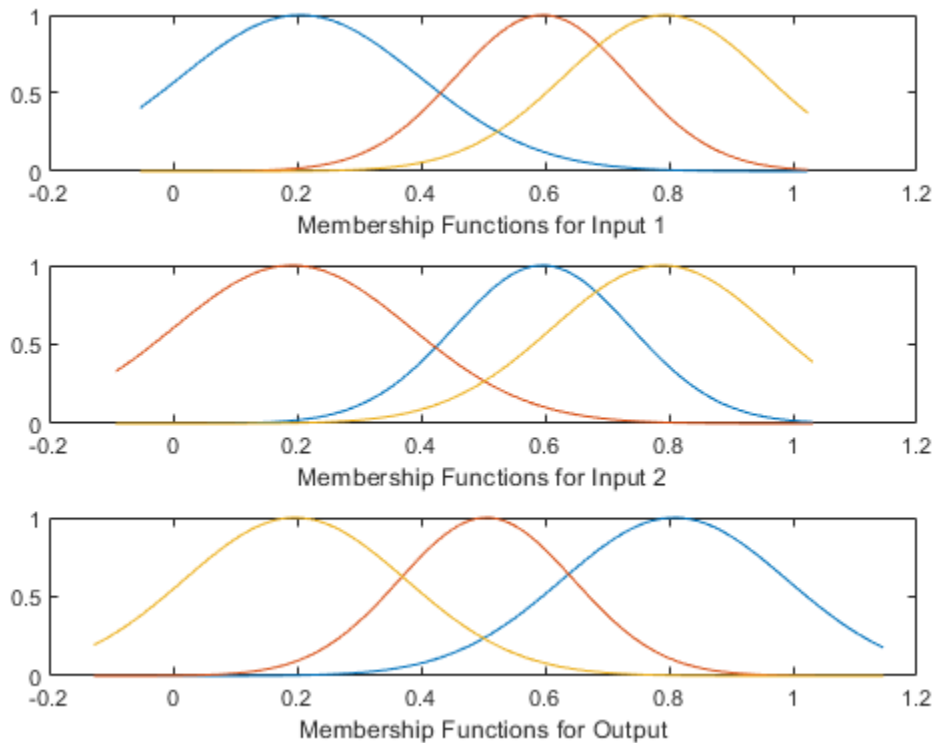
```
showrule(fis)
```

```
ans = 3x83 char array
```

```
  '1. If (in1 is in1cluster1) and (in2 is in2cluster1) then (out1 is out1cluster1) ('  
  '2. If (in1 is in1cluster2) and (in2 is in2cluster2) then (out1 is out1cluster2) ('  
  '3. If (in1 is in1cluster3) and (in2 is in2cluster3) then (out1 is out1cluster3) ('
```

Plot the input and output membership functions.

```
[x,mf] = plotmf(fis,'input',1);  
subplot(3,1,1)  
plot(x,mf)  
xlabel('Membership Functions for Input 1')  
[x,mf] = plotmf(fis,'input',2);  
subplot(3,1,2)  
plot(x,mf)  
xlabel('Membership Functions for Input 2')  
[x,mf] = plotmf(fis,'output',1);  
subplot(3,1,3)  
plot(x,mf)  
xlabel('Membership Functions for Output')
```

Input Arguments

inputData – Input data

array

Input data, specified as an N -column array, where N is the number of FIS inputs.

`inputData` and `outputData` must have the same number of rows.

outputData – Output data

array

Output data, specified as an M -column array, where M is the number of FIS outputs.

When using grid partitioning, `outputData` must have one column. If you specify more than one column for grid partitioning, `genfis` uses the first column as the output data.

`inputData` and `outputData` must have the same number of rows.

options — FIS generation options

`genfisOptions` option set

FIS generation options, specified as a `genfisOptions` option set. If you do not specify options, `genfis` uses a default grid partitioning option set.

You can generate fuzzy systems using one of the following methods, which you specify when you create the option set:

- Grid partitioning — Generate input membership functions by uniformly partitioning the input variable ranges, and create a single-output Sugeno fuzzy system. The fuzzy rule base contains one rule for each input membership function combination.

```
options = genfisOptions('GridPartition');
```

- Subtractive clustering — Generate a Sugeno fuzzy system using membership functions and rules derived from data clusters found using subtractive clustering of input and output data. For more information on subtractive clustering, see `subclust`.

```
options = genfisOptions('SubtractiveClustering');
```

- FCM Clustering — Generate a fuzzy system using membership function and rules derived from data clusters found using FCM clustering of input and output data. For more information on FCM clustering, see `fcm`.

```
options = genfisOptions('FCMClustering');
```

Output Arguments

fis — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, returned as a `mamfis` or `sugfis` object. The properties of `fis` depend on the type of clustering used and the corresponding options.

Clustering Type	Fuzzy System Type	Input Membership Functions	Fuzzy Rules	Output Membership Functions
Grid Partitioning	Sugeno	Each input variable has evenly distributed input membership function. Specify the number of membership functions using <code>options.NumMembershipFunctions</code> . Specify the membership function type using <code>options.InputMembershipFunctionType</code> .	One rule for each input membership function combination. The consequent of each rule corresponds to a different output membership function.	One output membership function for each fuzzy rule. Specify the membership function type using <code>options.OutputMembershipFunctionType</code> .
Subtractive Clustering	Sugeno	Each input variable has one 'gaussmf' input membership function for each fuzzy cluster.	One rule for each fuzzy cluster	Each output variable has one 'linear' output membership function for each fuzzy cluster.
FCM Clustering	Mamdani or Sugeno	Each input variable has one 'gaussmf' input membership function for each fuzzy cluster.	One rule for each fuzzy cluster	Each output variable has one output membership function for each fuzzy cluster. The membership function type is 'gaussmf' for Mamdani systems and 'linear' for Sugeno systems.

If `fis` is a single-output Sugeno system, you can tune the membership function parameters using the `anfis` command.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

`anfis` | `fcm` | `genfisOptions` | `subclust`

Introduced in R2017a

genfis1

(To be removed) Generate Fuzzy Inference System structure from data using grid partition

Note `genfis1` will be removed in a future release. Use `genfis` instead. For more information, see “Compatibility Considerations”.

Syntax

```
fismat = genfis1(data)
```

```
fismat = genfis1(data,numMFs,inmfype,outmfype)
```

Description

`genfis1` generates a Sugeno-type FIS structure used as initial conditions (initialization of the membership function parameters) for `anfis` training.

`genfis1(data)` generates a single-output Sugeno-type fuzzy inference system using a grid partition on the data.

`genfis1(data,numMFs,inmfype,outmfype)` generates an FIS structure from a training data set, `data`, with the number and type of input membership functions and the type of output membership functions explicitly specified.

The arguments for `genfis1` are as follows:

- `data` is the training data matrix, which must be entered with all but the last columns representing input data, and the last column representing the single output.
- `numMFs` is a vector whose coordinates specify the number of membership functions associated with each input. If you want the same number of membership functions to be associated with each input, then specify `numMFs` as a single number.
- `inmfype` is a character array in which each row specifies the membership function type associated with each input. This can be a character vector if the type of membership functions associated with each input is the same.

- `outmfType` is a character vector that specifies the membership function type associated with the output. There can only be one output, because this is a Sugeno-type system. The output membership function type must be either `linear` or `constant`. The number of membership functions associated with the output is the same as the number of rules generated by `genfis1`.

The default number of membership functions, `numMFs`, is 2; the default input membership function type is `'gbellmf'`; and the default output membership function type is `'linear'`. These are used whenever `genfis1` is invoked without the last three arguments.

The following table summarizes the default inference methods.

Inference Method	Default
AND	prod
OR	max
Implication	prod
Aggregation	max
Defuzzification	wtaver

Examples

Generate FIS Using Grid Partitioning

Generate a FIS using grid partitioning.

```
data = [rand(10,1) 10*rand(10,1)-5 rand(10,1)];
numMFs = [3 7];
mfType = char('pimf','trimf');
fismat = genfis1(data,numMFs,mfType);
```

To see the contents of `fismat`, use `showfis(fismat)`.

Plot the FIS input membership functions.

```
[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)')
```

Compatibility Considerations

genfis1 will be removed

Not recommended starting in R2017a

genfis1 will be removed in a future release. Use genfis instead. There are differences between these functions that require updates to your code.

To generate a fuzzy system using grid partitioning, first create a default genfisOptions set.

```
opt = genfisOptions('GridPartition');
```

You can modify the options using dot notation. Any options you do not modify remain at their default values.

Then, update your code to use genfis. For example, if your code has the following form:

```
fis = genfis1(data,numMFs,inmftype,outmftype);
```

Use the following code instead:

```
opt = genfisOptions('GridPartition');  
opt.NumMembershipFunctions = numMFs;  
opt.InputMembershipFunctionType = inmftype;  
opt.OutputMembershipFunctionType = outmftype;  
inputData = data(:,end-1);  
outputData = data(:,end);  
fis = genfis(inputData,outputData,opt);
```

See Also

[anfis](#) | [genfis](#) | [genfis2](#) | [genfis3](#)

Introduced before R2006a

genfis2

(To be removed) Generate Fuzzy Inference System structure from data using subtractive clustering

Note `genfis2` will be removed in a future release. Use `genfis` instead. For more information, see “Compatibility Considerations”.

Syntax

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

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

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

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

Description

`genfis2` generates a Sugeno-type FIS structure using subtractive clustering and requires separate sets of input and output data as input arguments. When there is only one output, `genfis2` may be used to generate an initial FIS for `anfis` training. `genfis2` accomplishes this by extracting a set of rules that models the data behavior.

The rule extraction method first uses the `subclust` function to determine the number of rules and antecedent membership functions and then uses linear least squares estimation to determine each rule's consequent equations. This function returns an FIS structure that contains a set of fuzzy rules to cover the feature space.

The arguments for `genfis2` are as follows:

- `Xin` is a matrix in which each row contains the input values of a data point.
- `Xout` is a matrix in which each row contains the output values of a data point.
- `radii` is a vector that specifies a cluster center's range of influence in each of the data dimensions, assuming the data falls within a unit hyperbox.

For example, if the data dimension is 3 (e.g., X_{in} has two columns and X_{out} has one column), $radii = [0.5 \ 0.4 \ 0.3]$ specifies that the ranges of influence in the first, second, and third data dimensions (i.e., the first column of X_{in} , the second column of X_{in} , and the column of X_{out}) are 0.5, 0.4, and 0.3 times the width of the data space, respectively. If $radii$ is a scalar value, then this scalar value is applied to all data dimensions, i.e., each cluster center has a spherical neighborhood of influence with the given radius.

- $xBounds$ is a 2-by- N optional matrix that specifies how to map the data in X_{in} and X_{out} into a unit hyperbox, where N is the data (row) dimension. The first row of $xBounds$ contains the minimum axis range values and the second row contains the maximum axis range values for scaling the data in each dimension.

For example, $xBounds = [-10 \ 0 \ -1; \ 10 \ 50 \ 1]$ specifies that data values in the first data dimension are to be scaled from the range $[-10 \ +10]$ into values in the range $[0 \ 1]$; data values in the second data dimension are to be scaled from the range $[0 \ 50]$; and data values in the third data dimension are to be scaled from the range $[-1 \ +1]$. If $xBounds$ is an empty matrix or not provided, then $xBounds$ defaults to the minimum and maximum data values found in each data dimension.

- $options$ is an optional vector for specifying algorithm parameters to override the default values. These parameters are explained in the help text for `subclust`. Default values are in place when this argument is not specified.
- $user_centers$ is an optional matrix for specifying custom cluster centers. $user_centers$ has a size of J -by- N where J is the number of clusters and N is the total number of inputs and outputs.

The input membership function type is 'gaussmf', and the output membership function type is 'linear'.

The following table summarizes the default inference methods.

Inference Method	Default
AND	prod
OR	probor
Implication	prod
Aggregation	max
Defuzzification	wtaver

Examples

Specify One Cluster Center Range of Influence For All Data Dimensions

Generate an FIS using subtractive clustering, and specify the cluster center range of influence.

```
Xin = [7*rand(50,1) 20*rand(50,1)-10];  
Xout = 5*rand(50,1);  
fismat = genfis2(Xin,Xout,0.5);
```

`fismat` uses a range of influence of 0.5 for all data dimensions.

To see the contents of `fismat`, use `showfis(fismat)`.

Plot the input membership functions.

```
[x,mf] = plotmf(fismat,'input',1);  
subplot(2,1,1)  
plot(x,mf)  
xlabel('Membership Functions for input 1')  
[x,mf] = plotmf(fismat,'input',2);  
subplot(2,1,2)  
plot(x,mf)  
xlabel('Membership Functions for input 2')
```

Specify Cluster Center Range of Influence For Each Data Dimension

Suppose the input data has two columns, and the output data has one column. Specify 0.5 and 0.25 as the range of influence for the first and second input data columns. Specify 0.3 as the range of influence for the output data.

```
Xin = [7*rand(50,1) 20*rand(50,1)-10];  
Xout = 5*rand(50,1);  
fismat = genfis2(Xin,Xout,[0.5 0.25 0.3]);
```

Specify Data Hyperbox Scaling Range

Suppose the input data has two columns, and the output data has one column. Specify the scaling range for the inputs and outputs to normalize the data into the [0 1] range. The

ranges for the first and second input data columns and the output data are: [-10 +10], [-5 +5], and [0 20].

```
Xin = [7*rand(50,1) 20*rand(50,1)-10];
Xout = 5*rand(50,1);
fismat = genfis2(Xin,Xout,0.5,[-10 -5 0;10 5 20]);
```

Here, the third input argument, 0.5, specifies the range of influence for all data dimensions. The fourth input argument specifies the scaling range for the input and output data.

Compatibility Considerations

genfis2 will be removed

Not recommended starting in R2017a

genfis2 will be removed in a future release. Use genfis instead. There are differences between these functions that require updates to your code.

To generate a fuzzy system using grid partitioning, first create a default genfisOptions set.

```
opt = genfisOptions('SubtractiveClustering');
```

You can modify the options using dot notation. Any options you do not modify remain at their default values.

Then, update your code to use genfis. For example, if your code has the following form:

```
fis = genfis2(inputData,outputData,radii,xBounds,options,userCenters);
```

Use the following code instead:

```
opt = genfisOptions('SubtractiveClustering');
opt.ClusterInfluenceRange = radii;
opt.DataScale = xBounds;
opt.SquashFactor = options(1);
opt.AcceptRatio = options(2);
opt.RejectRatio = options(3);
opt.Verbose = options(4);
```

```
opt.CustomClusterCenters = userCenters;  
fis = genfis(inputData,outputData,opt);
```

See Also

[anfis](#) | [genfis](#) | [genfis1](#) | [genfis3](#) | [subclust](#)

Introduced before R2006a

genfis3

(To be removed) Generate Fuzzy Inference System structure from data using FCM clustering

Note `genfis3` will be removed in a future release. Use `genfis` instead. For more information, see “Compatibility Considerations”.

Syntax

```
fismat = genfis3(Xin,Xout)
fismat = genfis3(Xin,Xout,type)
fismat = genfis3(Xin,Xout,type,cluster_n)
fismat = genfis3(Xin,Xout,type,cluster_n,fcmoptions)
```

Description

`genfis3` generates an FIS using fuzzy c-means (FCM) clustering by extracting a set of rules that models the data behavior. The function requires separate sets of input and output data as input arguments. When there is only one output, you can use `genfis3` to generate an initial FIS for `anfis` training. The rule extraction method first uses the `fcm` function to determine the number of rules and membership functions for the antecedents and consequents.

`fismat = genfis3(Xin,Xout)` generates a Sugeno-type FIS structure (`fismat`) given input data `Xin` and output data `Xout`. The matrices `Xin` and `Xout` have one column per FIS input and output, respectively.

`fismat = genfis3(Xin,Xout,type)` generates an FIS structure of the specified `type`, where `type` is either 'mamdani' or 'sugeno'.

`fismat = genfis3(Xin,Xout,type,cluster_n)` generates an FIS structure of the specified `type` and allows you to specify the number of clusters (`cluster_n`) to be generated by FCM.

The number of clusters determines the number of rules and membership functions in the generated FIS. `cluster_n` must be an integer or 'auto'. When `cluster_n` is 'auto', the function uses the `subclust` algorithm with a `radii` of 0.5 and the minimum and maximum values of `Xin` and `Xout` as `xBounds` to find the number of clusters. See `subclust` for more information.

`fismat = genfis3(Xin,Xout,type,cluster_n,fcoptions)` generates an FIS structure of the specified `type` and number of clusters and uses the specified `fcoptions` for the FCM algorithm. If you omit `fcoptions`, the function uses the default FCM values. See `fcm` for information about these parameters.

The input membership function type is 'gaussmf'. By default, the output membership function type is 'linear'. However, if you specify `type` as 'mamdani', then the output membership function type is 'gaussmf'.

The following table summarizes the default inference methods.

Inference Method	Default
AND	prod
OR	probor
Implication	prod
Aggregation	sum
Defuzzification	wtaver

Examples

Generate Sugeno-Type FIS and Specify Number of Clusters

Obtain the input and output data.

```
Xin = [7*rand(50,1) 20*rand(50,1)-10];
Xout = 5*rand(50,1);
```

Generate a Sugeno-type FIS with 3 clusters.

```
opt = NaN(4,1);
opt(4) = 0;
fismat = genfis3(Xin,Xout,'sugeno',3,opt);
```

The fourth input argument specifies the number of clusters. The fifth input argument, `opt`, specifies the options for the FCM algorithm. The NaN entries of `opt` specify default option values. `opt(4)` turns off the display of iteration information at the command line.

To see the contents of `fismat`, use `showfis(fismat)`.

Plot the input membership functions.

```
[x,mf] = plotmf(fismat,'input',1);
subplot(2,1,1), plot(x,mf)
xlabel('Membership Functions for Input 1')
[x,mf] = plotmf(fismat,'input',2);
subplot(2,1,2), plot(x,mf)
xlabel('Membership Functions for Input 2')
```

Compatibility Considerations

genfis3 will be removed

Not recommended starting in R2017a

`genfis3` will be removed in a future release. Use `genfis` instead. There are differences between these functions that require updates to your code.

To generate a fuzzy system using grid partitioning, first create a default `genfisOptions` set.

```
opt = genfisOptions('FCMClustering');
```

You can modify the options using dot notation. Any options you do not modify remain at their default values.

Then, update your code to use `genfis`. For example, if your code has the following form:

```
fis = genfis3(inputData,outputData,type,cluster_n,fcoptions);
```

Use the following code instead:

```
opt = genfisOptions('FCMClustering');
opt.FISType = type;
opt.NumClusters = cluster_n;
opt.Exponent = fcoptions(1);
```

```
opt.MaxNumIteration = fcmlptions(2);  
opt.MinImprovement = fcmlptions(3);  
opt.Verbose = fcmlptions(4);  
fis = genfis(inputData,outputData,opt);
```

See Also

[anfis](#) | [fcm](#) | [genfis](#) | [genfis1](#) | [genfis2](#)

Introduced before R2006a

genfisOptions

Option set for genfis command

Syntax

```
opt = genfisOptions(clusteringType)
opt = genfisOptions(clusteringType,Name,Value)
```

Description

`opt = genfisOptions(clusteringType)` creates a default option set for generating a fuzzy inference system structure using `genfis`. The option set, `opt`, contains different options that depend on the specified clustering algorithm, `clusteringType`. Use dot notation to modify this option set for your specific application. Options that you do not modify retain their default values.

`opt = genfisOptions(clusteringType,Name,Value)` creates an option set with options specified by one or more `Name,Value` pair arguments.

Examples

Specify Options for FIS Generation

Create a default option set for the grid partitioning generation method.

```
opt = genfisOptions('GridPartition');
```

Modify the options using dot notation. For example, specify 3 membership functions for the first input and 4 membership functions for the second input.

```
opt.NumMembershipFunctions = [3 4];
```

You can also specify options when creating the option set. For example, create an option set for FCM clustering using 4 clusters.

```
opt2 = genfisOptions('FCMClustering','NumClusters',4);
```

Input Arguments

clusteringType — Clustering method

'GridPartition' | 'SubtractiveClustering' | 'FCMClustering'

Clustering method for defining membership functions and fuzzy rules, specified as one of the following:

- 'GridPartition' — Generate input membership functions by uniformly partitioning the input variable ranges, and create a single-output Sugeno fuzzy system. The fuzzy rule base contains one rule for each input membership function combination.
- 'SubtractiveClustering' — Generate a Sugeno fuzzy system using membership functions and rules derived from data clusters found using subtractive clustering of input and output data. For more information on subtractive clustering, see `subclust`.
- 'FCMClustering' — Generate a fuzzy system using membership function and rules derived from data clusters found using FCM clustering of input and output data. For more information on FCM clustering, see `fcm`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'InputMembershipFunctionType','trimf' sets triangular input membership functions for the grid partitioning algorithm.

Grid Partitioning Options

NumMembershipFunctions — Number of input membership functions

2 (default) | integer greater than 1 | vector of integers greater than 1

Number of input membership functions for each input variable, specified as the comma-separated pair consisting of 'NumMembershipFunctions' and one of the following:

- Integer greater than 1 — Specify the same number of membership functions for all inputs.

- Vector of integer greater than 1 with length equal to the number of inputs — Specify a different number of membership functions for each input.

InputMembershipFunctionType — Input membership function type

'gbellmf' (default) | 'gaussmf' | 'trimf' | 'trapmf' | character vector | string array | ...

Input membership function type, specified as the comma-separated pair consisting of 'InputMembershipFunctionType' and one of the following:

- Character vector or string — Specify one of the following membership function types for all inputs.

Membership function type	Description	For more information
'gbellmf'	Generalized bell-shaped membership function	gbellmf
'gaussmf'	Gaussian membership function	gaussmf
'gauss2mf'	Gaussian combination membership function	gauss2mf
'trimf'	Triangular membership function	trimf
'trapmf'	Trapezoidal membership function	trapmf
'sigmf'	Sigmoidal membership function	sigmf
'dsigmf'	Difference between two sigmoidal membership functions	dsigmf
'psigmf'	Product of two sigmoidal membership functions	psigmf
'zmf'	Z-shaped membership function	zmf
'pimf'	Pi-shaped membership function	pimf
'smf'	S-shaped membership function	smf

Membership function type	Description	For more information
Character vector or string	Name of a custom membership function in the current working folder or on the MATLAB path	"Build Fuzzy Systems Using Custom Functions" on page 2-50

- Character array or string array — Specify a different membership function type for each input. For example, specify different membership functions for a three-input system:

```
["gbellmf", "gaussmf", "trimf"]
```

OutputMembershipFunctionType — Output membership function type

'linear' (default) | 'constant'

Output membership function type for a single-output Sugeno system, specified as the comma-separated pair consisting of 'OutputMembershipFunctionType' and one of the following:

- 'linear' — The output of each rule is a linear function of the input variables, scaled by the antecedent result value.
- 'constant' — The output of each rule is a constant, scaled by the antecedent result value.

Subtractive Clustering Options

ClusterInfluenceRange — Range of influence of the cluster center

0.5 (default) | scalar value in the range [0, 1] | vector

Range of influence of the cluster center for each input and output assuming the data falls within a unit hyperbox, specified as the comma-separated pair consisting of 'ClusterInfluenceRange' one of the following:

- Scalar value in the range [0 1] — Use the same influence range for all inputs and outputs.
- Vector — Use different influence ranges for each input and output.

Specifying a smaller range of influence usually creates more and smaller data clusters, producing more fuzzy rules.

DataScale — Data scale factors

'auto' (default) | 2-by-N array

Data scale factors for normalizing input and output data into a unit hyperbox, specified as the comma-separated pair consisting of 'DataScale' and a 2-by- N array, where N is the total number of inputs and outputs. Each column of DataScale specifies the minimum value in the first row and the maximum value in the second row for the corresponding input or output data set.

When DataScale is 'auto', the genfis command uses the actual minimum and maximum values in the data to be clustered.

SquashFactor — Squash factor

1.25 (default) | positive scalar

Squash factor for scaling the range of influence of cluster centers, specified as the comma-separated pair consisting of 'SquashFactor' and a positive scalar. A smaller squash factor reduces the potential for outlying points to be considered as part of a cluster, which usually creates more and smaller data clusters.

AcceptRatio — Acceptance ratio

0.5 (default) | scalar value in the range [0, 1]

Acceptance ratio, defined as a fraction of the potential of the first cluster center, above which another data point is accepted as a cluster center, specified as the comma-separated pair consisting of 'AcceptRatio' and a scalar value in the range [0, 1]. The acceptance ratio must be greater than the rejection ratio.

RejectRatio — Rejection ratio

0.15 (default) | scalar value in the range [0, 1]

Rejection ratio, defined as a fraction of the potential of the first cluster center, below which another data point is rejected as a cluster center, specified as the comma-separated pair consisting of 'RejectRatio' and a scalar value in the range [0, 1]. The rejection ratio must be less than acceptance ratio.

Verbose — Information display flag

false (default) | true

Information display flag indicating whether to display progress information during clustering, specified as the comma-separated pair consisting of 'Verbose' and one of the following:

- false — Do not display progress information.

- `true` — Display progress information.

CustomClusterCenters — Custom cluster centers

`[]` (default) | *C*-by-*N* array

Custom cluster centers, specified the comma-separated pair consisting of 'CustomClusterCenters' and as a *C*-by-*N* array, where *C* is the number of clusters and *N* is the total number of inputs and outputs.

FCM Clustering Options**FISType — Fuzzy inference system type**

'sugeno' (default) | 'mamdani'

Fuzzy inference system type, specified as the comma-separated pair consisting of 'FISType' and one of the following:

- 'sugeno' — Sugeno-type fuzzy system
- 'mamdani' — Mamdani-type fuzzy system

For more information on the types of fuzzy inference systems, see “Types of Fuzzy Inference Systems” on page 2-2.

NumClusters — Number of clusters

'auto' | integer greater than 1

Number of clusters to create, specified as the comma-separated pair consisting of 'NumClusters' and 'auto' or an integer greater than 1. When NumClusters is 'auto', the `genfis` command estimates the number of clusters using subtractive clustering with a cluster influence range of 0.5.

NumClusters determines the number of rules and membership functions in the generated FIS.

Exponent — Exponent for the fuzzy partition matrix

2.0 (default) | scalar greater than 1.0

Exponent for the fuzzy partition matrix, specified as the comma-separated pair consisting of 'Exponent' and a scalar greater than 1.0. This option controls the amount of fuzzy overlap between clusters, with larger values indicating a greater degree of overlap.

If your data set is wide with significant overlap between potential clusters, then the calculated cluster centers can be very close to each other. In this case, each data point

has approximately the same degree of membership in all clusters. To improve your clustering results, decrease this value, which limits the amount of fuzzy overlap during clustering.

For an example of fuzzy overlap adjustment, see “Adjust Fuzzy Overlap in Fuzzy C-Means Clustering” on page 4-8.

MaxNumIteration — Maximum number of iterations

100 (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of 'MaxNumIteration' and a positive integer.

MinImprovement — Minimum improvement in objective function

1e-5 (default) | positive scalar

Minimum improvement in objective function between two consecutive iterations, specified as the comma-separated pair consisting of 'MinImprovement' and a positive scalar.

Verbose — Information display flag

true (default) | false

Information display flag indicating whether to display the objective function value after each iteration, specified as the comma-separated pair consisting of 'Verbose' and one of the following:

- true — Display objective function.
- false — Do not display objective function.

Output Arguments

opt — Option set for genfis command

genfisOptions option set

Option set for genfis command, returned as a genfisOptions option set. The options in the option set depend on the specified clusteringType.

See Also

`fcm` | `genfis` | `subclust`

Introduced in R2017a

gensurf

Generate fuzzy inference system output surface

Syntax

```
gensurf(fis)
gensurf(fis,options)
[X,Y,Z] = gensurf(____)
```

Description

`gensurf(fis)` generates the output surface for the fuzzy inference system, `fis`, plotting the first output variable against the first two input variables. For fuzzy systems with more than two inputs, the remaining input variables use the midpoints of their respective ranges as reference values.

`gensurf(fis,options)` generates the output surface using the specified `options`. To generate a surface using different inputs or outputs, or to specify nondefault plotting options, use this syntax.

`[X,Y,Z] = gensurf(____)` returns the variables that define the output surface for any of the previous syntaxes and suppresses the surface plot.

Examples

Generate FIS Output Surface

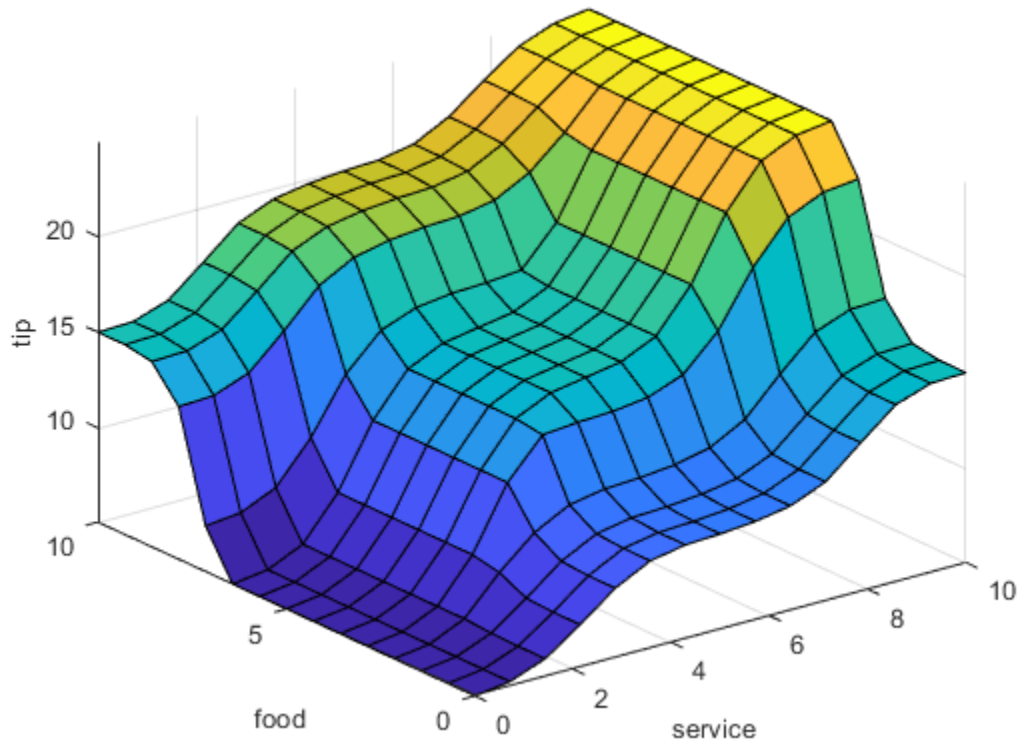
Load a fuzzy inference system.

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

This fuzzy system has two inputs and one output.

Generate the output surface for the system.

```
gensurf(fis)
```



Generate FIS Output Surface for Second Output

Load a fuzzy inference system with two inputs and two outputs.

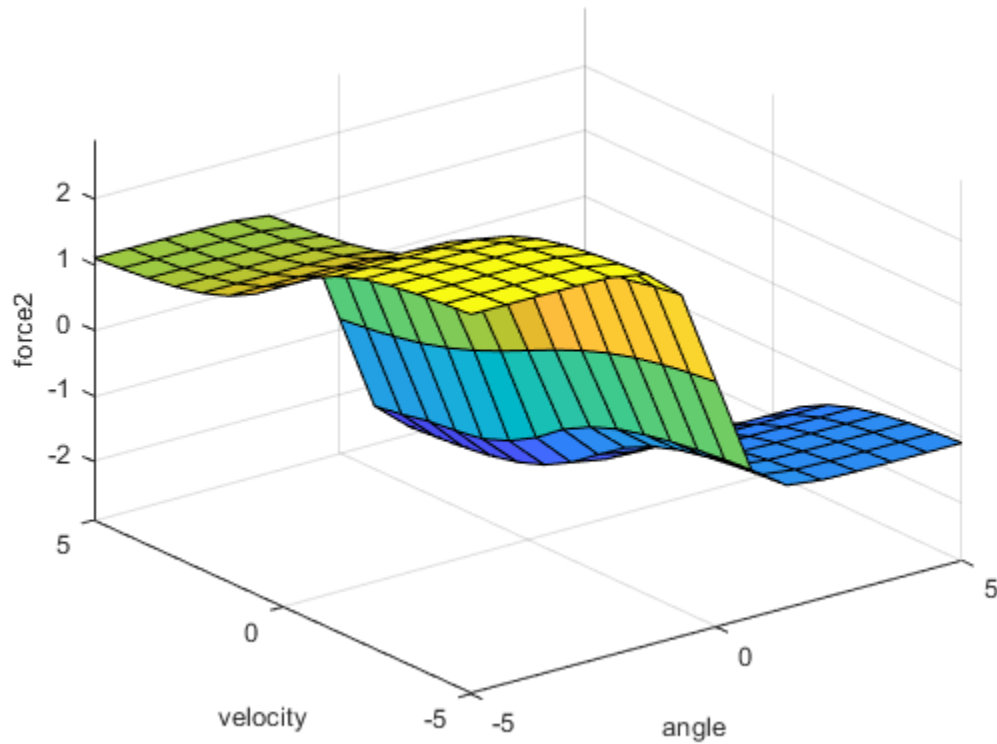
```
fis = readfis('mam22.fis');
```

Create a surface generation option set, specifying the second output as the output to plot. By default, this output is plotted against the first two input variables.

```
opt = gensurfOptions('OutputIndex',2);
```

Plot the surface, using the specified option set.

```
gensurf(fis,opt)
```



Specify Reference Inputs for Surface Plot

Load a fuzzy inference system with four inputs and one output.

```
fis = readfis('slbb.fis');
```

Create a default `gensurfOptions` option set.

```
opt = gensurfOptions;
```

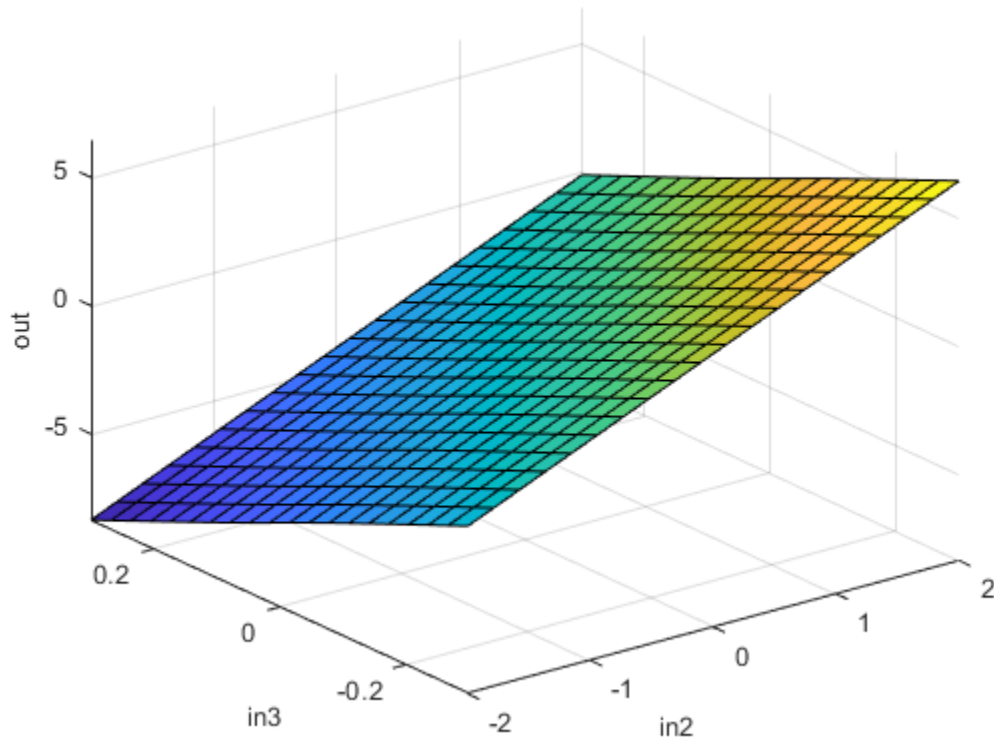
Specify plotting options to:

- Plot the output against the second and third input variable.
- Use 20 grid points for both inputs.
- Fix the first and fourth inputs at -0.5 and 0.1 respectively. Set the reference values for the second and third inputs to NaN.

```
opt.InputIndex = [2 3];  
opt.NumGridPoints = 20;  
opt.ReferenceInputs = [-0.5 NaN NaN 0.1];
```

Plot the output surface.

```
gensurf(fis,opt)
```



Return Surface Values and Suppress Plot

Load a fuzzy inference system.

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

Generate the output surface, returning the surface data.

```
[X,Y,Z] = gensurf(fis);
```

The output values, Z, are the FIS output evaluated at the corresponding X and Y grid points.

Input Arguments

fis — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, specified as a `mamfis` or `sugfis` object.

options — Surface generation options

`gensurfOptions` option set

Surface generation options, specified as a `gensurfOptions` option set.

Output Arguments

X — Grid values for first input variable

array | column vector

Grid values for first input variable, returned as one of the following:

- M -by- N array, where N and M are the number of grid points for the first and second inputs, respectively; that is `options.NumGridPoints = [N M]`. Each column of X contains one grid point value, repeated for every row.
- P -element column vector, where P is the number of grid points specified for a single input variable; that is `options.NumGridPoints = P`. Each element of contains one grid point value. This case applies when `fis` has only one input variable.

Y — Grid values for second input variable

array | []

Grid values for second input variable, returned as one of the following:

- M -by- N array, where N and M are the number of grid points for the first and second inputs respectively; that is `options.NumGridPoints = [N M]`. Each row of Y contains one grid point value, repeated for every column.
- [] when you specify only one input variable; that is, if you specify `options.InputIndex` as an integer.

Z — Surface output values

array | vector

Surface output values for the output variable of `fis` specified by `options.OutputIndex`, returned as one of the following:

- M -by- N array, where N and M are the number of grid points for the first and second inputs respectively; that is `options.NumGridPoints = [N M]`. Each element of Z is the value of the FIS output, evaluated at the corresponding X and Y input values. For example, for a two-input system:

$$Z(i,j) = \text{evalfis}([X(i,j) \ Y(i,j)], \text{fis});$$

- P -element column vector, where P is the number of grid points specified for a single input variable; that is `options.NumGridPoints = P`. Each element of Z is the value of the FIS output evaluated at the corresponding X input value.

When computing the value of Z , `gensurf` sets the values of any inputs not specified by `options.InputIndex` to their corresponding reference values, as specified in `options.ReferenceInputs`.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

`evalfis` | `gensurfOptions` | `surfview`

Introduced before R2006a

gensurfOptions

Option set for gensurf command

Syntax

```
opt = gensurfOptions  
opt = gensurfOptions(Name,Value)
```

Description

`opt = gensurfOptions` creates a default option set for generating a fuzzy inference system output surface using `gensurf`. Use dot notation to modify this option set for your specific application. Any options that you do not modify retain their default values.

`opt = gensurfOptions(Name,Value)` creates an option set with options specified by one or more `Name,Value` pair arguments.

Examples

Specify Options for Generating Output Surface

Create a default `gensurfOptions` option set.

```
opt = gensurfOptions;
```

Specify options using dot notation. For example, for a two-input, three-output fuzzy system, specify options to:

- Plot the surface for the second output against the values of the first and third inputs.
- Specify a reference value of 0.25 for the second input variable.

```
opt.OutputIndex = 2;  
opt.InputIndex = [1 3];  
opt.ReferenceInputs = [NaN 0.25 NaN];
```

Any values you do not specify remain at their default values.

You can also specify one or more options when creating the option set. For example, create an option set, specifying 25 grid points for both plotted input variables:

```
opt2 = gensurfOptions('NumGridPoints',25);
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'InputIndex', [2 3]` plots the output against the second and third input variables using a 3-D surface plot.

InputIndex — Indices of input variables

'auto' (default) | positive integer less than or equal to the number of inputs | two-element vector of positive integers

Indices of input variables to plot the output against, specified as the comma-separated pair consisting of 'InputIndex' and one of the following:

- Positive integer less than or equal to the number of inputs — Plot the output against a single input using a 2-D plot.
- Two-element vector of positive integers — Plot the output against two input variables using a 3-D surface plot.

When `InputIndex` is 'auto', `gensurf` uses the first two input variables by default.

OutputIndex — Index of output variable

'auto' (default) | positive integer less than or equal to the number of outputs

Index of output variable to plot, specified as the comma-separated pair consisting of 'OutputIndex' and a positive integer less than or equal to the number of outputs.

When `OutputIndex` is 'auto', `gensurf` uses the first output variable by default.

NumGridPoints — Number of grid points to plot

15 (default) | integer greater than 1 | two-element vector of integers greater than 1

Number of grid points to plot, specified as the comma-separated pair consisting of 'NumGridPoints' and one of the following:

- Integer greater than 1 — Specify the number of grid points when using a single input variable, or the same number of grid points for both inputs when using two inputs variables.
- Two-element vector of integers greater than 1 — Specify a different number of grid points for each input variable.

If you specify `InputIndex` as an integer and `NumGridPoints` as a vector, then `gensurf` uses the first element of `NumGridPoints` as the number of grid points for the specified input variable.

To plot a smoother surface, increase the number of grid points.

ReferenceInputs — Reference values for input variables

'auto' (default) | vector

Reference values for input variables not shown in the surface plot, specified as the comma-separated pair consisting of 'ReferenceInputs' and a vector with length equal to the number of FIS inputs. Specify NaN for the inputs specified in `InputIndex`.

When `ReferenceInputs` is 'auto', `gensurf` uses the midpoint of the range of each unused variable as a reference value.

NumSamplePoints — Number of sample points

101 (default) | integer greater than 1

Number of sample points to use when evaluating membership functions over the output variable range, specified as the comma-separated pair consisting of 'NumSamplePoints' and an integer greater than 1. For more information on membership function evaluation, see `evalfis`.

Note `NumSamplePoints` is not used by Sugeno-type systems.

Output Arguments

opt — Option set for `gensurf` command

`gensurfOptions` option set

Option set for `gensurf` command, returned as a `gensurfOptions` option set.

See Also

`evalfis` | `gensurf`

Introduced in R2017a

getfis

(To be removed) Get fuzzy system properties

Note `getfis` will be removed in a future release. Access fuzzy inference system properties using dot notation instead. For more information, see “Compatibility Considerations”.

Syntax

```
getfis(sys)
```

```
fisInfo = getfis(sys)  
fisInfo = getfis(sys, fisProperty)
```

```
varInfo = getfis(sys, varType, varIndex)  
varInfo = getfis(sys, varType, varIndex, varProperty)
```

```
mfInfo = getfis(sys, varType, varIndex, 'mf', mfIndex)  
mfInfo = getfis(sys, varType, varIndex, 'mf', mfIndex, mfProperty)
```

Description

`getfis(sys)` prints the properties of the specified fuzzy inference system, `sys`, to the Command Window.

`fisInfo = getfis(sys)` returns the properties of the specified fuzzy inference system.

`fisInfo = getfis(sys, fisProperty)` returns the value of the specified property of the fuzzy inference system.

`varInfo = getfis(sys, varType, varIndex)` returns the properties of the specified input or output variable of a fuzzy inference system.

`varInfo = getfis(sys, varType, varIndex, varProperty)` returns the value of the specified variable property.

`mfInfo = getfis(sys,varType,varIndex,'mf',mfIndex)` returns the properties of the specified membership function of an input or output variable.

`mfInfo = getfis(sys,varType,varIndex,'mf',mfIndex,mfProperty)` returns the value of the specified membership function property.

Examples

Display Properties of Fuzzy Inference System

Load a fuzzy inference system.

```
sys = readfis('tipper');
```

Display the system properties.

```
getfis(sys)

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
```

Obtain Fuzzy Inference System Properties

Load fuzzy system.

```
sys = readfis('tipper');
```

Obtain the system properties.

```
prop = getfis(sys);
```

To obtain the value of a given property, specify the property name. For example, obtain the type of the fuzzy system.

```
type = getfis(sys, 'type');
```

Obtain Variable Properties

Load fuzzy system.

```
sys = readfis('tipper');
```

Obtain the properties of the first input variable.

```
prop = getfis(sys, 'input', 1);
```

To obtain the value of a given property, specify the property name. For example, obtain the range of the output variable.

```
range = getfis(sys, 'output', 1, 'range');
```

Obtain Membership Function Properties

Load fuzzy system.

```
sys = readfis('tipper');
```

For the second input variable, obtain the properties of its first membership function.

```
prop = getfis(sys, 'input', 2, 'mf', 1);
```

To obtain the value of a given property, specify the property name. For example, obtain the parameters of the second membership function of the output variable.

```
params = getfis(sys,'output',1,'mf',2,'params');
```

Input Arguments

sys — Fuzzy inference system

FIS structure

Fuzzy inference system, specified as an FIS structure.

fisProperty — Fuzzy inference system property

'name' | 'type' | 'numInputs' | 'numOutputs' | ...

Fuzzy inference system property, specified as one of the following:

- 'name' — FIS name
- 'type' — FIS type
- 'numInputs' — Number of inputs
- 'numOutputs' — Number of outputs
- 'numRules' — Number of fuzzy rules.
- 'andMethod' — And method
- 'orMethod' — Or method
- 'defuzzMethod' — Defuzzification method
- 'impMethod' — Implication method
- 'aggMethod' — Aggregation method
- 'ruleList' — List of fuzzy rules

varType — Variable type

'input' | 'output'

Variable type, specified as either 'input' or 'output', for input and output variables, respectively.

varIndex — Variable index

positive integer

Variable index, specified as a positive integer.

varProperty — Variable property

'name' | 'range' | 'nummfs'

Variable property, specified as one of the following:

- 'name' — Variable name
- 'range' — Variable value range
- 'nummfs' — Number of membership functions

mfIndex — Membership function index

positive integer

Membership function index, specified as a positive integer.

mfProperty — Membership function property

'name' | 'type' | 'params'

Membership function property, specified as one of the following:

- 'name' — Membership function name
- 'type' — Membership function type
- 'params' — Membership function parameters

For more information on membership functions, see “Membership Functions” on page 1-14.

Output Arguments

fisInfo — Fuzzy inference system information

structure | character vector | nonnegative integer | array

Fuzzy inference system information, returned as a structure, character vector, nonnegative integer, or array, depending on the value of `fisProperty`.

If you do not specify `fisProperty`, then `fisInfo` is returned as a structure with the following fields.

Field	Description
name	FIS name, returned as a character vector.
type	FIS type, returned as a character vector.
andMethod	AND fuzzy operator method, returned as a character vector.
orMethod	OR fuzzy operator method, returned as a character vector.
defuzzMethod	Defuzzification method, returned as a character vector.
impMethod	Implication method, returned as a character vector.
aggMethod	Aggregation method, returned as a character vector.
input	Input variable information, returned as a structure or structure array. Each input variable structure contains the following fields: <ul style="list-style-type: none"> • name — Variable name • range — Variable range • mf — Membership function names
output	Output variable information, returned as a structure or structure array. Each output variable structure contains the following fields: <ul style="list-style-type: none"> • name — Variable name • range — Variable range • mf — Membership function names
rule	Fuzzy rule list, returned as a structure or structure array. Each rule structure contains the following fields: <ul style="list-style-type: none"> • antecedent — Input membership function indices • consequent — Output membership function indices • weight — Rule weight • connection — Fuzzy operator: 1 (AND), 2 (OR)

Otherwise, the value of `fisInfo` depends on the value of `fisProperty` according to the following table.

fisProperty	fisInfo
'name'	FIS name, returned as a character vector.

fisProperty	fisInfo
'type'	FIS type, returned as one of the following: <ul style="list-style-type: none"> • 'mamdani' — Mamdani-type fuzzy system • 'sugeno' — Sugeno-type fuzzy system
'numinputs'	Number of input variables, returned as a nonnegative integer.
'numioutputs'	Number of output variables, returned as a nonnegative integer.
'numrules'	Number of fuzzy rules, returned as a nonnegative integer.
'andmethod'	AND fuzzy operator method, returned as one of the following: <ul style="list-style-type: none"> • 'min' — Minimum of fuzzified input values • 'prod' — Product of fuzzified input values • Character vector — Name of a custom AND function in the current working folder or on the MATLAB path
'ormethod'	OR fuzzy operator method, returned as one of the following: <ul style="list-style-type: none"> • 'max' — Maximum of fuzzified input values • 'probor' — Probabilistic OR of fuzzified input values • Character vector — Name of a custom OR function in the current working folder or on the MATLAB path

fisProperty	fisInfo
'defuzzmethod'	<p>Defuzzification method for computing crisp output values, returned as one of the following for Mamdani systems:</p> <ul style="list-style-type: none"> • 'centroid' — Centroid of the area under the output fuzzy set • 'bisector' — Bisector of the area under the output fuzzy set • 'mom' — Mean of the values for which the output fuzzy set is maximum • 'lom' — Largest value for which the output fuzzy set is maximum • 'som' — Smallest value for which the output fuzzy set is maximum <p>For more information on defuzzification methods for Mamdani systems, see “Defuzzification Methods”.</p> <p>For Sugeno systems, specify the defuzzification method as one of the following:</p> <ul style="list-style-type: none"> • 'wtaver' — Weighted average of all rule outputs • 'wtsum' — Weighted sum of all rule outputs <p>The defuzzification method can also be returned as a character vector that contains the name of a custom defuzzification function in the current working folder or on the MATLAB path.</p>
'impmethod'	<p>Implication method for computing consequent fuzzy set, returned as one of the following:</p> <ul style="list-style-type: none"> • 'min' — Truncate the consequent membership function at the antecedent result value. • 'prod' — Scale the consequent membership function by the antecedent result value. • Character vector — Name of a custom implication function in the current working folder or on the MATLAB path

fisProperty	fisInfo
'aggmethod'	<p>Aggregation method for combining rule consequents, returned as one of the following:</p> <ul style="list-style-type: none"> • 'max' — Maximum of consequent fuzzy sets • 'sum' — Sum of consequent fuzzy sets • 'probor' — Probabilistic OR of consequent fuzzy sets • Character vector — Name of a custom aggregation function in the current working folder or on the MATLAB path.
'rulelist'	<p>Fuzzy rule list, returned as an array. For each fuzzy rule, the rule list contains one row with the following columns:</p> <ul style="list-style-type: none"> • N_u columns of input membership function indices, where N_u is the number of inputs. If a given variable is not included in a rule, the corresponding column entry is 0. Negative values indicate a NOT operation. • N_y columns of output membership function indices, where N_y is the number of outputs. If a given variable is not included in a rule, the corresponding column entry is 0. Negative values indicate a NOT operation. • Rule weight • Fuzzy operator: 1 (AND), 2 (OR)

varInfo — Variable information

structure | character vector | nonnegative integer | row vector of length 2

Variable information, returned as a structure, nonnegative integer, character vector, or row vector, depending on the value of varProperty.

If you do not specify varProperty, then varInfo is returned as a structure with the following fields.

Field	Description
Name	Variable name, returned as a character vector.
NumMFs	Number of membership functions, returned as a nonnegative integer.

Field	Description
mf1, mf2, ..., mfN	Membership function names, returned as character vectors. mfInfo contains one field for each membership function.
range	Variable range, returned as a row vector of length 2.

Otherwise, the value of varInfo depends on the value of varProperty according to the following table.

varProperty	varInfo
'name'	Variable name, returned as a character vector.
'nummfs'	Number of membership functions, returned as a nonnegative integer.
'range'	Variable range, returned as a row vector of length 2.

mfInfo — Membership function information

structure | character vector | row vector

Membership function information, returned as a structure, character vector, or row vector, depending on the value of mfProperty.

If you do not specify mfProperty, then mfInfo is returned as a structure with the following fields.

Field	Description
Name	Membership function name, returned as a character vector.
Type	Membership function type, returned as a character vector.
params	Membership function parameters, returned as a row vector.

Otherwise, the value of mfInfo depends on the value of mfProperty according to the following table.

mfProperty	mfInfo
'name'	Membership function name, returned as a character vector.
'type'	Membership function type, returned as a character vector.
'params'	Membership function parameters, returned as a row vector.

For more information on membership function, see “Membership Functions” on page 1-14.

Compatibility Considerations

getfis will be removed

Not recommended starting in R2018b

getfis will be removed in a future release. Access fuzzy inference system properties using dot notation instead. There are differences between these approaches that require updates to your code.

This table shows some typical usages of getfis for accessing fuzzy inference system properties and how to update your code to use dot notation instead.

If your code has this form:	Use this code instead:
get(fis, 'andmethod')	fis.AndMethod
getfis(fis, 'input', 1)	fis.Inputs(1)
getfis(fis, 'input', 1, 'name')	fis.Inputs(1).Name
getfis(fis, 'input', 2, 'mf', 1)	fis.Inputs(2).MembershipFunctions(1)
getfis(fis, 'input', 2, 'mf', 1, ... params)	fis.Inputs(2).MembershipFunctions(1).Parameters

Previously, fuzzy inference systems were represented as structures. Now, fuzzy inference systems are represented as objects. Fuzzy inference system object properties have different names than the corresponding structure fields. For more information on fuzzy inference system objects, see `mamfis` and `sugfis`.

See Also

setfis | showfis

Introduced before R2006a

getFISCodeGenerationData

Create homogeneous fuzzy inference system structure

To generate code for evaluating a fuzzy inference system using MATLAB Coder, you must convert your fuzzy inference system object into a homogeneous structure using `getFISCodeGenerationData`.

Syntax

```
fisOut = getFISCodeGenerationData(fisIn)
```

Description

`fisOut = getFISCodeGenerationData(fisIn)` converts a fuzzy inference system `fisIn` into a homogeneous structure `fisOut`. `fisIn` can be a FIS object or the name of a `.fis` file.

Examples

Convert FIS Object into Homogeneous Structure

Create a fuzzy inference system. For this example, load a fuzzy system from a file

```
fisObject = readfis('tipper');
```

Convert the resulting `mamfis` object into a homogeneous structure.

```
fisStructure = getFISCodeGenerationData(fisObject);
```

In this structure, if a field is a structure array, all the elements of that array are the same size. For example, consider the elements of input variable array `fisStructure.input`.

```
fisStructure.input(1)
```

```
ans = struct with fields:
    name: 'service'
```



```

origNameLength: 7
    range: [0 10]
    mf: [1x3 struct]
origNumMF: 3

```

```

fisStructure.input(2)

```

```

ans = struct with fields:
    name: 'food'
    origNameLength: 4
    range: [0 10]
    mf: [1x3 struct]
    origNumMF: 2

```

The name fields are character vectors of the same length. Also, even though the second input variable has only two membership functions, the mf fields both contain three membership function structures. The original number of membership functions for a given input variable are stored in the origNumMF field.

Load Fuzzy Inference System from File into Homogenous Structure

Load the fuzzy inference system saved in the file `tipper.fis` into a homogeneous structure.

```

fis = getFISCodeGenerationData('tipper.fis');

```

Input Arguments

fisIn — Input fuzzy inference system

mamfis object | sugfis object | string | character vector

Input fuzzy inference system, specified as one of the following:

- mamfis or sugfis object. getFISCodeGenerationData supports fuzzy inference system objects for simulation only.

- String or character vector specifying a `.fis` file in the current working folder or on the MATLAB path. `getFISCodeGenerationData` supports fuzzy inference system file names for both simulation and code generation.

When `getFISCodeGenerationData` loads a fuzzy system that uses custom functions, it writes additional files to the current folder to support code generation for the custom functions.

Output Arguments

fisOut — Output fuzzy inference system

homogeneous structure

Output fuzzy inference system, returned as a homogeneous structure. In the homogeneous structure, if a field is a structure array, all the elements of that array are the same size. For example, in the input variable array `fisOut.input`, the:

- Names of all the variables are character vectors of the same length
- Lengths of the membership function arrays for all variables are the same

For any character vectors or structure arrays that are padded to increase their lengths, the original lengths of these elements are saved within the structure.

The `fisOut` structure is different than the structure created using `convertToStruct`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `getFISCodeGenerationData` supports fuzzy inference system objects for simulation only. To generate code for `getFISCodeGenerationData`, specify the input fuzzy inference system using a file name.

- It is recommended to not use `getFISCodeGenerationData` within a MATLAB Function block. This function is a utility function for generating code for evaluating a fuzzy inference system using MATLAB Coder.

See Also

`evalfis` | `evalfisOptions` | `mamfis` | `sugfis`

Introduced in R2018b

mam2sug

(To be removed) Transform Mamdani fuzzy inference system into Sugeno fuzzy inference system

Note mam2sug will be removed in a future release. Use `convertToSugeno` instead. For more information, see “Compatibility Considerations”.

Syntax

```
sugFIS = mam2sug(mamFIS)
```

Description

sugFIS = mam2sug(mamFIS) transforms a Mamdani fuzzy inference system into a Sugeno fuzzy inference system.

Examples

Transform Mamdani FIS into Sugeno FIS

Load a Mamdani fuzzy inference system.

```
mam_fismat = readfis('mam22.fis');
```

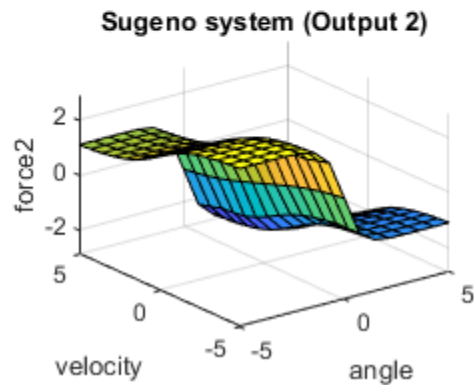
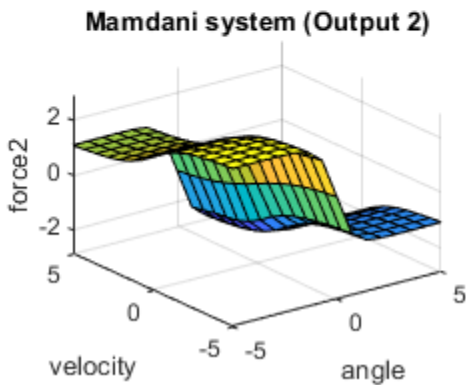
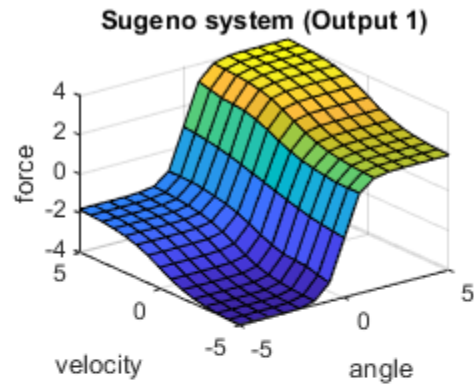
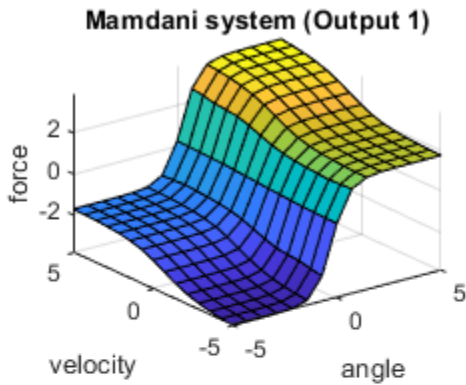
Convert this system to a Sugeno fuzzy inference system.

```
sug_fismat = mam2sug(mam_fismat);
```

Plot the output surfaces for both fuzzy systems.

```
subplot(2,2,1)
gensurf(mam_fismat)
title('Mamdani system (Output 1)')
subplot(2,2,2)
```

```
gensurf(sug_fismat)
title('Sugeno system (Output 1)')
subplot(2,2,3)
gensurf(mam_fismat,gensurfOptions('OutputIndex',2))
title('Mamdani system (Output 2)')
subplot(2,2,4)
gensurf(sug_fismat,gensurfOptions('OutputIndex',2))
title('Sugeno system (Output 2)')
```



The output surfaces for both systems are similar.

Input Arguments

mamFIS — Mamdani fuzzy inference system

structure

Mamdani fuzzy inference system, specified as a structure. Construct `mamFIS` at the command line or using the Fuzzy Logic Designer. For more information, see “Build Fuzzy Systems at the Command Line” on page 2-38 and “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14.

Output Arguments

sugFIS — Sugeno fuzzy inference system

structure

Sugeno fuzzy inference system, returned as a structure. `sugFIS`:

- Has constant output membership functions, whose values correspond to the centroids of the output membership functions in `mamFIS`
- Uses the weighted-average defuzzification method
- Uses the product implication method
- Uses the sum aggregation method

The remaining properties of `sugFIS`, including the input membership functions and rule definitions remain unchanged from `mamFIS`.

Tips

- If you have a functioning Mamdani fuzzy inference system, consider using `mam2sug` to convert to a more computationally efficient Sugeno structure to improve performance.
- If `sugFIS` has a single output variable and you have appropriate measured input/output training data, you can tune the membership function parameters of `sugFIS` using `anfis`.

Compatibility Considerations

mam2sug will be removed

Not recommended starting in R2018b

mam2sug will be removed in a future release. Use `convertToSugeno` instead. To update your code, change the function name from `mam2sug` to `convertToSugeno`. The syntaxes are equivalent.

See Also

Fuzzy Logic Designer | `convertToSugeno`

Topics

“What Is Mamdani-Type Fuzzy Inference?” on page 2-4

“What Is Sugeno-Type Fuzzy Inference?” on page 2-5

“Build Fuzzy Systems at the Command Line” on page 2-38

“Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14

Introduced before R2006a

mf2mf

(To be removed) Translate parameters between membership functions

Note mf2mf will be removed in a future release. Convert membership functions using dot notation on fsmf objects instead. For more information, see “Compatibility Considerations”.

Syntax

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

Description

This function translates any built-in membership function type into another, in terms of its parameter set. In principle, mf2mf mimics the symmetry points for both the new and old membership functions.

Note Occasionally this translation results in lost information, so that if the output parameters are translated back into the original membership function type, the transformed membership function does not look the same as it did originally.

The input arguments for mf2mf are as follows:

- `inParams` — Parameters of the membership function you are transforming from, specified as a row vector.
- `inType` — Type of membership function you are transforming from.
- `outType` — Type of membership function you are transforming to.

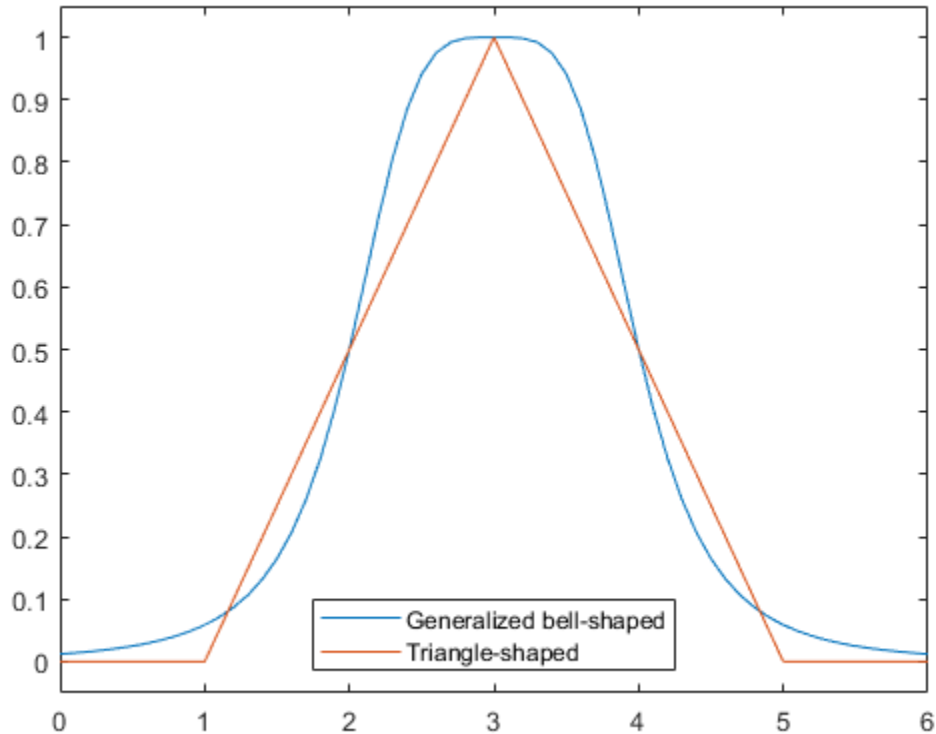
You can specify `inType` and `outType` as any of the following membership functions types:

Membership function type	Description	For more information
'gbellmf'	Generalized bell-shaped membership function	gbellmf
'gaussmf'	Gaussian membership function	gaussmf
'gauss2mf'	Gaussian combination membership function	gauss2mf
'trimf'	Triangular membership function	trimf
'trapmf'	Trapezoidal membership function	trapmf
'sigmf'	Sigmoidal membership function	sigmf
'dsigmf'	Difference between two sigmoidal membership functions	dsigmf
'psigmf'	Product of two sigmoidal membership functions	psigmf
'zmf'	Z-shaped membership function	zmf
'pimf'	Pi-shaped membership function	pimf
'smf'	S-shaped membership function	smf

Examples

Translate Parameters Between Membership Functions

```
x = 0:0.1:5;
mf1 = [1 2 3];
mf2 = mf2mf(mf1, 'gbellmf', 'trimf');
plot(x, gbellmf(x, mf1), x, trimf(x, mf2))
legend('Generalized bell-shaped', 'Triangle-shaped', 'Location', 'South')
ylim([-0.05 1.05])
```



Compatibility Considerations

mf2mf will be removed

Not recommended starting in R2018b

mf2mf will be removed in a future release. Convert membership functions using dot notation on `fismf` objects instead. There are differences between these approaches that require updates to your code.

Previously, to change the type of a membership function in a fuzzy inference system, you converted the parameters using `mf2mf`.

```
fis = readfis('tipper');  
oldType = fis.input(1).mf(1).type;  
oldParams = fis.input(1).mf(1).params;  
fis.input(1).mf(1).type = newType;  
fis.input(1).mf(1).params = mf2mf(oldParams,oldType,newType);
```

Now, when you change the type of membership function, the parameters are converted automatically.

```
fis = readfis('tipper');  
fis.Inputs(1).MembershipFunctions(1).Type = newType;
```

Previously, membership functions were represented as structures within a fuzzy inference system structure. Now, membership functions are represented as `fismf` objects within `mamfis` and `sugfis` objects. For more information on fuzzy inference system objects, see `mamfis` and `sugfis`.

See Also

`dsigmf` | `evalmf` | `gauss2mf` | `gaussmf` | `gbellmf` | `pimf` | `psigmf` | `sigmf` | `smf` | `trapmf` | `trapmf` | `trimf` | `trimf` | `zmf`

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

mfedit

Open Membership Function Editor

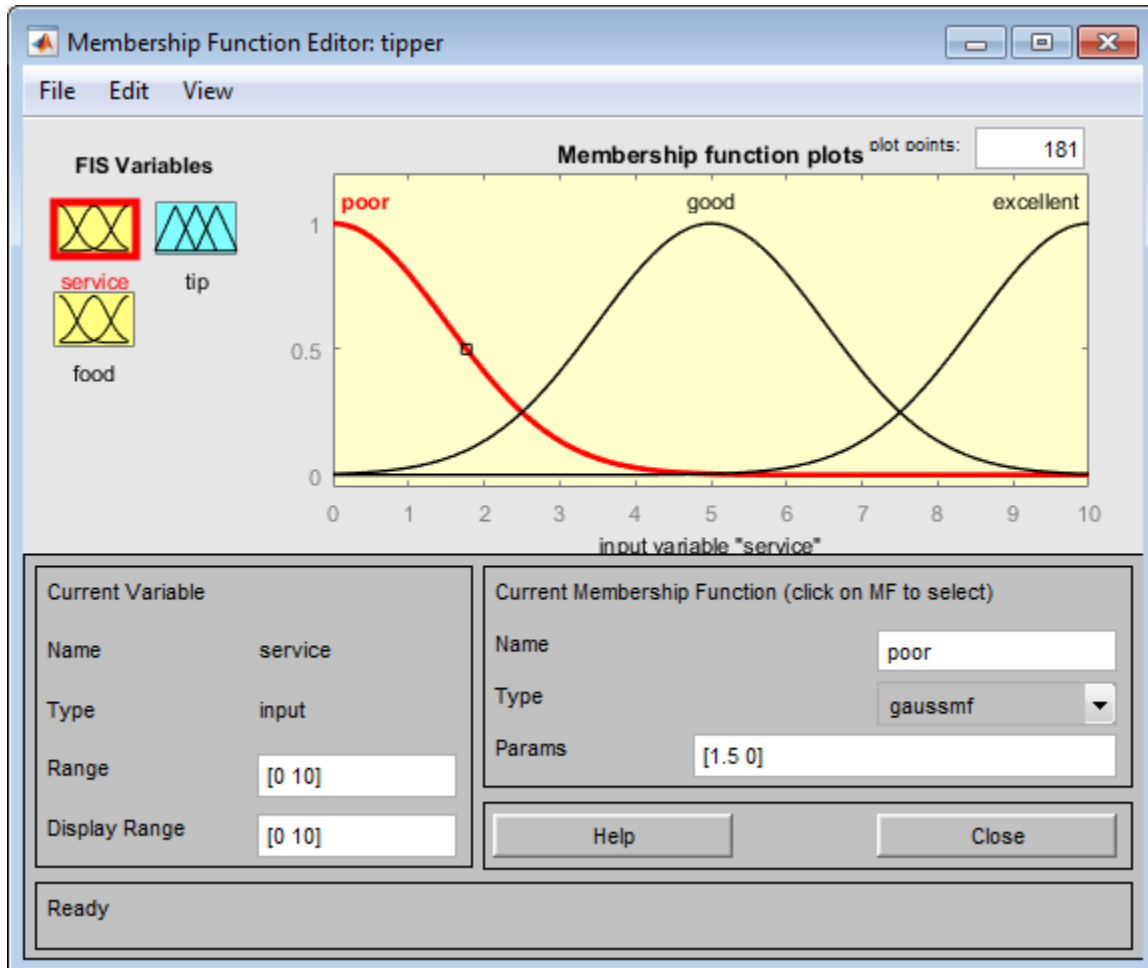
Syntax

```
mfedit(fileName)
```

```
mfedit(fis)
```

```
mfedit
```

Description



`mfedit(fileName)` generates a membership function editor that allows you to inspect and modify all the membership functions for the FIS stored in the file, `fileName`. Specify `fileName` as a character vector or string with or without the `.fis` extension.

`mfedit(fis)` operates on a `mamfis` or `sugfis` object, `fis`.

`mfedit` opens the membership function editor with no FIS loaded.

For each membership function you can change the name, the type, and the parameters. Eleven built-in membership functions are provided for you to choose from, although of course you can always create your own specialized versions. Refer to “The Membership Function Editor” on page 2-22 for more information about how to use `mfedit`.

Select the icon for the variable on the upper left side of the diagram (under **FIS Variables**) to display its associated membership functions in the plot region. Select membership functions by clicking once on them or their labels.

Menu Items

In the Membership Function Editor, there is a menu bar that allows you to open related UI tools, open and save systems, and so on. The **File** menu for the Membership Function Editor is the same as the one found in the **Fuzzy Logic Designer**.

- Under **Edit**, select:

Undo to undo the most recent change.

Add MFs to add membership functions to the current variable.

Add Custom MF to add a customized membership function to the current variable.

Remove Selected MF to delete the current membership function.

Remove All MFs to delete all membership functions of the current variable.

FIS properties to open the **Fuzzy Logic Designer**.

Rules to invoke the Rule Editor.

- Under **View**, select:

Rules to invoke the Rule Viewer.

Surface to invoke the Surface Viewer.

Membership Function Pop-up Menu

There are 11 built-in membership functions to choose from, and you also have the option of installing a customized membership function.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

Apps

Fuzzy Logic Designer

Functions

`addmf` | `plotmf` | `ruleedit` | `ruleview` | `surfview`

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

newfis

(To be removed) Create new fuzzy inference system

Note newfis will be removed in a future release. Use mamfis or sugfis instead. For more information, see “Compatibility Considerations”.

Syntax

```
fis = newfis(name)
fis = newfis(name,Name,Value)
```

Description

fis = newfis(name) returns a default Mamdani fuzzy inference system with the specified name.

fis = newfis(name,Name,Value) returns a fuzzy inference system with properties specified using one or more Name,Value pair arguments.

Examples

Create Fuzzy Inference System

Create a default Mamdani fuzzy inference system with the name, 'fis'.

```
sys = newfis('fis')

sys = struct with fields:
    name: 'fis'
    type: 'mamdani'
    andMethod: 'min'
    orMethod: 'max'
    defuzzMethod: 'centroid'
    impMethod: 'min'
```



```

aggMethod: 'max'
input: []
output: []
rule: []

```

Create Sugeno Fuzzy Inference System

Create a default Sugeno fuzzy inference system with the name, 'fis'.

```
sys = newfis('fis', 'FISType', 'sugeno')
```

```

sys = struct with fields:
    name: 'fis'
    type: 'sugeno'
    andMethod: 'prod'
    orMethod: 'probor'
    defuzzMethod: 'wtaver'
    impMethod: 'prod'
    aggMethod: 'sum'
    input: []
    output: []
    rule: []

```

Specify Implication Methods for New Fuzzy Inference System

Create a Mamdani fuzzy inference system that uses 'bisector' defuzzification and 'prod' implication.

```

sys = newfis('fis', 'DefuzzificationMethod', 'bisector', ...
    'ImplicationMethod', 'prod');

```

Input Arguments

name — Fuzzy inference system name

character vector | string

Fuzzy inference system name, specified as a character vector or string.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'OrMethod', 'probor'` configures the fuzzy OR operator as a probabilistic OR function.

FISType — Fuzzy inference system type

`'mamdani'` (default) | `'sugeno'`

Fuzzy inference system type, specified as one of the following:

- `'mamdani'` — Mamdani-type fuzzy system
- `'sugeno'` — Sugeno-type fuzzy system

For more information on the types of fuzzy inference systems, see “Types of Fuzzy Inference Systems” on page 2-2.

AndMethod — AND fuzzy operator method

`'min'` | `'prod'` | character vector | string

AND fuzzy operator method, specified as one of the following:

- `'min'` — Minimum of fuzzified input values. This method is the default when `FISType` is `'mamdani'`.
- `'prod'` — Product of fuzzified input values. This method is the default when `FISType` is `'sugeno'`.
- Character vector or string — Name of a custom AND function in the current working folder or on the MATLAB path. For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on fuzzy operators and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

OrMethod — OR fuzzy operator method

`'max'` | `'probor'` | character vector | string

OR fuzzy operator method, specified as one of the following:

- 'max' — Maximum of fuzzified input values. This method is the default when FISType is 'mamdani'.
- 'probor' — Probabilistic OR of fuzzified input values. For more information, see `probor`. This method is the default when FISType is 'sugeno'.
- Character vector or string — Name of a custom OR function in the current working folder or on the MATLAB path. For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on fuzzy operators and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

ImplicationMethod — Implication method

'min' | 'prod' | character vector | string

Implication method for computing consequent fuzzy set, specified as one of the following:

- 'min' — Truncate the consequent membership function at the antecedent result value. This method is the default when FISType is 'mamdani'.
- 'prod' — Scale the consequent membership function by the antecedent result value. This method is the default when FISType is 'sugeno'.
- Character vector or string — Name of a custom implication function in the current working folder or on the MATLAB path. For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

Note No matter what implication method you specify, Sugeno systems always use 'prod' aggregation.

For more information on implication and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

AggregationMethod — Aggregation method

'max' | 'sum' | character vector | string

Aggregation method for combining rule consequents, specified as one of the following:

- 'max' — Maximum of consequent fuzzy sets. This method is the default when FISType is 'mamdani'.
- 'sum' — Sum of consequent fuzzy sets. This method is the default when FISType is 'sugeno'.

- 'probor' — Probabilistic OR of consequent fuzzy sets. For more information, see `probor`.
- Character vector or string — Name of a custom aggregation function in the current working folder or on the MATLAB path. For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

Note No matter what aggregation method you specify, Sugeno systems always use 'sum' aggregation.

For more information on aggregation and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

DefuzzificationMethod — Defuzzification method

'centroid' | 'bisector' | 'mom' | 'lom' | 'som' | 'wtaver' | 'wtsum' | character vector | string

Defuzzification method for computing crisp output values.

If `FISType` is 'mamdani', specify the defuzzification method as one of the following:

- 'centroid' — Centroid of the area under the output fuzzy set. This method is the default for Mamdani systems.
- 'bisector' — Bisector of the area under the output fuzzy set
- 'mom' — Mean of the values for which the output fuzzy set is maximum
- 'lom' — Largest value for which the output fuzzy set is maximum
- 'som' — Smallest value for which the output fuzzy set is maximum

For more information on defuzzification methods for Mamdani systems, see “Defuzzification Methods”.

If `FISType` is 'sugeno', specify the defuzzification method as one of the following:

- 'wtaver' — Weighted average of all rule outputs. This method is the default for Sugeno systems.
- 'wtsum' — Weighted sum of all rule outputs

You can also specify the defuzzification method using a character vector or string that contains the name of a custom function in the current working folder or on the MATLAB

path. For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on defuzzification and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

Output Arguments

fis — Fuzzy inference system

FIS structure

Fuzzy inference system with the specified name, returned as an FIS structure. The fuzzy system is configured using the specified Name, Value pair arguments.

fis has no input variables, output variables, or rules. To add variables or rules to fis, use addvar or addrule. You can also edit the fuzzy system using **Fuzzy Logic Designer**.

Compatibility Considerations

newfis will be removed

Not recommended starting in R2018b

newfis will be removed in a future release. Use mamfis or sugfis instead. There are differences between these functions that require updates to your code.

To create a Mamdani or Sugeno FIS, use mamfis or sugfis, respectively.

This table shows some typical usages of newfis for creating fuzzy systems and how to update your code to use mamfis or sugfis instead.

If your code has this form:	Use this code instead:
<code>fis = newfis(name)</code>	<code>fis = mamfis('Name',name)</code>
<code>fis = newfis(name,'FISType','mamdani')</code>	<code>fis = mamfis('Name',name)</code>
<code>fis = newfis(name,'FISType','sugeno')</code>	<code>fis = sugfis('Name',name)</code>

If your code has this form:	Use this code instead:
<pre>fis = newfis(name,... 'FISType','mamdani',... 'AndMethod','prod')</pre>	<pre>fis = mamfis('Name',name,... 'AndMethod','prod')</pre>
<pre>fis = newfis(name,... 'FISType','sugeno',... 'OrMethod','probor')</pre>	<pre>fis = sugfis('Name',name,... 'OrMethod','probor')</pre>

See Also

[mamfis](#) | [readfis](#) | [sugfis](#) | [writefis](#)

Topics

“Foundations of Fuzzy Logic” on page 1-10

“Fuzzy Inference Process” on page 1-28

Introduced before R2006a

parsrule

(To be removed) Parse fuzzy rules

Note `parsrule` will be removed in a future release. Use `addRule` instead. For more information, see “Compatibility Considerations”.

Syntax

```
outFIS = parsrule(inFIS,ruleList)
outFIS = parsrule(inFIS,ruleList,Name,Value)
```

Description

`outFIS = parsrule(inFIS,ruleList)` returns a fuzzy inference system, `outFIS`, that is equivalent to the input fuzzy system, `inFIS`. but with fuzzy rules replaced by the rules specified in `ruleList`.

`outFIS = parsrule(inFIS,ruleList,Name,Value)` parses the rules in `ruleList` using options specified by one or more `Name,Value` pair arguments.

Examples

Add Rules to Fuzzy Inference System

Load a fuzzy inference system (FIS).

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

Specify if-then rules using the default 'verbose' format.

```
rule1 = "If service is poor or food is rancid then tip is cheap";
rule2 = "If service is excellent and food is not rancid then tip is generous";
rules = [rule1 rule2];
```

Add the rules to the FIS.

```
fis2 = parsrule(fis,rules);
```

fis2 is equivalent to fis, except that the rule base is replaced with the specified rules.

Add Rules Using Symbolic Expressions

Load a fuzzy inference system (FIS).

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

Specify the following rules using symbols:

- If service is poor or food is rancid then tip is cheap.
- If service is excellent and food is not rancid then tip is generous.

```
rule1 = "service==poor | food==rancid => tip=cheap";  
rule2 = "service==excellent & food~=rancid => tip=generous";  
rules = [rule1 rule2];
```

Add the rules to the FIS using the 'symbolic' format.

```
fis2 = parsrule(fis,rules,'Format','symbolic');
```

Add Rules Using Membership Function Indices

Load fuzzy inference system (FIS).

```
fis = readfis('mam22.fis');
```

Specify the following rules using membership function indices:

- If angle is small and velocity is big, then force is negBig and force2 is posBig2.
- If angle is not small and velocity is small, then force is posSmall and force2 is negSmall2.

```
rule1 = "1 2, 1 4 (1) : 1";  
rule2 = "-1 1, 3 2 (1) : 1";  
rules = [rule1 rule2];
```

Add rules to FIS using the 'indexed' format.


```
fis2 = parsrule(fis,rules,'Format','indexed');
```

Add Rules Using French Language

Load a fuzzy inference system (FIS).

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

Specify if-then rules using French keywords.

```
rule1 = "Si service est poor ou food est rancid alors tip est cheap";  
rule2 = "Si service est excellent et food n'est_pas rancid alors tip est generous";  
rules = [rule1 rule2];
```

Add the rules to the FIS.

```
fis2 = parsrule(fis,rules,'Language','francais');
```

Add Single Rule to Fuzzy Inference System

Load a fuzzy inference system (FIS).

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

Add a rule to the FIS.

```
ruleTxt = 'If service is poor then tip is cheap';  
a2 = parsrule(a,ruleTxt,'verbose');
```

Input Arguments

inFIS — Fuzzy inference system

FIS structure

Input fuzzy inference system, specified as an FIS structure. `parsrule` does not modify `inFIS`.

ruleList — Fuzzy rules

character array | string array | character vector | string

Fuzzy rules, specified as one of the following:

- Character array where each row corresponds to a rule. For example:

```
rule1 = 'If service is poor or food is rancid then tip is cheap';  
rule2 = 'If service is good then tip is average';  
rule3 = 'If service is excellent or food is delicious then tip is generous';  
ruleList = char(rule1,rule2,rule3);
```

- String array, where each element corresponds to a rule. For example:

```
ruleList = ["If service is poor or food is rancid then tip is cheap";  
           "If service is good then tip is average";  
           "If service is excellent or food is delicious then tip is generous"];
```

- Character vector or string to specify a single rule.

You can change the rule format and language using the `Format` and `Language` options.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Format', 'symbolic'` sets the rule format to symbolic expressions.

Format — Rule format

`'verbose'` (default) | `'symbolic'` | `'indexed'`

Rule format, specified as the comma-separated pair consisting `'Format'` and one of the following:

- `'verbose'` — Use linguistic expressions.

```
'If service is poor or food is rancid then tip is cheap 1'
```

Specify the rule weight at the end of the rule text. If you omit the weight, a default value of 1 is used.

You can specify the rule language using the `Language` option.

- `'symbolic'` — Use language-neutral symbolic expressions.

```
'service==poor | food==rancid => tip=cheap 1'
```

Specify symbolic expressions using the following symbols.

Rule Component	Symbol
AND	&
OR	
IS (in antecedent)	==
IS (in consequent)	=
IS NOT	~=
Implication (then)	=>

Specify the rule weight at the end of the rule text. If you omit the weight, a default value of 1 is used.

- 'indexed' — Use input and output membership function (MF) indices.

Specify indexed rules in the following format:

```
'<input MFs>, <output MFs>, (<weight>) : <logical operator - 1(AND), 2(OR)>'
```

For example:

```
'1 1, 1 (1) : 2'
```

To indicate NOT operations for input and output membership functions, use negative indices. For example, to specify “not the second membership function,” use -2.

To indicate a don't care condition for an input or output membership function, use 0.

Language — Rule language

```
'english' (default) | 'français' | 'deutsch'
```

Rule language for 'verbose' format, specified as one of the following:

- 'english' — Specify rules in English.
'If service is poor or food is rancid then tip is cheap'
- 'français' — Specify rules in French.
'Si service est poor ou food est rancid alors tip est cheap'
- 'deutsch' — Specify rules in German.
'Wenn service ist poor oder food ist rancid dann tip ist cheap'

The software parses the rules in ruleList using the following keywords.

Rule Component	English	French	German
Start of antecedent	if	si	wenn
AND	and	et	und
OR	or	ou	oder
Start of consequent (implication)	then	alors	dann
IS	is	est	ist
IS NOT	is not	n'est_pas	ist nicht

Output Arguments

outFIS — Output fuzzy inference system

FIS structure

Fuzzy inference system, returned as an FIS structure. `outFIS` is the same as `inFIS`, except that the rule list contains only the rules specified in `ruleList`.

Compatibility Considerations

parsrule will be removed

Not recommended starting in R2018b

`parsrule` will be removed in a future release. Use `addRule` instead.

If you previously added rules using linguistic or symbolic expressions with `parsrule`, you can specify rules using the same expressions with `addrule`. `addRule` automatically detects the format of the strings or character vectors in your rule list. Therefore, it is no longer necessary to specify the rule format. To add a rule list using `addRule`, use the following command:

If you previously added rules using indexed expressions with `parsrule`, update your code to use arrays of indices instead. For example, if your code has the following form:

```
rule1 = "1 2, 1 4 (1) : 1";  
rule2 = "-1 1, 3 2 (1) : 1";
```

```
rules = [rule1 rule2];  
fis = parsrule(fis,rules,'Format','indexed');
```

use this code instead:

```
rule1 = [1 2 1 4 1 1];  
rule2 = [-1 1 3 2 1 1];  
rules = [rule1; rule2];  
fis = addRule(fis,rules);
```

If you previously specified rules using the 'Language' name-value pair argument with `parsrule`, this functionality has been removed and there is no replacement. Specify your rules using `addRule` a different rule format.

Previously, `parsrule` replaced the entire rule list in your fuzzy system. `addRule` appends your specified rules to the rule list.

See Also

`addrule` | `ruleedit` | `showrule`

Introduced before R2006a

pimf

Π -shaped membership function

Syntax

$y = \text{pimf}(x, [a \ b \ c \ d])$

Description

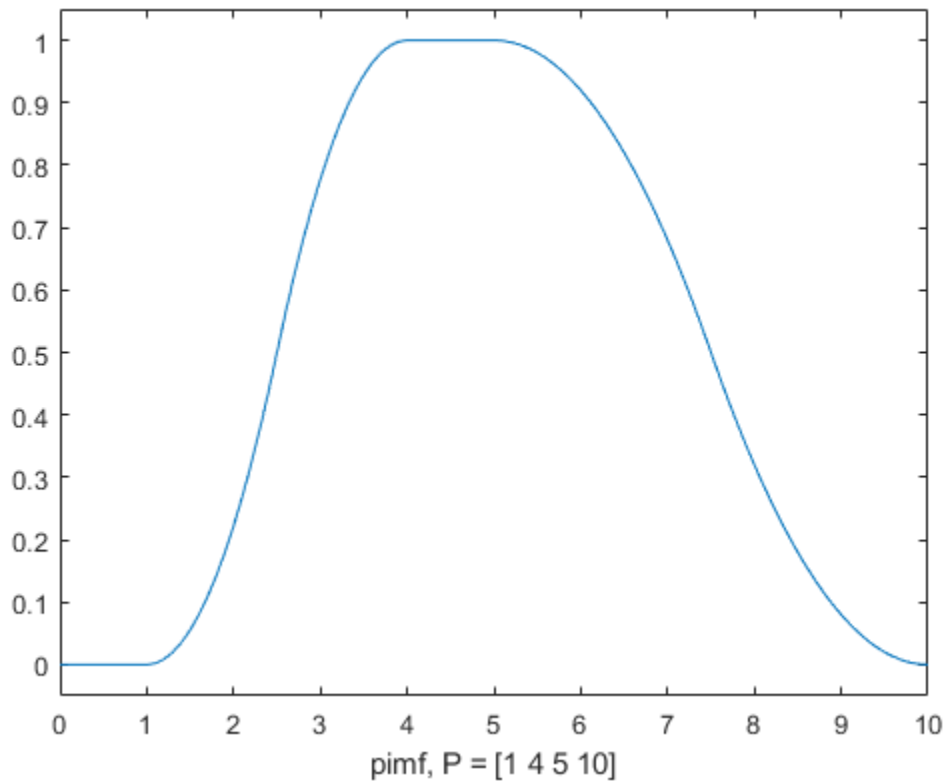
This spline-based curve is so named because of its Π shape. The membership function is evaluated at the points determined by the vector x . The parameters a and d locate the "feet" of the curve, while b and c locate its "shoulders." The membership function is a product of smf and zmf membership functions, and is given by:

$$f(x; a, b, c, d) = \begin{cases} 0, & x \leq a \\ 2 \left(\frac{x-a}{b-a} \right)^2, & a \leq x \leq \frac{a+b}{2} \\ 1 - 2 \left(\frac{x-b}{b-a} \right)^2, & \frac{a+b}{2} \leq x \leq b \\ 1, & b \leq x \leq c \\ 1 - 2 \left(\frac{x-c}{d-c} \right)^2, & c \leq x \leq \frac{c+d}{2} \\ 2 \left(\frac{x-d}{d-c} \right)^2, & \frac{c+d}{2} \leq x \leq d \\ 0, & x \geq d \end{cases}$$

Examples

Pi-Shaped Membership Function

```
x = 0:0.1:10;  
y = pimf(x,[1 4 5 10]);  
plot(x,y)  
xlabel('pimf, P = [1 4 5 10]')  
ylim([-0.05 1.05])
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dsigmf` | `evalmf` | `gauss2mf` | `gaussmf` | `gbellmf` | `mf2mf` | `psigmf` | `sigmf` | `smf` | `trapmf` | `trapmf` | `trimf` | `trimf` | `zmf`

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

plotfis

Display fuzzy inference system

Syntax

```
plotfis(fis)
```

Description

`plotfis(fis)` displays a high-level diagram of a fuzzy inference system (FIS). The center of the display shows the name, type, and rule count for the FIS. The input variables with associated membership functions are displayed to the right, and the outputs with their associated membership functions are displayed on the left.

Examples

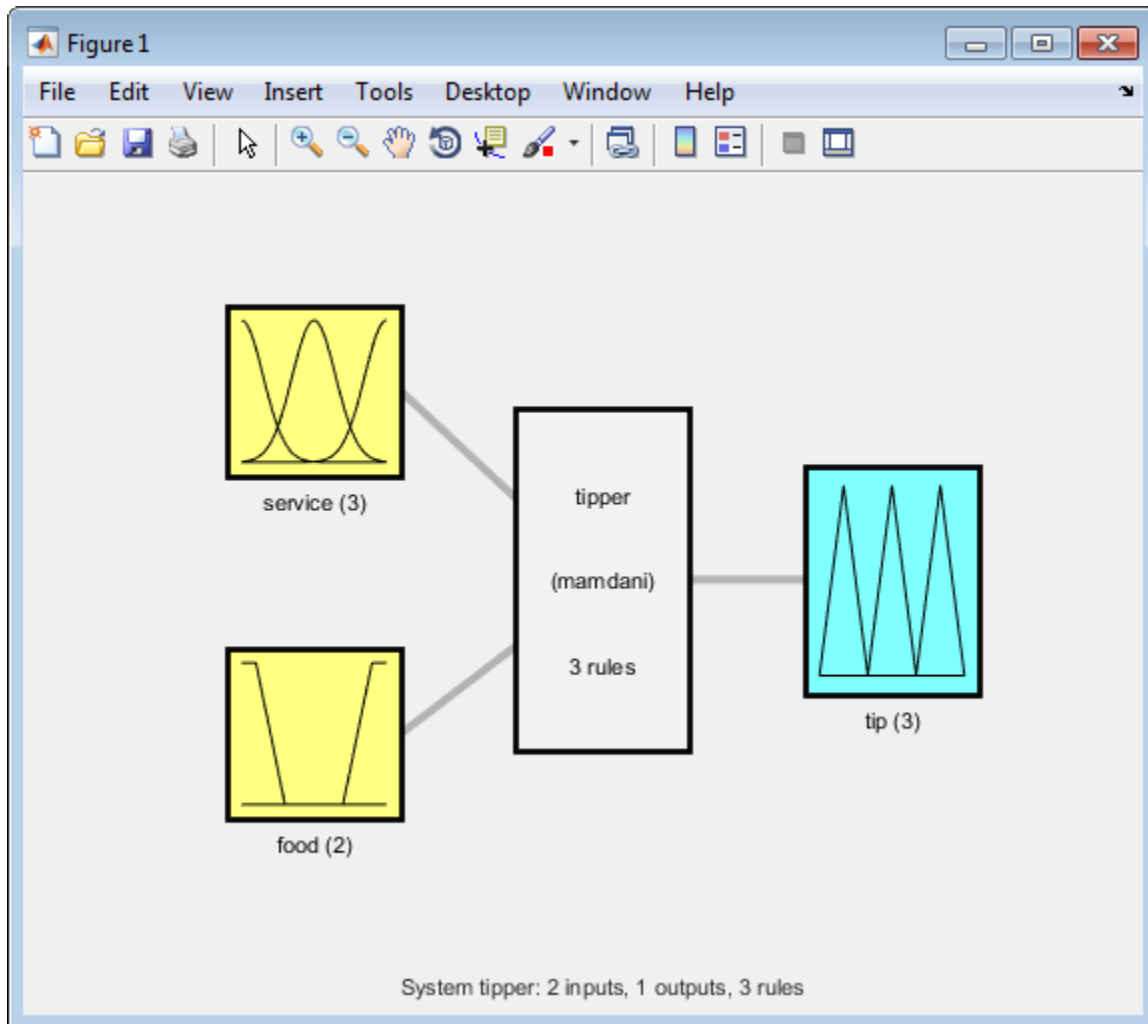
Display Fuzzy Inference System

Create a fuzzy inference system (FIS). For this example, read the FIS from the `tipper.fis` file.

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

Display the fuzzy system

```
plotfis(fis)
```



Input Arguments

fis — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, specified as one of the following:

- `mamfis` object — Mamdani fuzzy inference system
- `sugfis` object — Sugeno fuzzy inference system

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

`evalmf` | `mamfis` | `plotmf` | `readfis` | `sugfis`

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced before R2006a

plotmf

Plot membership functions for input or output variable

Syntax

```
plotmf(fis,variableType,variableIndex)  
plotmf( ____,numPoints)
```

```
[x0ut,y0ut] = plotmf( ____)
```

Description

`plotmf(fis,variableType,variableIndex)` plots the membership functions for an input or output variable in the fuzzy inference system `fis`.

`plotmf(____,numPoints)` specifies the number of data points to plot for each membership function.

`[x0ut,y0ut] = plotmf(____)` returns the x-axis and y-axis data for the membership functions without plotting them.

Examples

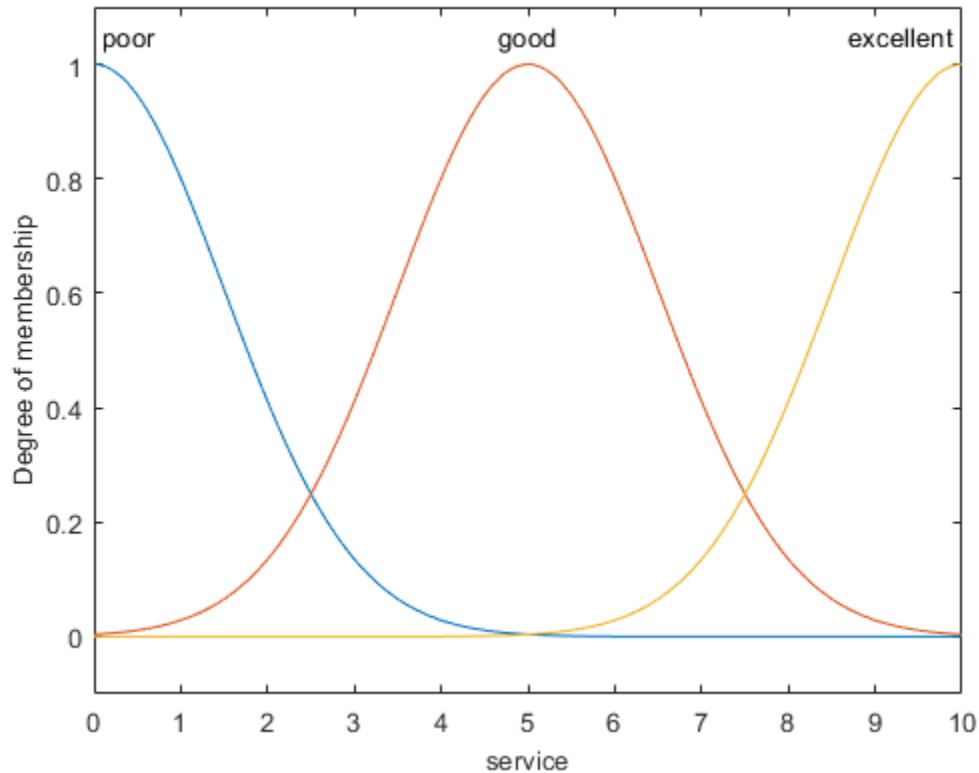
Plot Membership Functions for Input Variable

Create a fuzzy inference system.

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

Plot the membership functions for the first input variable.

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



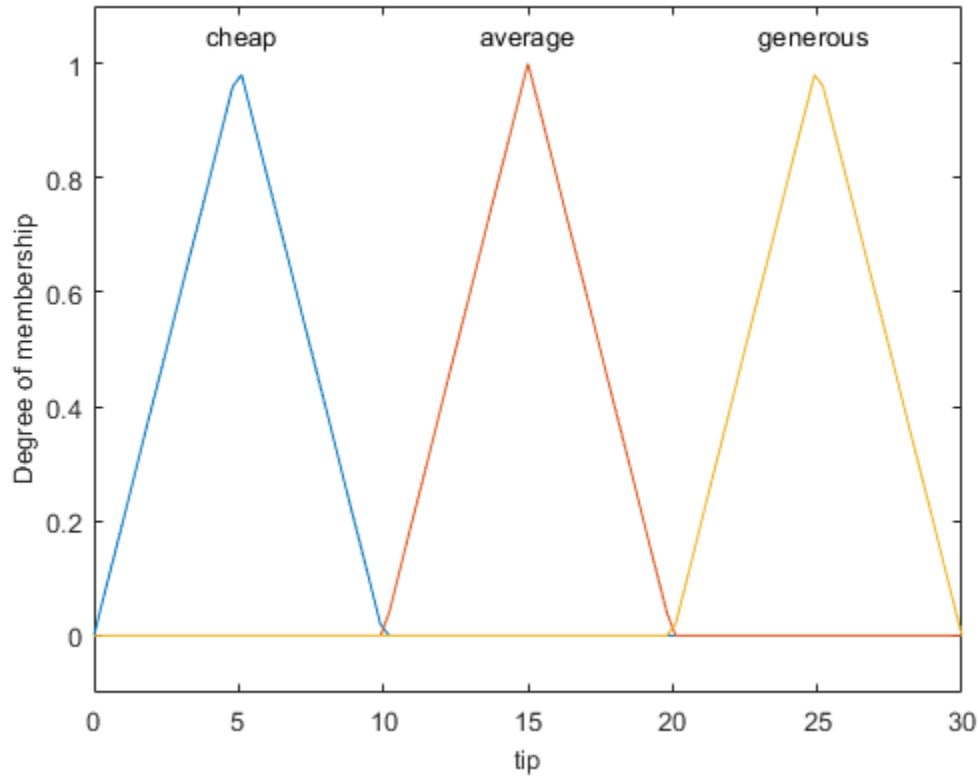
Specify Number of Points for Membership Function Plot

Create a fuzzy inference system.

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

Plot the membership functions for the first output variable using 101 data points for each membership function.

```
plotmf(fis, 'output', 1, 101)
```



Obtain Membership Function Plot Data

Create a fuzzy inference system.

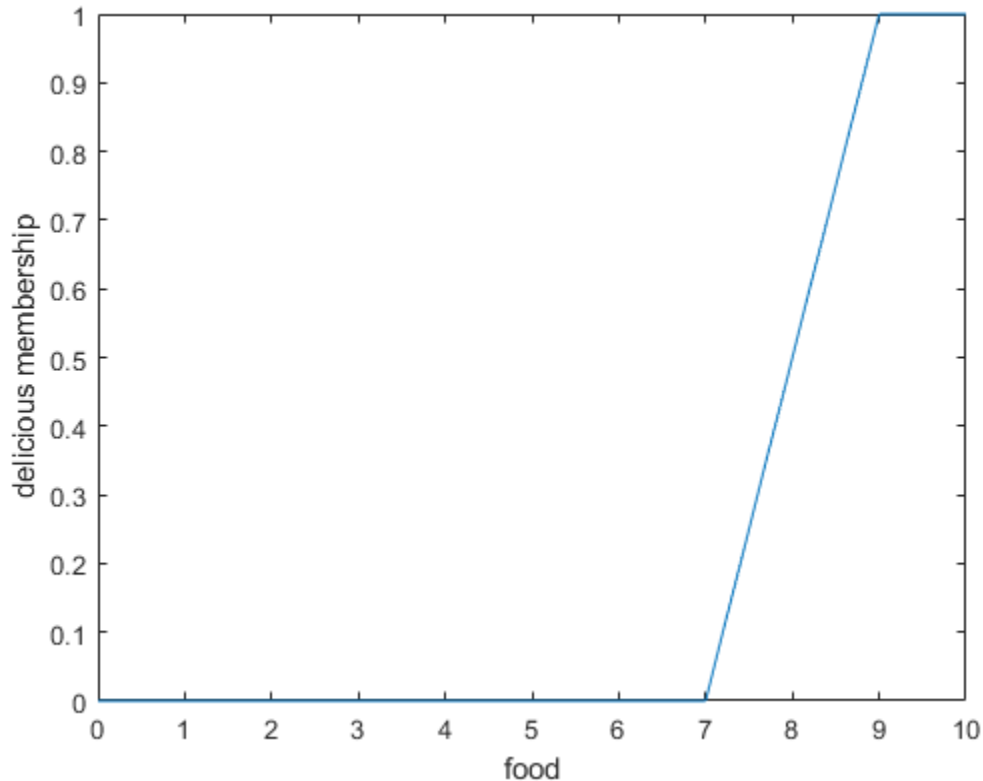
```
fis = readfis('tipper');
```

Obtain the x-axis and y-axis data for the membership functions of the second input variable.

```
[xOut,yOut] = plotmf(fis,'input',2);
```

You can then, for example, plot a single membership function using this data.

```
plot(xOut(:,2),yOut(:,2))  
xlabel('food')  
ylabel('delicious membership')
```



Input Arguments

fis — Fuzzy inference system

mamfis object | sugfis object

Fuzzy inference system, specified as a mamfis or sugfis object. Construct **fis** at the command line or using the **Fuzzy Logic Designer**. For more information, see “Build

Fuzzy Systems at the Command Line” on page 2-38 and “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14.

`plotmf` does not support plotting output membership functions of Sugeno systems.

variableType — Variable type

'input' | 'output'

Variable type, specified as one of the following:

- 'input' — Input variable
- 'output' — Output variable

variableIndex — Variable index

positive integer

Variable index, specified as a positive integer. If `variableType` is:

- 'input', then `variableIndex` must be less than or equal to the number of input variables in `fis`
- 'output', then `variableIndex` must be less than or equal to the number of output variables in `fis`

numPoints — Number of data points to plot

181 (default) | positive integer

Number of data points to plot, specified as a positive integer.

Output Arguments

xOut — Plot x-axis data

array

Plot x-axis data, returned as a `numPoints`-by- N_{MF} array, where N_{MF} is the number of membership functions for the variable specified by `variableType` and `variableIndex`.

yOut — Plot y-axis data

array

Plot y-axis data, returned as a `numPoints`-by- N_{MF} array, where N_{MF} is the number of membership functions for the variable specified by `variableType` and `variableIndex`.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

Functions

`evalmf` | `plotfis`

Introduced before R2006a

probor

Probabilistic OR

Syntax

```
y = probor(x)
```

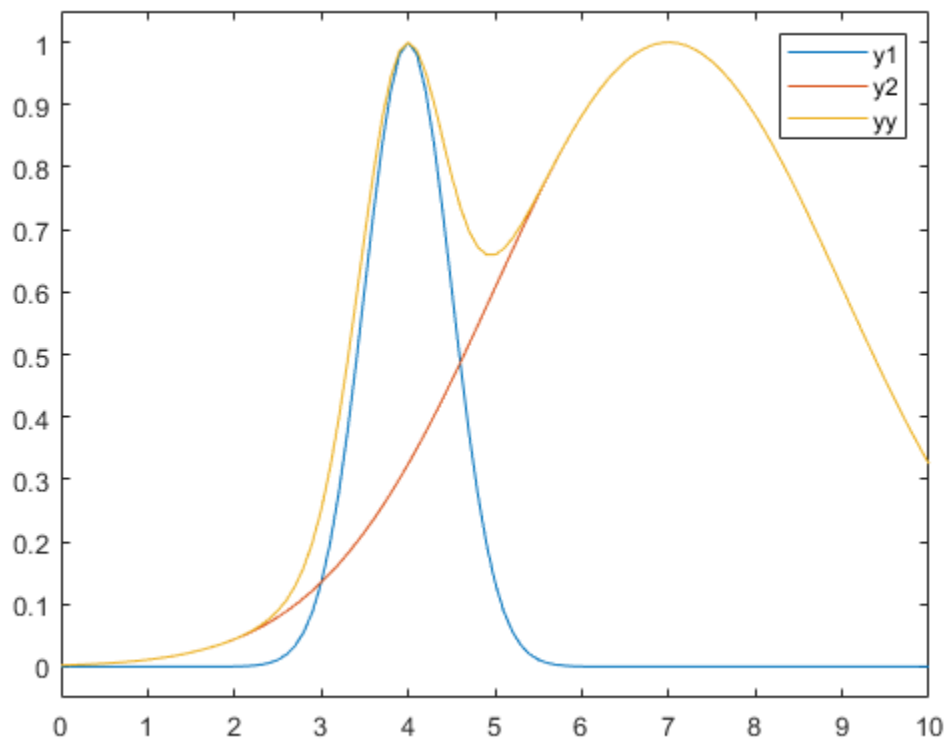
Description

`y = probor(x)` returns the probabilistic OR (also known as the *algebraic sum*) of the columns of `x`. If `x` has two rows such that `x = [a; b]`, then `y = a + b - ab`. If `x` has only one row, then `y = x`.

Examples

Probabilistic OR

```
x = 0:0.1:10;  
y1 = gaussmf(x, [0.5 4]);  
y2 = gaussmf(x, [2 7]);  
yy = probor([y1;y2]);  
plot(x, [y1;y2;yy])  
legend('y1', 'y2', 'yy')  
ylim([-0.05 1.05])
```



See Also

Introduced before R2006a

psigmf

Product of two sigmoidal membership functions

Syntax

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

Description

The sigmoid curve plotted for the vector x depends on two parameters a and c as given by

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

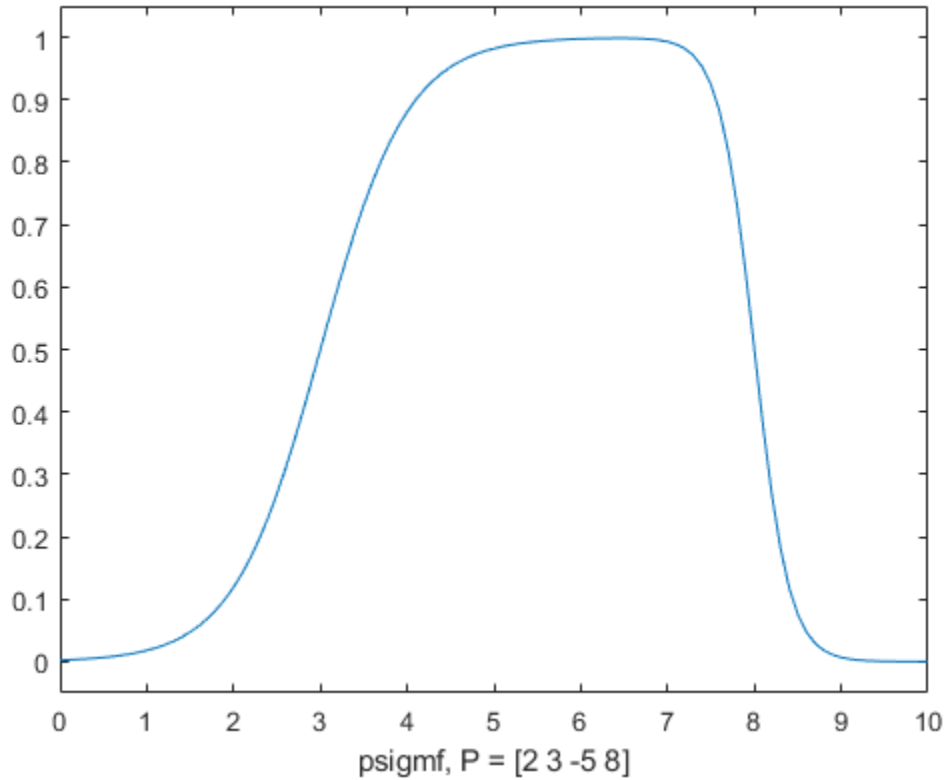
`psigmf` is simply the product of two such curves plotted for the values of the vector x
 $f_1(x; a_1, c_1) \times f_2(x; a_2, c_2)$

The parameters are listed in the order $[a_1 c_1 a_2 c_2]$.

Examples

Product of Two Sigmoidal Membership Functions

```
x = 0:0.1:10;  
y = psigmf(x,[2 3 -5 8]);  
plot(x,y)  
xlabel('psigmf, P = [2 3 -5 8]')  
ylim([-0.05 1.05])
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dsigmf` | `evalmf` | `gauss2mf` | `gaussmf` | `gbellmf` | `mf2mf` | `pimf` | `sigmf` | `smf` | `trapmf` | `trapmf` | `trimf` | `trimf` | `zmf`

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

readfis

Load fuzzy inference system from file

You can load a fuzzy inference system (FIS) from a `.fis` file using the `readfis` function. To save a FIS to a file, use the `writeFIS` function.

Do not manually edit the contents of a `.fis` file. Doing so can produce unexpected results when loading the file using `readfis`.

Syntax

```
fis = readfis(fileName)
fis = readfis
```

Description

`fis = readfis(fileName)` reads a FIS from the file specified by `fileName`.

`fis = readfis` opens a dialog box for selecting and opening loading a `.fis` file.

Examples

Load Fuzzy Inference System from File

Load the fuzzy system stored in the file `tipper.fis`.

```
fis = readfis('tipper')

fis =
    mamfis with properties:
        Name: "tipper"
        AndMethod: "min"
        OrMethod: "max"
        ImplicationMethod: "min"
```

```
AggregationMethod: "max"  
DefuzzificationMethod: "centroid"  
    Inputs: [1x2 fisvar]  
    Outputs: [1x1 fisvar]  
    Rules: [1x3 fisrule]  
DisableStructuralChecks: 0
```

Input Arguments

fileName — File name

string | character vector

File name specified as a string or character vector with or without the `.fis` extension. This file must be in the current working directory or on the MATLAB path.

Output Arguments

fis — Fuzzy inference system

mamfis object | sugfis object

Fuzzy inference system, returned as one of the following:

- mamfis object — Mamdani fuzzy inference system
- sugfis object — Sugeno fuzzy inference system

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

`writeFIS`

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced before R2006a

removeInput

Remove input variable from fuzzy inference system

Syntax

```
fisOut = removeInput(fisIn,inputName)
```

Description

`fisOut = removeInput(fisIn,inputName)` removes the input variable with the name `inputName` from fuzzy inference system `fisIn`, and returns the resulting fuzzy system in `fisOut`.

Examples

Remove Input Variable from Fuzzy Inference System

Load fuzzy system.

```
fis = readfis("tipper");
```

View the input variables of `fis`.

```
fis.Inputs
```

```
ans =  
  1x2 fisvar array with properties:
```

```
    Name  
    Range  
    MembershipFunctions
```

```
Details:
```

Name	Range	MembershipFunctions
------	-------	---------------------

```

1   "service"    0   10   [1x3 fismf]
2   "food"      0   10   [1x2 fismf]

```

View the rules of `fis`.

`fis.Rules`

```

ans =
    1x3 fisrule array with properties:

```

```

    Description
    Antecedent
    Consequent
    Weight
    Connection

```

Details:

Description

```

1   "service==poor | food==rancid => tip=cheap (1)"
2   "service==good => tip=average (1)"
3   "service==excellent | food==delicious => tip=generous (1)"

```

Remove the `service` input variable.

```

fis = removeInput(fis,"service");

```

View the updated input variables.

`fis.Inputs`

```

ans =
    fisvar with properties:
        Name: "food"
        Range: [0 10]
        MembershipFunctions: [1x2 fismf]

```

View the updated rules.

`fis.Rules`

```
ans =  
  1x2 fisrule array with properties:  
  
    Description  
    Antecedent  
    Consequent  
    Weight  
    Connection  
  
Details:  
          Description  
-----  
  1    "food==rancid => tip=cheap (1)"  
  2    "food==delicious => tip=generous (1)"
```

service has been removed from the variables and rules of `fis`.

Input Arguments

fisIn — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, specified as a `mamfis` or `sugfis` object.

inputName — Input variable name

string | character vector

Input variable name, specified as a string or character vector.

Output Arguments

fisOut — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, returned as a `mamfis` or `sugfis` object. `fisOut` has the same properties as `fisIn` except:

- The input variable with the specified name is removed.
- The specified input variable is removed from any fuzzy rules. If a rule has only the specified input variable in its antecedent, then the entire rule is removed. If a rule has more than one input variable in its antecedent, then the specified input variable is removed from the antecedent.

See Also

addInput | fisvar | mamfis | sugfis

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

removeMF

Remove membership function from fuzzy variable

Syntax

```
fisOut = removeMF(fisIn,varName,mfName)
fisOut = removeMF(fisIn,varName,mfName,'VariableType',varType)

varOut = removeMF(varIn,varName,mfName)
```

Description

`fisOut = removeMF(fisIn,varName,mfName)` removes the membership function `mfName` from the input or output variable `varName` in the fuzzy inference system `fisIn` and returns the resulting fuzzy system in `fisOut`. To use this syntax, `varName` must be a unique variable name within `fisIn`.

`fisOut = removeMF(fisIn,varName,mfName,'VariableType',varType)` removes the membership function from either an input or output variable as specified by `varType`. Use this syntax when your FIS has an input variable with the same name as an output variable.

`varOut = removeMF(varIn,varName,mfName)` removes the membership function `mfName` from the fuzzy variable `varIn` and returns the resulting fuzzy variable in `varOut`.

Examples

Remove Membership Function from Fuzzy Inference System

Create a Mamdani fuzzy inference system with two inputs and one output. By default, when you specify the number of inputs and outputs, `mamfis` adds three membership functions to each variable.

```
fis = mamfis('NumInputs',3,'NumOutputs',1)

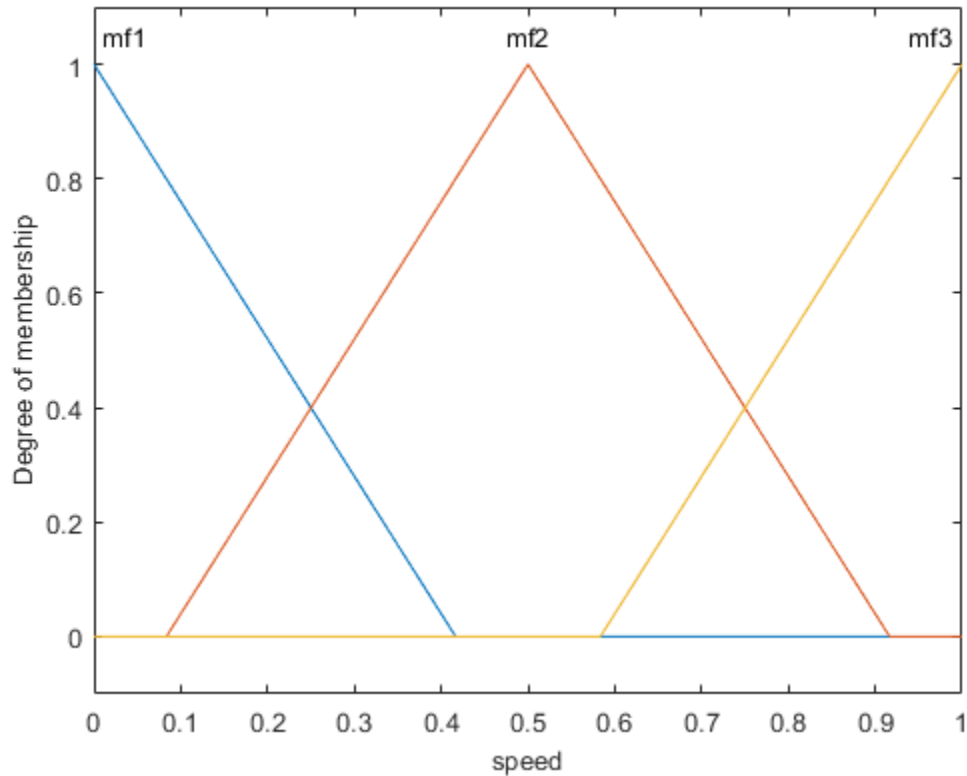
fis =
    mamfis with properties:
        Name: "fis"
        AndMethod: "min"
        OrMethod: "max"
        ImplicationMethod: "min"
        AggregationMethod: "max"
        DefuzzificationMethod: "centroid"
        Inputs: [1x3 fisvar]
        Outputs: [1x1 fisvar]
        Rules: [1x27 fisrule]
        DisableStructuralChecks: 0
```

Name the variables. For this example, give the second input variable and the output variable the same name.

```
fis.Inputs(1).Name = "speed";
fis.Inputs(2).Name = "throttle";
fis.Inputs(3).Name = "distance";
fis.Outputs(1).Name = "throttle";
```

View the membership functions for the first input variable.

```
plotmf(fis, "input", 1)
```

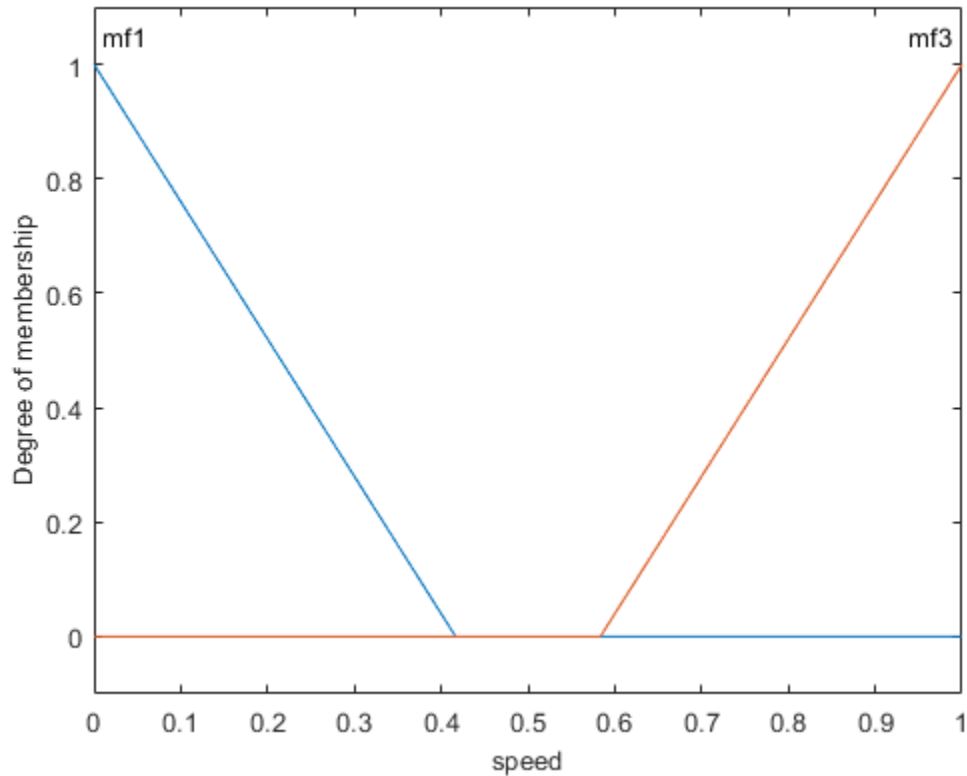


Remove the second membership function, mf2, from the first input variable.

```
fis = removeMF(fis, "speed", "mf2");
```

View the membership functions again. The specified membership function has been removed.

```
plotmf(fis, "input", 1)
```

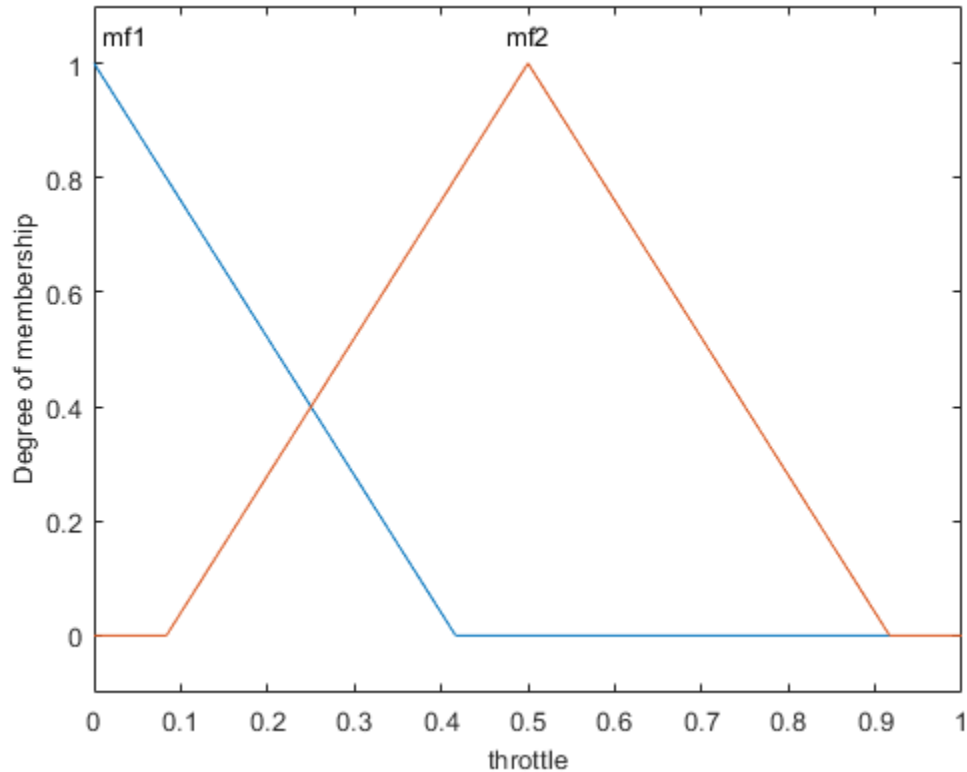



If your system has an input variable with the same name as an output variable, you must specify the variable type when removing a membership function. For example, remove the mf3 membership function from the output variable.

```
fis = removeMF(fis,"throttle","mf3",'VariableType',"output");
```

View the membership functions of the output variable.

```
plotmf(fis,"output",1)
```



Remove Membership Function from Fuzzy Variable

Create a fuzzy variable with a specified range and add three membership functions

```
var = fisvar([0 10]);  
var = addMF(var, "trimf", [0 2.5 5], "Name", "small");  
var = addMF(var, "trimf", [2.5 5 7.5], "Name", "medium");  
var = addMF(var, "trimf", [5 7.5 10], "Name", "large");
```

View the membership functions.

```
var.MembershipFunctions
```

```
ans =
  1x3 fismf array with properties:

    Name
    Type
    Parameters

Details:
      Name      Type      Parameters
  _____  _____  _____
  1  "small"    "trimf"    0   2.5   5
  2  "medium"   "trimf"    2.5 5   7.5
  3  "large"    "trimf"    5   7.5  10
```

Remove the `medium` membership function from the variable.

```
var = removeMF(var,"medium");
```

Verify that the membership was removed.

```
var.MembershipFunctions
```

```
ans =
  1x2 fismf array with properties:

    Name
    Type
    Parameters

Details:
      Name      Type      Parameters
  _____  _____  _____
  1  "small"    "trimf"    0   2.5   5
  2  "large"    "trimf"    5   7.5  10
```

Input Arguments

fisIn — Input fuzzy inference system

mamfis object | sugfis object

Fuzzy inference system, specified as a `mamfis` or `sugfis` object.

varName — Variable name

string | character vector

Variable name, specified as a string or character vector. You can specify the name of either an input or output variable in your FIS.

mfName — Membership function name

string | character vector

Membership function name, specified as a string or character vector.

varType — Variable type

string | character vector

Variable type, specified as one of the following:

- "input" — Input variable
- "output" — Output variable

If your system has an input variable with the same name as an output variable, specify which variable to remove the membership function from using `varType`.

varIn — Fuzzy variable

`fisvar` object

Fuzzy variable, specified as a `fisvar` object.

Output Arguments

fisOut — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, returned as a `mamfis` or `sugfis` object. `fisOut` has the same properties as `fisIn` except:

- The membership function with the specified name is removed from the specified variable.
- The specified membership function is removed from any fuzzy rules. If a rule has only the specified membership function in its antecedent, then the entire rule is removed. If

a rule has more than one membership function in its antecedent, then the specified membership function is removed from the antecedent.

varOut — Fuzzy variable

fisvar object

Fuzzy variable, returned as a `fisvar` object. `varOut` has the same properties as `varIn` except the membership function with the specified name is removed.

See Also

`addMF` | `fisvar` | `mamfis` | `sugfis`

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

removeOutput

Remove output variable from fuzzy inference system

Syntax

```
fisOut = removeOutput(fisIn,outputName)
```

Description

`fisOut = removeOutput(fisIn,outputName)` removes the output variable with the name `outputName` from fuzzy inference system `fisIn`, and returns the resulting fuzzy system in `fisOut`.

Examples

Remove Output Variable from Fuzzy Inference System

Load fuzzy system.

```
fis = readfis("mam22");
```

View the output variables of `fis`.

```
fis.Outputs
```

```
ans =  
  1x2 fisvar array with properties:
```

```
    Name  
    Range  
    MembershipFunctions
```

```
Details:
```

Name	Range	MembershipFunctions
------	-------	---------------------

```

1    "force"    -5    5    [1x4 fismf]
2    "force2"  -5    5    [1x4 fismf]

```

View the rules of `fis`.

`fis.Rules`

```

ans =
  1x4 fisrule array with properties:

```

```

  Description
  Antecedent
  Consequent
  Weight
  Connection

```

Details:

Description

```

1    "angle==small & velocity==small => force=negBig, force2=posBig2 (1)"
2    "angle==small & velocity==big => force=negSmall, force2=posSmall2 (1)"
3    "angle==big & velocity==small => force=posSmall, force2=negSmall2 (1)"
4    "angle==big & velocity==big => force=posBig, force2=negBig2 (1)"

```

Remove the `forceBig` output variable.

```

fis = removeOutput(fis,"force2");

```

View the updated output variables.

`fis.Outputs`

```

ans =
  fisvar with properties:
      Name: "force"
      Range: [-5 5]
  MembershipFunctions: [1x4 fismf]

```

View the updated rules.

`fis.Rules`

```
ans =  
    1x4 fisrule array with properties:
```

```
    Description  
    Antecedent  
    Consequent  
    Weight  
    Connection
```

Details:

Description

```
1    "angle==small & velocity==small => force=negBig (1)"  
2    "angle==small & velocity==big => force=negSmall (1)"  
3    "angle==big & velocity==small => force=posSmall (1)"  
4    "angle==big & velocity==big => force=posBig (1)"
```

`force2` has been removed from the variables and rules of `fis`.

Input Arguments

fisIn — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, specified as a `mamfis` or `sugfis` object.

outputName — Output variable name

string | character vector

Output variable name, specified as a string or character vector.

Output Arguments

fisOut — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, returned as a `mamfis` or `sugfis` object. `fisOut` has the same properties as `fisIn` except:

- The input variable with the specified name is removed.
- The specified input variable is removed from any fuzzy rules. If a rule has only the specified input variable in its antecedent, then the entire rule is removed. If a rule has more than one input variable in its antecedent, then the specified input variable is removed from the antecedent.

See Also

`addOutput` | `fisvar` | `mamfis` | `sugfis`

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

rmmf

(To be removed) Remove membership function from fuzzy inference system

Note rmmf will be removed in a future release. Use removeMF instead. For more information, see “Compatibility Considerations”.

Syntax

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

Description

`fis = rmmf(fis,varType,varIndex,'mf',mfIndex)` removes the membership function, `mfIndex`, of variable type `varType`, of index `varIndex`, from the fuzzy inference system associated with the workspace FIS structure, `fis`:

- Specify `varType` as either 'input' or 'output'.
- `varIndex` is an integer for the index of the variable. This index represents the order in which the variables are listed.
- `mfIndex` is an integer for the index of the membership function. This index represents the order in which the membership functions are listed.

Examples

Remove Membership Function From Variable

Create fuzzy inference system.

```
fis = newfis('mysys');
```

Add an input variable with a single membership function to the system.

```
fis = addvar(fis, 'input', 'temperature', [0 100]);  
fis = addmf(fis, 'input', 1, 'cold', 'trimf', [0 30 60]);
```

View the variable properties.

```
getfis(fis, 'input', 1)  
  
ans = struct with fields:  
    Name: 'temperature'  
    NumMFs: 1  
        mf1: 'cold'  
    range: [0 100]
```

Remove the membership function. To do so, remove membership function 1 from input 1.

```
fis = rmmf(fis, 'input', 1, 'mf', 1);
```

View the variable properties.

```
getfis(fis, 'input', 1)  
  
ans = struct with fields:  
    Name: 'temperature'  
    NumMFs: 0  
    range: [0 100]
```

The variable now has no membership function.

Compatibility Considerations

rmmf will be removed

Not recommended starting in R2018b

rmmf will be removed in a future release. Use removeMF instead. There are differences between these functions that require updates to your code.

The following table shows some typical usages of rmmf and how to update your code to use removeMF instead. Previously, you specified the index of the variable from which you wanted to remove the membership function and the index of the membership function

that you wanted to remove. Now, to remove a membership function, specify the variable name and the membership function name.

If your code has this form:	Use this code instead:
<code>fis = rmmf(fis,'input',1,'mf',1)</code>	<code>fis = removeMF(fis,"service","poor")</code>
<code>fis = rmmf(fis,'output',1,'mf',1)</code>	<code>fis = removeMF(fis,"tip","cheap")</code>

See Also

`addmf` | `addrule` | `addvar` | `plotmf` | `removeMF` | `rmvar`

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

rmvar

(To be removed) Remove variables from fuzzy inference system

Note rmvar will be removed in a future release. Use `removeInput` or `removeOutput` instead. For more information, see “Compatibility Considerations”.

Syntax

```
fis = rmvar(fis,varType,varIndex)
[fis,errorStr] = rmvar(fis,varType,varIndex)
```

Description

`fis = rmvar(fis,varType,varIndex)` removes the variable `varType`, of index `varIndex`, from the fuzzy inference system associated with the workspace FIS structure, `fis`:

- Specify `varType` as either 'input' or 'output'.
- `varIndex` is an integer for the index of the variable. This index represents the order in which the variables are listed.

`[fis,errorStr] = rmvar(fis,varType,varIndex)` returns any error messages to the character vector, `errorStr`.

This command automatically alters the rule list to keep its size consistent with the current number of variables. You must delete from the FIS any rule that contains a variable you want to remove, before removing it. You cannot remove a fuzzy variable currently in use in the rule list.

Examples

Remove Membership Function From Variable

Create fuzzy inference system.

```
fis = newfis('mysys');
```

Add an input variable with a single membership function to the system.

```
fis = addvar(fis,'input','temperature',[0 100]);  
fis = addmf(fis,'input',1,'cold','trimf',[0 30 60]);
```

View the variable properties.

```
getfis(fis,'input',1)  
  
ans = struct with fields:  
    Name: 'temperature'  
    NumMFs: 1  
        mf1: 'cold'  
        range: [0 100]
```

Remove the membership function. To do so, remove membership function 1 from input 1.

```
fis = rmmf(fis,'input',1,'mf',1);
```

View the variable properties.

```
getfis(fis,'input',1)  
  
ans = struct with fields:  
    Name: 'temperature'  
    NumMFs: 0  
        range: [0 100]
```

The variable now has no membership function.

Compatibility Considerations

rmvar will be removed

Not recommended starting in R2018b

rmvar will be removed in a future release. Use `removeInput` or `removeOutput` instead. There are differences between these functions that require updates to your code.

To remove input or output variables from a fuzzy system, use `removeInput` or `removeOutput`, respectively.

This table shows some typical usages of `rmvar` and how to update your code to use `removeInput` or `removeOutput` instead. Previously, you specified the index of the variable that you wanted to remove. Now, to remove a variable, specify the variable name.

If your code has this form:	Use this code instead:
<code>fis = rmvar(fis,'input',1)</code>	<code>fis = removeInput(fis,"service")</code>
<code>fis = rmvar(fis,'output',1)</code>	<code>fis = removeOutput(fis,"tip")</code>

Previously, you had to delete any rules from your fuzzy system that contained the variable you wanted to remove. `removeInput` and `removeOutput` automatically remove these variables from the rule set of your fuzzy system.

See Also

`addmf` | `addrule` | `addvar` | `removeInput` | `removeOutput` | `rmmf`

Introduced before R2006a

ruleedit

Open Rule Editor

Syntax

```
ruleedit(fileName)
```

```
ruleedit(fis)
```

Description

`ruleedit(fileName)` opens the Rule Editor for the FIS stored in the file, `fileName`. Specify `fileName` as a character vector or string with or without the `.fis` extension. You can use the Rule Editor to view or modify the rules of an FIS.

To use this editor to create rules, you must first define all of the input and output variables you want to use with the FIS Editor. You can create the rules using the drop-down and check box choices for input and output variables, connections, and weights. Refer to “The Rule Editor” on page 2-30 for more information about how to use `ruleedit`.

`ruleedit(fis)` operates on a `mamfis` or `sugfis` object, `fis`.

Menu Items

In the Rule Editor, there is a menu bar that allows you to open related UI tools, open and save systems, and so on. The **File** menu for the Rule Editor is the same as the one found **Fuzzy Logic Designer**:

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

Undo to undo the most recent change.

FIS properties to open the **Fuzzy Logic Designer**.

Membership functions to invoke the Membership Function Editor.

- Use the following **View** menu items:

Rules to invoke the Rule Viewer.

Surface to invoke the Surface Viewer.

- Use the **Options** menu items:

Language to select the language: **English**, **Deutsch**, and **Francais**

Format to select the format:

Verbose uses the words "if," "then," "AND," "OR," and so on to create actual sentences.

Symbolic substitutes some symbols for the words used in the verbose mode. For example, "if A AND B then C" becomes "A & B => C."

Indexed mirrors how the rule is stored in the FIS object.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

Apps

Fuzzy Logic Designer

Functions

addrule | mfeddit | ruleview | showrule | surfview

Topics

“The Rule Editor” on page 2-30

Introduced before R2006a

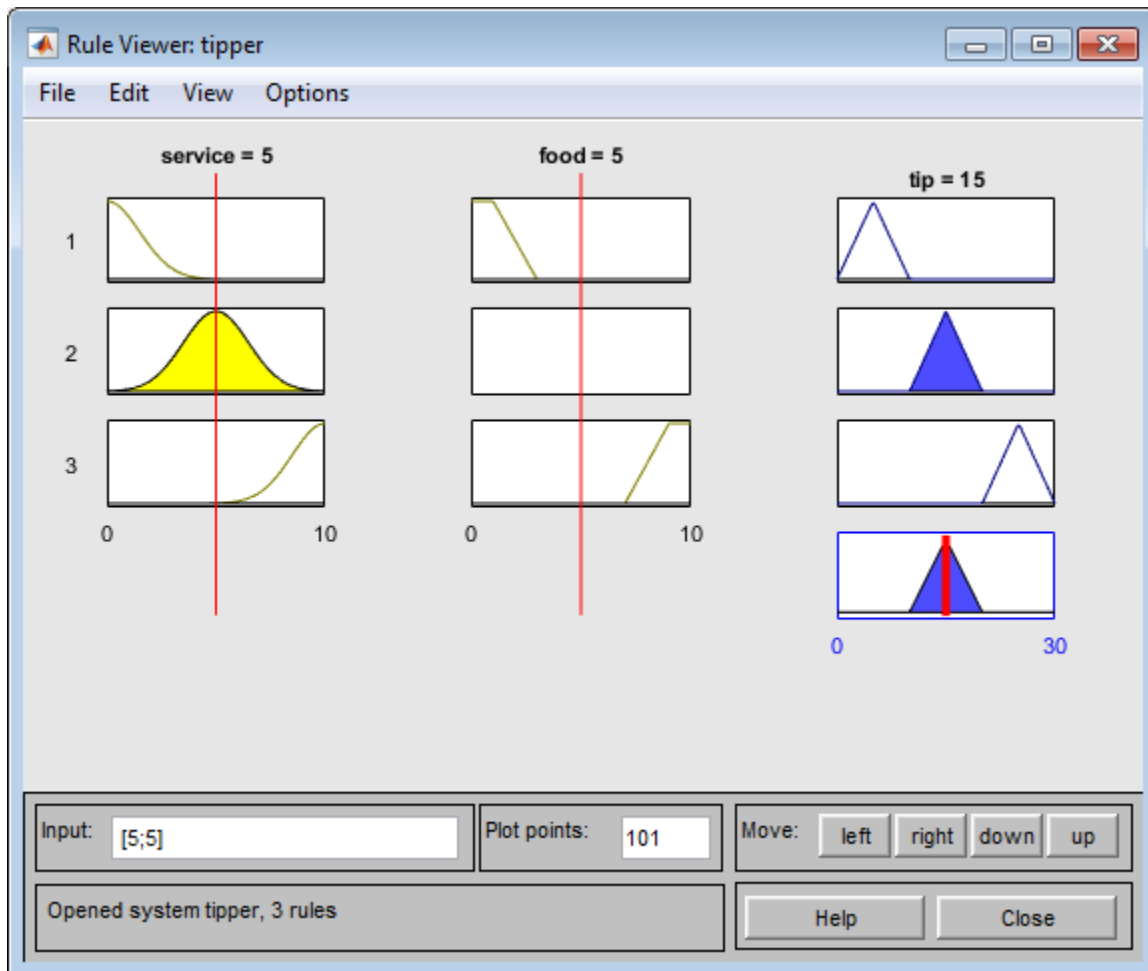
ruleview

Open Rule Viewer

Syntax

```
ruleview(fis)  
ruleview(fileName)
```

Description



`ruleview(fis)` opens the Rule Viewer for the fuzzy inference system, `fis`. Specify `fis` as a `mamfis` or `sugfis` object. The Rule Viewer is used to view the entire implication process from beginning to end. You can move around the line indices that correspond to the inputs and then watch the system readjust and compute the new output. Refer to “The Rule Viewer” on page 2-33 for more information about how to use `ruleview`.

`ruleview(fileName)` depicts the fuzzy inference diagram for the fuzzy inference system stored in file `fileName`. Specify `fileName` as a character vector or string with or without the `.fis` extension.

Menu Items

In the Rule Viewer, there is a menu bar that allows you to open related UI tools, open and save systems, and so on. The **File** menu for the Rule Viewer is the same as the one found in the **Fuzzy Logic Designer**.

- Use the **Edit** menu items:

Undo to undo the most recent action

FIS properties to open the **Fuzzy Logic Designer**

Membership functions to invoke the Membership Function Editor

Rules to invoke the Rule Editor

- Use the **View** menu item:

Surface to invoke the Surface Viewer

- Use the **Options** menu item:

Format to set the format in which the rule appears: **Verbose**, **Symbolic**, or **Indexed**.

If you click on the rule numbers on the left side of the fuzzy inference diagram, the rule associated with that number appears in the status bar at the bottom of the Rule Viewer.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

Apps

Fuzzy Logic Designer

Functions

`addrule` | `mfedit` | `ruleedit` | `showrule` | `surfview`

Topics

“The Rule Viewer” on page 2-33

Introduced before R2006a

setfis

(To be removed) Set fuzzy system properties

Note `setfis` will be removed in a future release. Set fuzzy inference system properties using dot notation instead. For more information, see “Compatibility Considerations”.

Syntax

```
fis = setfis(fis,fisPropertyName,fisPropertyValue)
```

```
fis = setfis(fis,varType,varIndex,varPropertyName,varPropertyValue)
```

```
fis = setfis(fis,varType,varIndex,'mf',mfIndex,mfPropertyName,mfPropertyValue)
```

Description

The command `setfis` can be called with three, five, or seven input arguments, depending on whether you want to set a property of the entire FIS structure, for a particular variable belonging to that FIS structure, or for a particular membership function belonging to one of those variables. The arguments are:

- `fis` — FIS structure in the MATLAB workspace.
- `varType` — Variable type, specified as either `'input'` or `'output'`.
- `varIndex` — Variable index, specified as a positive integer.
- `mfIndex` — Membership function index, specified as a positive integer.
- `fisPropertyName` — FIS property you want to set, specified as one of the following:
 - `'name'`
 - `'type'`
 - `'andmethod'`
 - `'ormethod'`
 - `'impmethod'`

- 'aggmethod'
- 'defuzzmethod'
- `fisPropertyValue` — New value of the FIS property you want to set, specified as a character vector or string.
- `varPropertyName` — Variable property you want to set, specified as either 'name' or 'range'.
- `varPropertyValue` — New value of the variable property you want to set, specified as a character vector or string (for 'name'), or a two-element row vector (for 'range').
- `mfPropertyName` — Membership function property you want to set, specified as either 'name', 'type', or 'params'.
- `mfPropertyValue` — New value of the membership function property you want to set, specified as a character vector or string (for 'name' or 'type'), or a numerical row vector (for 'params').

Examples

Set Fuzzy Inference System Properties

Load a fuzzy inference system.

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

Set the defuzzification method to the bisector method.

```
fis = setfis(fis, 'defuzzmethod', 'bisector');
```

View the defuzzification method of the updated FIS.

```
getfis(fis, 'defuzzmethod')
```

```
ans =  
'bisector'
```

Set Variable Properties in FIS

Load fuzzy inference system.


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

Set the name of the first input variable to 'help'.

```
fis = setfis(fis,'input',1,'name','help');
```

View the name of the variable in the updated system.

```
getfis(fis,'input',1,'name')
```

```
ans =  
'help'
```

Set Membership Function Properties in FIS

Load a fuzzy inference system.

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

Change the type of the second membership function of the first input variable to a triangular membership function.

```
fis = setfis(fis,'input',1,'mf',2,'type','trimf');
```

When changing the type of a membership function, you must also set the parameters accordingly. To convert the original Gaussian membership function parameters to triangular membership function parameters, use the `mf2mf` command.

```
gaussParams = getfis(fis,'input',1,'mf',2,'params');  
triParams = mf2mf(gaussParams,'gaussmf','trimf');
```

Set the membership function parameters to the converted values.

```
fis = setfis(fis,'input',1,'mf',2,'params',triParams);
```

View the updated membership function properties.

```
getfis(fis,'input',1,'mf',2)
```

```
ans = struct with fields:  
    Name: 'good'  
    Type: 'trimf'
```

```
params: [1.4680 5 8.5320]
```

Compatibility Considerations

setfis will be removed

Not recommended starting in R2018b

`setfis` will be removed in a future release. Set fuzzy inference system properties using dot notation instead. There are differences between these approaches that require updates to your code.

This table shows some typical usages of `setfis` for setting fuzzy inference system properties and how to update your code to use dot notation instead.

If your code has this form:	Use this code instead:
<pre>fis = setfis(fis,'andmethod','prod')</pre>	<pre>fis.AndMethod = 'prod'</pre>
<pre>fis = setfis(fis,'input',1,... 'name','service')</pre>	<pre>fis.Inputs(1).Name = "service"</pre>
<pre>fis = setfis(fis,'input',2,... 'mf',1,... params,[5 10 15])</pre>	<pre>fis.Inputs(2).MembershipFunctions(1).Parameters [5 10 15]</pre>

Previously, fuzzy inference systems were represented as structures. Now, fuzzy inference systems are represented as objects. Fuzzy inference system object properties have different names than the corresponding structure fields. For more information on fuzzy inference system objects, see `mamfis` and `sugfis`.

See Also

`getfis`

Introduced before R2006a

showfis

(To be removed) Display annotated Fuzzy Inference System

Note `showfis` will be removed in a future release. View the properties of your FIS directly instead. For more information, see “Compatibility Considerations”.

Syntax

```
showfis(fismat)
```

Description

`showfis(fismat)` prints a version of the MATLAB workspace variable `FIS`, `fismat`, allowing you to see the significance and contents of each field of the structure.

Examples

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

Returns:

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

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

Compatibility Considerations

showfis will be removed

Not recommended starting in R2018b

showfis will be removed in a future release. View the properties of your FIS, myFIS, directly instead. If your code has the form:

```
showfis(myFIS)
```

use this code instead:

```
myFIS
```

To view additional FIS properties, use dot notation. For example, view information about the membership functions of the first input variable.

```
myFIS.Inputs(1).MembershipFunctions
```

For more information on fuzzy inference systems and their properties, see `mamfis` and `sugfis`.

See Also

`getfis`

Introduced before R2006a

showrule

Display fuzzy inference system rules

Syntax

```
showrule(fis)
showrule(fis,Name,Value)
```

Description

`showrule(fis)` displays the rules in the fuzzy inference system, `fis`.

`showrule(fis,Name,Value)` displays rules using options specified by one or more `Name,Value` pair arguments.

Examples

Display All Rules for a Fuzzy Inference System

Load fuzzy inference system.

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

Display rules using linguistic expressions.

```
showrule(fis)
```

```
ans = 3x78 char array
    '1. If (service is poor) or (food is rancid) then (tip is cheap) (1)      '
    '2. If (service is good) then (tip is average) (1)                      '
    '3. If (service is excellent) or (food is delicious) then (tip is generous) (1)'
```

Display rules using symbolic expressions.

```
showrule(fis, 'Format', 'symbolic')
ans = 3x65 char array
'1. (service==poor) | (food==rancid) => (tip=cheap) (1)      '
'2. (service==good) => (tip=average) (1)                    '
'3. (service==excellent) | (food==delicious) => (tip=generous) (1)'
```

Display rules using membership function indices.

```
showrule(fis, 'Format', 'indexed')
ans = 3x15 char array
'1 1, 1 (1) : 2 '
'2 0, 2 (1) : 1 '
'3 2, 3 (1) : 2 '
```

Select Fuzzy Rules to Display

Load fuzzy inference system.

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

Display the first and third rules.

```
showrule(fis, 'RuleIndex', [1 3])
```

```
ans = 2x78 char array
'1. If (service is poor) or (food is rancid) then (tip is cheap) (1)      '
'3. If (service is excellent) or (food is delicious) then (tip is generous) (1)'
```

Display Fuzzy Rules in German Language

Load fuzzy inference system.

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

Display the rules in German using the 'deutsch' language.

```
showrule(fis, 'Language', 'deutsch')  
  
ans = 3x85 char array  
    '1. Wenn (service ist poor) oder (food ist rancid) dann (tip ist cheap) (1)  
    '2. Wenn (service ist good) dann (tip ist average) (1)  
    '3. Wenn (service ist excellent) oder (food ist delicious) dann (tip ist generous)
```

Input Arguments

fis — Fuzzy inference system

mamfis object | sugfis object

Fuzzy inference system, specified as a mamfis or sugfis object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Format', 'symbolic'` sets the rule display format to use language-neutral symbolic expressions.

RuleIndex — Rules to display

positive integer | vector of positive integers

Rules to display, specified as the comma-separated pair consisting of `'RuleIndex'` and one of the following:

- Positive integer — Index of a single rule to display
- Vector of positive integers — Indices of multiple rules to display

The default vector includes the indices for all the rules in `fis`.

Format — Rule format

'verbose' (default) | 'symbolic' | 'indexed'

Rule format, specified as the comma-separated pair consisting of 'Format' and one of the following:

- 'verbose' — Use linguistic expressions.

```
'If (service is poor) or (food is rancid) then (tip is cheap) (1)'
```

The rule weight is displayed in parentheses at the end of the rule.

You can specify the rule language using the Language option.

- 'symbolic' — Use language-neutral symbolic expressions.

```
'(service==poor) | (food==rancid) => (tip=cheap) (1)'
```

The symbolic rules use the following symbols.

Rule Component	Symbol
AND	&
OR	
IS (in antecedent)	==
IS (in consequent)	=
IS NOT	~=
Implication (then)	=>

The rule weight is displayed in parentheses at the end of the rule.

- 'indexed' — Use input and output membership function (MF) indices and integer representation of fuzzy operators.

The indexed rules display in the following format:

```
'<input MFs>, <output MFs>, (<weight>) : <logical operator - 1 (AND), 2 (OR)>'
```

For example:

```
'1 1, 1 (1) : 2'
```

To indicate NOT operations for input and output membership functions, the software uses negative indices. For example, to indicate “not the second membership function,” the software uses -2.

To indicate a don't care condition for an input or output membership function, the software uses \emptyset .

Language — Rule language

'english' (default) | 'français' | 'deutsch'

Rule language for 'verbose' format, specified as the comma-separated pair consisting of 'Language' and one of the following:

- 'english' — Display rules in English.

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

- 'français' — Display rules in French.

'Si (service est poor) ou (food est rancid) alors (tip est cheap) (1)'

- 'deutsch' — Display rules in German.

'Wenn (service ist poor) oder (food ist rancid) dann (tip ist cheap) (1)'

The software displays the FIS rules using the following keywords.

Rule Component	English	French	German
Start of antecedent	if	si	wenn
AND	and	et	und
OR	or	ou	oder
Start of consequent (implication)	then	alors	dann
IS	is	est	ist
IS NOT	is not	n'est_pas	ist nicht

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

`addrule` | `ruleedit`

Introduced before R2006a

sigmf

Sigmoidal membership function

Syntax

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

Description

The sigmoidal function, `sigmf(x,[a c])`, as given in the following equation by $f(x,a,c)$ is a mapping on a vector x , and depends on two parameters a and c .

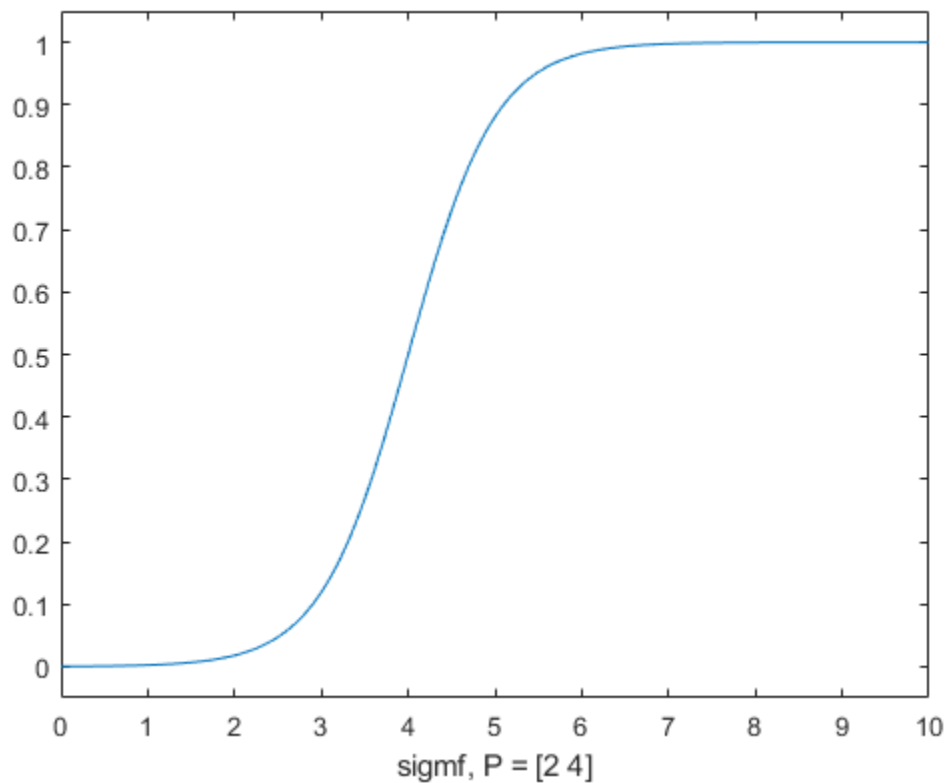
$$f(x,a,c) = \frac{1}{1 + e^{-a(x-c)}}$$

Depending on the sign of the parameter a , the sigmoidal membership function is inherently open to the right or to the left, and thus is appropriate for representing concepts such as “very large” or “very negative.” More conventional-looking membership functions can be built by taking either the product or difference of two different sigmoidal membership functions. For more information see `dsigmf` and `psigmf`.

Examples

Sigmoidal Membership Function

```
x = 0:0.1:10;  
y = sigmf(x,[2 4]);  
plot(x,y)  
xlabel('sigmf, P = [2 4]')  
ylim([-0.05 1.05])
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dsigmf` | `evalmf` | `gauss2mf` | `gaussmf` | `gbellmf` | `mf2mf` | `pimf` | `psigmf` | `smf` |
`trapmf` | `trapmf` | `trimf` | `trimf` | `zmf`

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

smf

S-shaped membership function

Syntax

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

Description

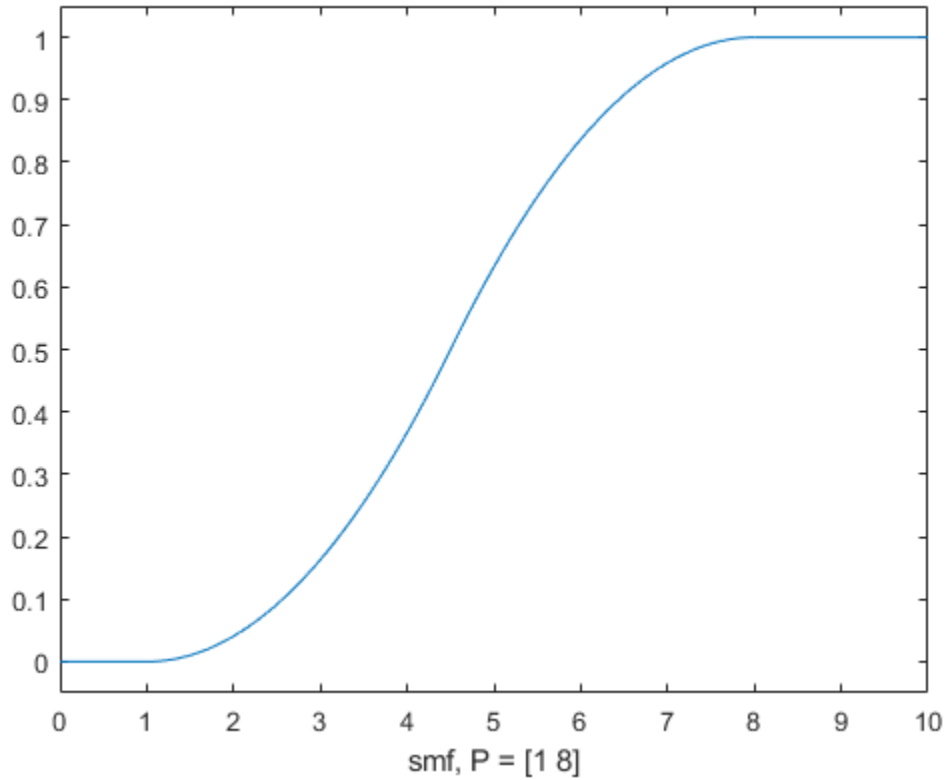
This spline-based curve is a mapping on the vector x , and is named because of its S-shape. The parameters a and b locate the extremes of the sloped portion of the curve, as given by:

$$f(x;a,b) = \begin{cases} 0, & x \leq a \\ 2\left(\frac{x-a}{b-a}\right)^2, & a \leq x \leq \frac{a+b}{2} \\ 1-2\left(\frac{x-b}{b-a}\right)^2, & \frac{a+b}{2} \leq x \leq b \\ 1, & x \geq b \end{cases}$$

Examples

S-Shaped Membership Function

```
x = 0:0.1:10;
y = smf(x,[1 8]);
plot(x,y)
xlabel('smf, P = [1 8]')
ylim([-0.05 1.05])
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dsigmf` | `evalmf` | `gauss2mf` | `gaussmf` | `gbellmf` | `mf2mf` | `pimf` | `psigmf` | `sigmf` | `trapmf` | `trapmf` | `trimf` | `trimf` | `zmf`

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

subclust

Find cluster centers using subtractive clustering

Syntax

```
centers = subclust(data,clusterInfluenceRange)
centers = subclust(data,clusterInfluenceRange,Name,Value)
[centers,sigma] = subclust( ___ )
```

Description

`centers = subclust(data,clusterInfluenceRange)` clusters input data using subtractive clustering with the specified cluster influence range, and returns the computed cluster centers. The subtractive clustering algorithm on page 8-261 estimates the number of clusters in the input data.

`centers = subclust(data,clusterInfluenceRange,Name,Value)` clusters data using algorithm options specified by one or more `Name, Value` pair arguments.

`[centers,sigma] = subclust(___)` returns the sigma values specifying the range of influence of a cluster center in each of the data dimensions.

Examples

Find Cluster Centers Using Subtractive Clustering

Load data set.

```
load clusterdemo.dat
```

Find cluster centers using the same range of influence for all dimensions.

```
C = subclust(clusterdemo,0.6);
```

Each row of `C` contains one cluster center.

C

C = 3×3

```
0.5779    0.2355    0.5133
0.7797    0.8191    0.1801
0.1959    0.6228    0.8363
```

Specify Bounds for Subtractive Clustering

Load data set.

```
load clusterdemo.dat
```

Define minimum and maximum normalization bounds for each data dimension. Use the same bounds for each dimension.

```
dataScale = [-0.2 -0.2 -0.2;
             1.2  1.2  1.2];
```

Find cluster centers.

```
C = subclust(clusterdemo,0.5, 'DataScale', dataScale);
```

Specify Options for Subtractive Clustering

Load data set.

```
load clusterdemo.dat
```

Specify the following clustering options:

- Squash factor of 2.0 - Only find clusters that are far from each other.
- Accept ratio 0.8 - Only accept data points with a strong potential for being cluster centers.
- Reject ratio of 0.7 - Reject data points if they do not have a strong potential for being cluster centers.

- Verbosity flag of 0 - Do not print progress information to the command window.

```
options = [2.0 0.8 0.7 0];
```

Find cluster centers, using a different range of influence for each dimension and the specified options.

```
C = subclust(clusterdemo,[0.5 0.25 0.3], 'Options', options);
```

Obtain Cluster Influence Range for Each Data Dimension

Load data set.

```
load clusterdemo.dat
```

Cluster data, returning cluster sigma values, S.

```
[C,S] = subclust(clusterdemo,0.5);
```

Cluster sigma values indicate the range of influence of the computed cluster centers in each data dimension.

Input Arguments

data — Data set to be clustered

M-by-*N* array

Data to be clustered, specified as an *M*-by-*N* array, where *M* is the number of data points and *N* is the number of data dimensions.

clusterInfluenceRange — Range of influence of the cluster center

scalar value in the range [0, 1] | vector

Range of influence of the cluster center for each input and output assuming the data falls within a unit hyperbox, specified as the comma-separated pair consisting of 'ClusterInfluenceRange' one of the following:

- Scalar value in the range [0 1] — Use the same influence range for all inputs and outputs.

- **Vector** — Use different influence ranges for each input and output.

Specifying a smaller range of influence usually creates more and smaller data clusters, producing more fuzzy rules.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'DataScale', 'auto'` sets the normalizing factors for the input and output signals using the minimum and maximum values in the data set to be clustered.

DataScale — Data scale factors

`'auto'` (default) | 2-by- N array

Data scale factors for normalizing input and output data into a unit hyperbox, specified as the comma-separated pair consisting of `'DataScale'` and a 2-by- N array, where N is the total number of inputs and outputs. Each column of `DataScale` specifies the minimum value in the first row and the maximum value in the second row for the corresponding input or output data set.

When `DataScale` is `'auto'`, the `genfis` command uses the actual minimum and maximum values in the data to be clustered.

Options — Clustering options

vector

Clustering options, specified as the comma-separated pair consisting of `'Options'` and a vector with the following elements:

Options(1) — Squash factor

1.25 (default) | positive scalar

Squash factor for scaling the range of influence of cluster centers, specified as a positive scalar. A smaller squash factor reduces the potential for outlying points to be considered as part of a cluster, which usually creates more and smaller data clusters.

Options(2) — Acceptance ratio

0.5 (default) | scalar value in the range $[0, 1]$

Acceptance ratio, defined as a fraction of the potential of the first cluster center, above which another data point is accepted as a cluster center, specified as a scalar value in the range [0, 1]. The acceptance ratio must be greater than the rejection ratio.

Options(3) — Rejection ratio

0.15 (default) | scalar value in the range [0, 1]

Rejection ratio, defined as a fraction of the potential of the first cluster center, below which another data point is rejected as a cluster center, specified as a scalar value in the range [0, 1]. The rejection ratio must be less than acceptance ratio.

Options(4) — Information display flag

false (default) | true

Information display flag indicating whether to display progress information during clustering, specified as one of the following:

- `false` — Do not display progress information.
- `true` — Display progress information.

Output Arguments

centers — Cluster centers

J-by-*N* array

Cluster centers, returned as a *J*-by-*N* array, where *J* is the number of clusters and *N* is the number of data dimensions.

sigma — Range of influence of cluster centers

N-element row vector

Range of influence of cluster centers for each data dimension, returned as an *N*-element row vector. All cluster centers have the same set of `sigma` values.

Tips

- To generate a fuzzy inference system using subtractive clustering, use the `genfis` command. For example, suppose you cluster your data using the following syntax:

```
C = subclust(data,clusterInfluenceRange,'DataScale',dataScale,'Options',options);
```

where the first M columns of `data` correspond to input variables, and the remaining columns correspond to output variables.

You can generate a fuzzy system using the same training data and subtractive clustering configuration. To do so:

- 1 Configure clustering options.

```
opt = genfisOptions('SubtractiveClustering');
opt.ClusterInfluenceRange = clusterInfluenceRange;
opt.DataScale = dataScale;
opt.SquashFactor = options(1);
opt.AcceptRatio = options(2);
opt.RejectRatio = options(3);
opt.Verbose = options(4);
```

- 2 Extract input and output variable data.

```
inputData = data(:,1:M);
outputData = data(:,M+1:end);
```

- 3 Generate FIS structure.

```
fis = genfis(inputData,outputData,opt);
```

The fuzzy system, `fis`, contains one fuzzy rule for each cluster, and each input and output variable has one membership function per cluster. You can generate only Sugeno fuzzy systems using subtractive clustering. For more information, see `genfis` and `genfisOptions`.

Algorithms

Subtractive clustering assumes that each data point is a potential cluster center. The algorithm does the following:

- 1 Calculate the likelihood that each data point would define a cluster center, based on the density of surrounding data points.
- 2 Choose the data point with the highest potential to be the first cluster center.
- 3 Remove all data points near the first cluster center. The vicinity is determined using `clusterInfluenceRange`.
- 4 Choose the remaining point with the highest potential as the next cluster center.

- 5 Repeat steps 3 and 4 until all the data is within the influence range of a cluster center.

The subtractive clustering method is an extension of the mountain clustering method proposed in [2].

References

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

See Also

genfis

Topics

"Fuzzy Clustering" on page 4-2

"Model Suburban Commuting Using Subtractive Clustering" on page 4-21

Introduced before R2006a

surfview

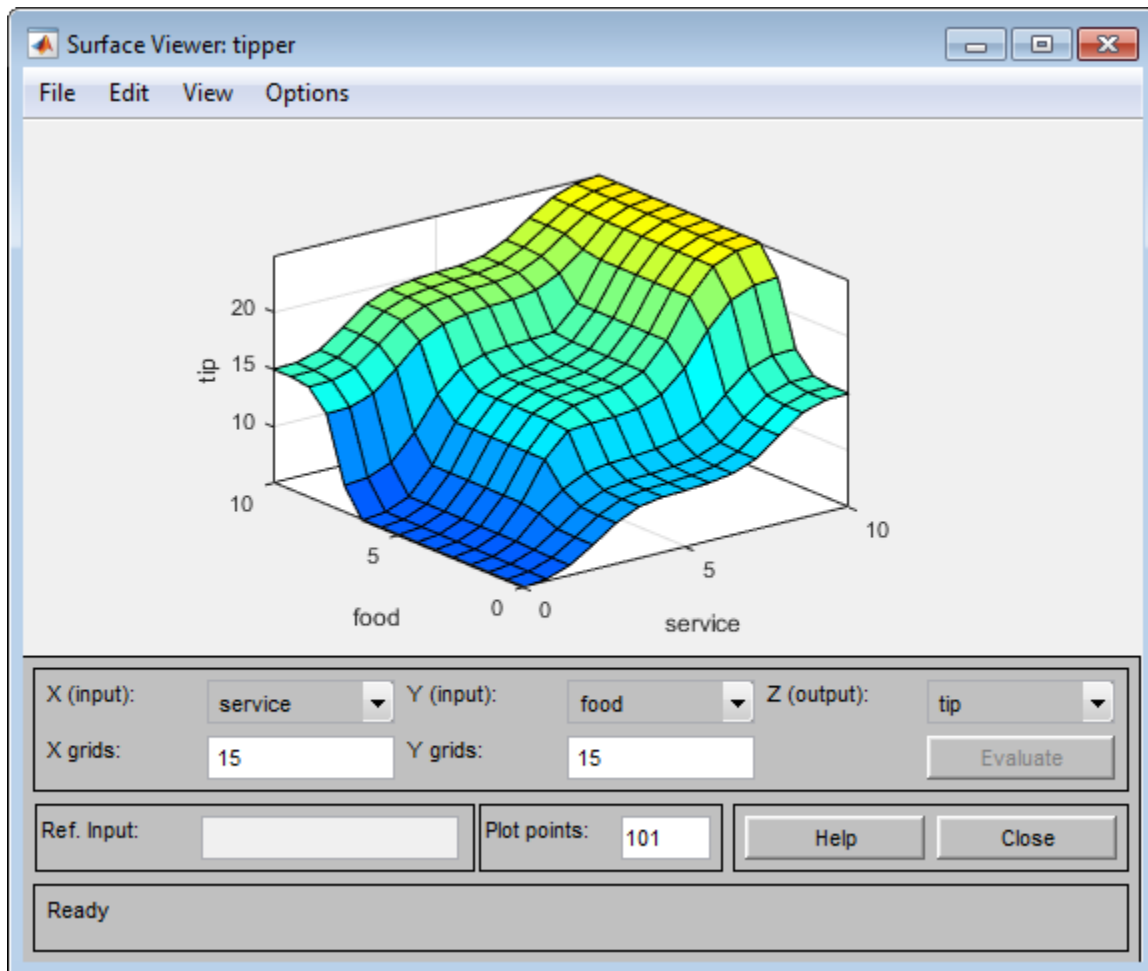
Open Surface Viewer

Syntax

```
surfview(fis)
```

```
surfview(fileName)
```

Description



The Surface Viewer is a graphical interface that lets you examine the output surface of an FIS for any one or two inputs. You can examine an FIS that is:

- Stored in a file using `surfview(fileName)`, where `fileName` is a character vector or string with or without the `.fis` extension.

- In the MATLAB workspace using `surfview(fis)`, where `fis` is a `mamfis` or `sugfis` object.

Because it does not alter the fuzzy system or its associated FIS object in any way, Surface Viewer is a read-only editor. Using the drop-down menus, you select the two input variables you want assigned to the two input axes (X and Y), as well the output variable you want assigned to the output (or Z) axis.

By default, the surface plot updates automatically when you change the input or output variable selections or the number of grid points. To disable automatic plot updates, in the **Options** menu, clear the **Always evaluate** option. When this option is disabled, to update the plot, click **Evaluate**.

If you want to create a smoother plot, use the **Plot points** field to specify the number of points on which the membership functions are evaluated in the input or output range. This field defaults to the minimum number of plot plots, 101. If you specify fewer plot points, the field value automatically resets to 101. When you specify the number of plot points, the surface plot automatically updates.

By clicking on the plot axes and dragging the mouse, you can manipulate the surface so that you can view it from different angles.

If there are more than two inputs to your system, you must supply the constant values associated with any unspecified inputs in the reference input section.

Refer to “The Surface Viewer” on page 2-35 for more information about how to use `surfview`.

Menu Items

In the Surface Viewer, there is a menu bar that allows you to open related UI tools, open and save systems, and so on. The Surface Viewer uses the same **File** menu as the one on the **Fuzzy Logic Designer**:

- Use the **Edit** menu items:

Undo to undo the most recent action

FIS properties to open the **Fuzzy Logic Designer**

Membership functions to invoke the Membership Function Editor

Rules... to invoke the Rule Editor

- Use the **View** menu item:

Rules to invoke the Rule Viewer

- Use the **Options** menu items:

Plot to choose among eight different kinds of plot styles.

Color Map to choose among several different color schemes.

Always evaluate to automatically evaluate and plot a new surface every time you make a change that affects the plot, such as changing the number of grid points. This option is selected by default. To clear this option, select it once more.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

Apps
Fuzzy Logic Designer

Functions

gensurf | mfeedit | ruleedit | ruleview

Topics

“The Surface Viewer” on page 2-35

Introduced before R2006a

trapmf

Trapezoidal-shaped membership function

Syntax

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

Description

The trapezoidal curve is a function of a vector, x , and depends on four scalar parameters a , b , c , and d , as given by

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

or, more compactly, by

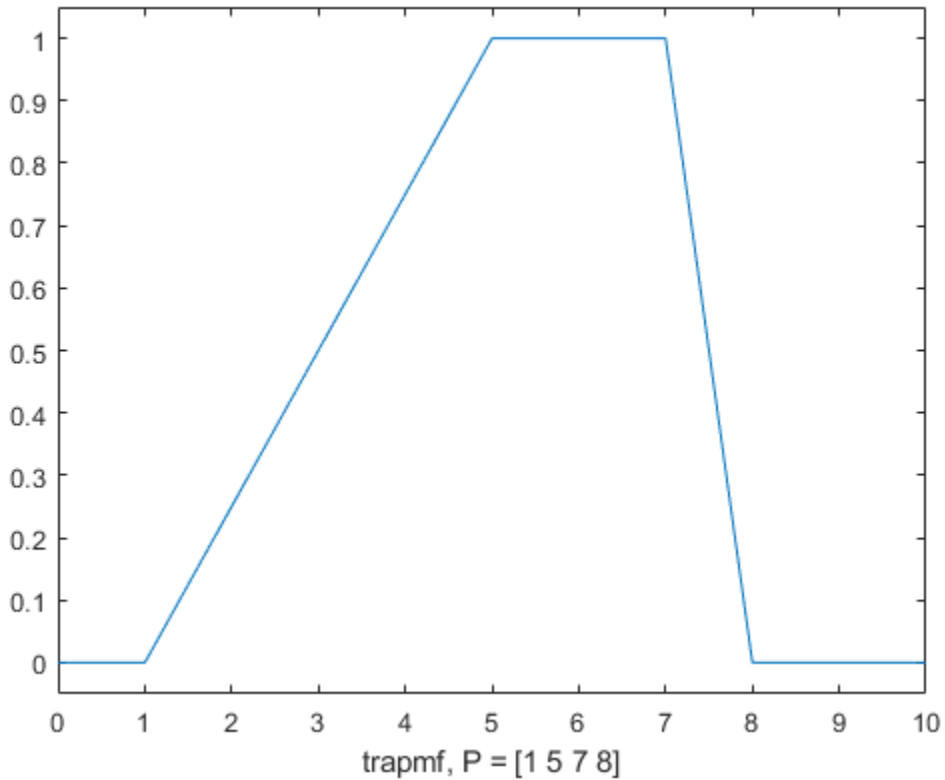
$$f(x; a, b, c, d) = \max\left(\min\left(\frac{x-a}{b-a}, 1, \frac{d-x}{d-c}\right), 0\right)$$

The parameters a and d locate the “feet” of the trapezoid and the parameters b and c locate the “shoulders.”

Examples

Trapezoid-Shaped Membership Function

```
x = 0:0.1:10;  
y = trapmf(x,[1 5 7 8]);  
plot(x,y)  
xlabel('trapmf, P = [1 5 7 8]')  
ylim([-0.05 1.05])
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

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

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

trimf

Triangular-shaped membership function

Syntax

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

Description

The triangular curve is a function of a vector, x , and depends on three scalar parameters a , b , and c , as given by

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

or, more compactly, by

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

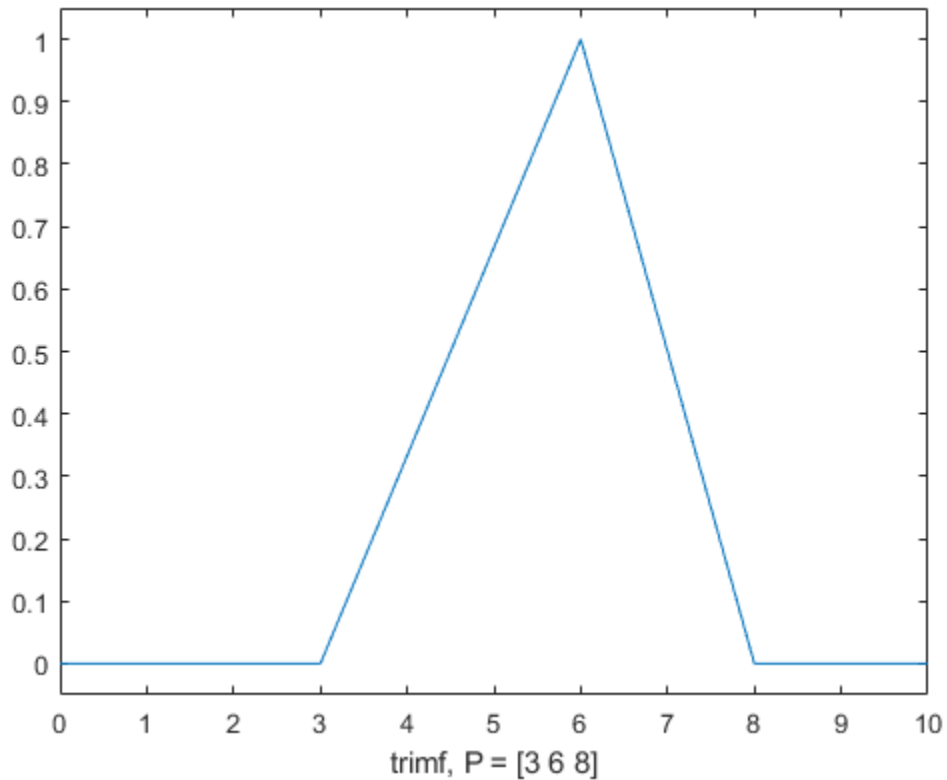
The parameters a and c locate the “feet” of the triangle and the parameter b locates the peak.

Examples

Triangle-Shaped Membership Function

```
x = 0:0.1:10;
y = trimf(x,[3 6 8]);
```

```
plot(x,y)
xlabel('trimf, P = [3 6 8]')
ylim([-0.05 1.05])
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

dsigmf | evalmf | gauss2mf | gaussmf | gbellmf | mf2mf | pimf | psigmf | sigmf | smf | trapmf | trapmf | trimf | zmf

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

update

Update fuzzy rule using fuzzy inference system

Syntax

```
ruleOut = update(ruleIn, fis)
```

Description

`ruleOut = update(ruleIn, fis)` updates the fuzzy rule `ruleIn` using the information in fuzzy inference system `fis` and returns the resulting fuzzy rule in `ruleOut`.

Examples

Create Fuzzy Rule Using Text Description

Create a fuzzy rule using a verbose text description.

```
rule = fisrule("if service is poor and food is delicious then tip is average (1)");
```

Alternatively, you can specify the same rule using a symbolic text description.

```
rule = fisrule("service==poor & food==delicious => tip=average")
```

```
rule =
```

```
  fisrule with properties:
```

```
    Description: "service==poor & food==delicious => tip=average (1)"
```

```
    Antecedent: []
```

```
    Consequent: []
```

```
    Weight: 1
```

```
    Connection: 1
```

Before using `rule` with a fuzzy system, update the rule `Antecedent` and `Consequent` properties using the `update` function.

```
fis = readfis("tipper");
rule = update(rule,fis)

rule =
  fisrule with properties:

    Description: "service==poor & food==delicious => tip=average (1)"
    Antecedent: [1 2]
    Consequent: 2
    Weight: 1
    Connection: 1
```

Create Fuzzy Rule Using Numeric Description

Create a fuzzy rule using a numeric description. Specify that the rule has two input variables.

```
rule = fisrule([1 2 2 0.5 1],2)

rule =
  fisrule with properties:

    Description: "input1==mf1 & input2==mf2 => output1=mf2 (0.5)"
    Antecedent: [1 2]
    Consequent: 2
    Weight: 0.5000
    Connection: 1
```

Before using rule with a fuzzy system, update the rule Description property using the update function.

```
fis = readfis("tipper");
rule = update(rule,fis)

rule =
  fisrule with properties:

    Description: "service==poor & food==delicious => tip=average (0.5)"
    Antecedent: [1 2]
    Consequent: 2
```

Weight: 0.5000
Connection: 1

Input Arguments

ruleIn — Fuzzy rule

`fisrule` object | array of `fisrule` objects

Fuzzy rule, specified as a `fisrule` object or an array of `fisrule` objects. If `ruleIn` was created using a:

- Text description, its `Antecedent` and `Consequent` properties are updated using the input and output membership function indices in `fis` that correspond to the membership function names in the `Description` property of `ruleIn`.
- Numeric description, its `Description` property is updated using the input and output membership function names in `fis` that correspond to the membership function indices in the `Antecedent` and `Consequent` properties of `ruleIn`.

If you specify `ruleIn` as an array of `fisrule` objects, then all of the rules are updated accordingly.

fis — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, specified as a `mamfis` or `sugfis` object.

Output Arguments

ruleOut — Fuzzy rule

`fisrule` object | array of `fisrule` objects

Fuzzy rule, returned as a `fisrule` object or an array of `fisrule` objects.

See Also

`fisrule` | `mamfis` | `sugfis`

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

writeFIS

Save fuzzy inference system to file

You can save a fuzzy inference system (FIS) in a `.fis` file using the `writeFIS` function. To load the saved file, use the `readfis` function.

Do not manually edit the contents of a `.fis` file. Doing so can produce unexpected results when loading the file using `readfis`.

Syntax

```
writeFIS(fis, fileName)
```

```
writeFIS(fis)  
writeFIS(fis, fileName, "dialog")
```

Description

`writeFIS(fis, fileName)` saves the fuzzy inference system `fis` to the current working folder using file name `fileName`.

`writeFIS(fis)` opens a dialog box for saving a FIS. In this dialog box, specify the name and location of the `.fis` file.

`writeFIS(fis, fileName, "dialog")` opens a dialog box for saving a FIS, setting the name of the file in the dialog box to `fileName`. In the dialog box, specify the location for the file.

Examples

Save Fuzzy Inference System to File

Create a fuzzy inference system, and add an input variable with membership functions.


```
fis = mamfis('Name','tipper');
fis = addInput(fis,[0 10],'Name','service');
fis = addMF(fis,'service','gaussmf',[1.5 0],'Name','poor');
fis = addMF(fis,'service','gaussmf',[1.5 5],'Name','good');
fis = addMF(fis,'service','gaussmf',[1.5 10],'Name','excellent');
```

Save the fuzzy system in the current working folder in the file `myFile.fis`.

```
writefis(fis,'myFile');
```

Input Arguments

fis — Fuzzy inference system

`mamfis` object | `sugfis` object

Fuzzy inference system, specified as one of the following:

- `mamfis` object — Mamdani fuzzy inference system
- `sugfis` object — Sugeno fuzzy inference system

fileName — File name

string | character vector

File name specified as a string or character vector. If you do not specify the `.fis` extension in the file name, `writeFIS` adds the extension.

Compatibility Considerations

writefis is now **writeFIS**

Behavior changed in R2018b

`writefis` is now `writeFIS`. To update your code, change the function name from `writefis` to `writeFIS`. The syntaxes are equivalent.

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

See Also

`readfis`

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

zmf

Z-shaped membership function

Syntax

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

Description

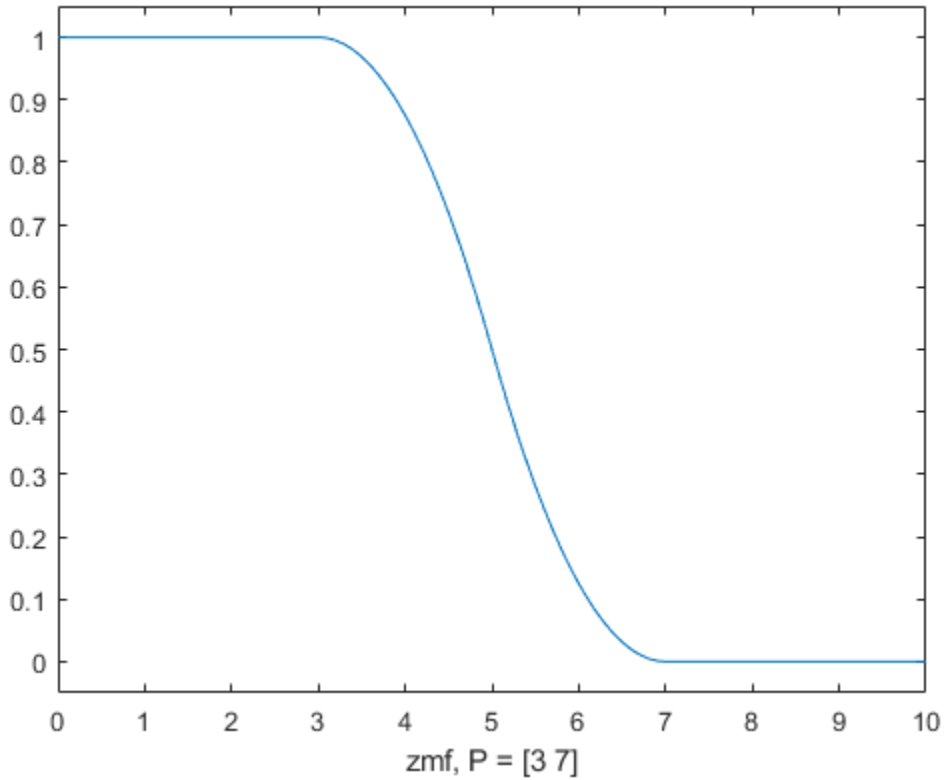
This spline-based function of x is so named because of its Z-shape. The parameters a and b locate the extremes of the sloped portion of the curve as given by.

$$f(x; a, b) = \begin{cases} 1, & x \leq a \\ 1 - 2\left(\frac{x-a}{b-a}\right)^2, & a \leq x \leq \frac{a+b}{2} \\ 2\left(\frac{x-b}{b-a}\right)^2, & \frac{a+b}{2} \leq x \leq b \\ 0, & x \geq b \end{cases}$$

Examples

Z-Shaped Membership Function

```
x = 0:0.1:10;
y = zmf(x, [3 7]);
plot(x,y)
xlabel('zmf, P = [3 7]')
ylim([-0.05 1.05])
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

dsigmf | evalmf | gauss2mf | gaussmf | gbellmf | mf2mf | pimf | psigmf | sigmf | smf | trapmf | trapmf | trimf | trimf

Topics

“Membership Functions” on page 1-14

“The Membership Function Editor” on page 2-22

Introduced before R2006a

Objects – Alphabetical List

evalfisOptions

Option set for evalfis function

Description

Use an evalfisOptions object to specify options for the evalfis function.

Creation

Syntax

```
opt = evalfisOptions  
opt = evalfisOptions(Name,Value)
```

Description

opt = evalfisOptions creates an option set for the evalfis function with default options. To modify the properties of this option set, use dot notation.

opt = evalfisOptions(Name,Value) sets properties using name-value pairs. For example, evalfisOptions('NumSamplePoints',51) creates an option set and sets the number of output fuzzy set samples to 51. You can specify multiple name-value pairs. Enclose each property name in single quotes.

Properties

NumSamplePoints — Number of sample points in output fuzzy sets

101 (default) | integer greater than 1

Number of sample points in output fuzzy sets, specified as an integer greater than 1.

To reduce memory usage while evaluating a Mamdani FIS, specify fewer samples. Doing so sacrifices the accuracy of the defuzzified output value.

Reducing the number of samples can make the output area for defuzzification zero. In this case, the defuzzified output value is the midpoint of the output variable range.

Note `evalfis` ignores this property when evaluating a Sugeno FIS.

OutOfRangeInputValueMessage — Diagnostic message behavior when an input is out of range

"warning" (default) | "error" | "none"

Diagnostic message behavior when an input is out of range, specified as one of the following:

- "warning" — Report the diagnostic message as a warning.
- "error" — Report the diagnostic message as an error.
- "none" — Do not report the diagnostic message.

When an input value is out of range, corresponding rules in the fuzzy system can have unexpected firing strengths.

NoRuleFiredMessage — Diagnostic message behavior when no rules fire

"warning" (default) | "error" | "none"

Diagnostic message behavior when no rules fire, specified as one of the following:

- "warning" — Report the diagnostic message as a warning.
- "error" — Report the diagnostic message as an error.
- "none" — Do not report the diagnostic message.

When `NoRuleFiredMessage` is "warning" or "none" and no rules fire for a given output, the defuzzified output value is set to its mean range value.

EmptyOutputFuzzySetMessage — Diagnostic message behavior when an output fuzzy set is empty

"warning" (default) | "error" | "none"

Diagnostic message behavior when an output fuzzy set is empty, specified as one of the following:

- "warning" — Report the diagnostic message as a warning.

- "error" — Report the diagnostic message as an error.
- "none" — Do not report the diagnostic message.

When `EmptyOutputFuzzySetMessage` is "warning" or "none" and an output fuzzy set is empty, the defuzzified value for the corresponding output is set to its mean range value.

This diagnostic message applies only to Mamdani systems.

Object Functions

`evalfis` Evaluate fuzzy inference system

Examples

Create Option Set for Evaluating FIS

Create option set object, specifying the number of sample points for output fuzzy sets.

```
options = evalfisOptions('NumSamplePoints',51)
```

```
options =  
    EvalFISOptions with properties:  
  
        NumSamplePoints: 51  
    OutOfRangeInputValueMessage: "warning"  
        NoRuleFiredMessage: "warning"  
    EmptyOutputFuzzySetMessage: "warning"
```

Alternatively, create a default option set, and configure properties using dot notation.

```
options = evalfisOptions;  
options.NumSamplePoints = 51;
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When used for code generation, an `evalfisOptions` object stores its `OutOfRangeInputValueMessage`, `NoRuleFiredMessage`, and `EmptyOutputFuzzySetMessage` properties as character vectors rather than strings.
- When evaluating a fuzzy inference system in Simulink, it is recommended to not use `evalfis` or `evalfisOptions` within a MATLAB Function. Instead, evaluate your fuzzy inference system using the Fuzzy Logic Controller block.

See Also

Functions

`evalfis`

Introduced in R2018a

fismf

Fuzzy membership function

Description

Use `fismf` objects to represent fuzzy membership functions. For each input and output variable in a fuzzy inference system (FIS), one or more membership functions define the possible linguistic sets for that variable. For more information on membership functions, see “Foundations of Fuzzy Logic” on page 1-10.

Creation

Syntax

```
mf = fismf
mf = fismf(type,parameters)
mf = fismf('Name',name)
mf = fismf(type,parameters,'Name',name)
```

Description

`mf = fismf` creates a fuzzy membership function (MF) with default `type`, `parameters`, and `name`. To change the membership function properties, use dot notation.

`mf = fismf(type,parameters)` sets the `Type` and `Parameters` properties.

`mf = fismf('Name',name)` sets the `Name` property.

`mf = fismf(type,parameters,'Name',name)` sets the `Type`, `Parameters`, and `Name` properties.

Properties

Name — Membership function name

"mf" (default) | string | character vector

Membership function name, specified as a string or character vector.

Type — Membership function type

"trimf" (default) | string | character vector | function handle

Membership function type, specified as a string or character vector that contains the name of a function in the current working folder or on the MATLAB path. You can also specify a handle to such a function. When you specify Type, you must also specify Parameters.

This table describes the values that you can specify for Type.

Membership Function Type	Description	For More Information
"gbellmf"	Generalized bell-shaped membership function	gbellmf
"gaussmf"	Gaussian membership function	gaussmf
"gauss2mf"	Gaussian combination membership function	gauss2mf
"trimf"	Triangular membership function	trimf
"trapmf"	Trapezoidal membership function	trapmf
"sigmf"	Sigmoidal membership function	sigmf
"dsigmf"	Difference between two sigmoidal membership functions	dsigmf
"psigmf"	Product of two sigmoidal membership functions	psigmf
"zmf"	Z-shaped membership function	zmf
"pimf"	Pi-shaped membership function	pimf
"smf"	S-shaped membership function	smf

Membership Function Type	Description	For More Information
"constant"	Constant membership function for Sugeno output membership functions	"What Is Sugeno-Type Fuzzy Inference?" on page 2-5
"linear"	Linear membership function for Sugeno output membership functions	
String or character vector	Name of a custom membership function in the current working folder or on the MATLAB path. Custom output membership functions are not supported for Sugeno systems.	"Build Fuzzy Systems Using Custom Functions" on page 2-50
Function handle	Handle to a custom membership function in the current working folder or on the MATLAB path. Custom output membership functions are not supported for Sugeno systems.	

Note When you change Type using dot notation, the values in Parameters are automatically converted for the new membership function type.

Parameters — Membership function parameters

[0 0.5 1] (default) | vector

Membership function parameters, specified as a vector. The length of the parameter vector depends on the membership function type. When you specify Parameters, you must also specify Type.

Object Functions

evalmf Evaluate fuzzy membership function

Examples

Create Membership Function

Create fuzzy membership function with default settings.

```
mf = fismf;
```

To modify the membership function settings, use dot notation. For example, specify a Gaussian membership function with a standard deviation of 2 and a mean of 10.

```
mf.Type = "gaussmf";  
mf.Parameters = [2 10];
```

Create Membership Function with Specified Parameters

Create a trapezoidal membership function with specified parameters.

```
mf = fismf("trapmf",[10 15 20 25]);
```

Create Membership Function with Specified Name

Create a membership function with the name "large".

```
mf = fismf("Name","large");
```

See Also

[fisrule](#) | [fisvar](#) | [mamfis](#) | [sugfis](#)

Topics

"Build Fuzzy Systems at the Command Line" on page 2-38

Introduced in R2018b

fisrule

Fuzzy rule

Description

Use `fisrule` objects to represent fuzzy if-then rules that relate input membership function conditions to corresponding output membership functions. The *if* portion of a fuzzy rule is the *antecedent*, which specifies the membership function for each input variable. The *then* portion of a fuzzy rule is the *consequent*, which specifies the membership function for each output variable. For more information on membership functions and fuzzy rules, see “Foundations of Fuzzy Logic” on page 1-10.

Creation

To create fuzzy rule objects, use the `fisrule` function (described here). Using this function, you can create a single fuzzy rule or a vector of multiple fuzzy rules.

Syntax

```
rule = fisrule
rule = fisrule(ruleText)
rule = fisrule(ruleValues,numInputs)
```

Description

`rule = fisrule` creates a single fuzzy rule with the default description "input1==mf1 => output1=mf1".

`rule = fisrule(ruleText)` creates one or more fuzzy rules using the text descriptions in `ruleText`.

`rule = fisrule(ruleValues,numInputs)` creates one or more fuzzy rules using the numeric rule values in `ruleValues`. Specify the number of rule input variables using `numInputs`.

Input Arguments

ruleText — Text rule description

string | character vector | string array | character array

Text rule description, specified as one of the following:

- String or character vector specifying a single rule


```
rule = "If service is poor or food is rancid then tip is cheap";
```
- String array, where each element corresponds to a rule. For example:


```
ruleList = ["If service is poor or food is rancid then tip is cheap";
            "If service is good then tip is average";
            "If service is excellent or food is delicious then tip is generous"];
```
- Character array where each row corresponds to a rule. For example:


```
rule1 = 'If service is poor or food is rancid then tip is cheap';
rule2 = 'If service is good then tip is average';
rule3 = 'If service is excellent or food is delicious then tip is generous';
ruleList = char(rule1,rule2,rule3);
```

For each rule, use one of the following rule text formats:

- Verbose — Linguistic expression in the following format, using the IF and THEN keywords:

```
"IF <antecedent> THEN <consequent> (<weight>)"
```

In <antecedent>, specify the membership function for each input variable using the IS or IS NOT keyword. Connect these conditions using the AND or OR keywords. If a rule does not use a given input variable, omit it from the antecedent.

In <consequent>, specify the condition for each output variable using the IS or IS NOT keyword, and separate these conditions using commas. The IS NOT keyword is not supported for Sugeno outputs. If a rule does not use a given output variable, omit it from the consequent.

Specify the weight using a positive numerical value.

For example:

```
"IF A IS a AND B IS NOT b THEN X IS x, Y IS NOT y (1)"
```

- Symbolic — Expression that uses the symbols in the following table instead of keywords. There is no symbol for the IF keyword.

Symbol	Keyword
==	IS (in rule antecedent)
~=	IS NOT
&	AND
	OR
=>	THEN
=	IS (in rule consequent)

For example, the following symbolic rule is equivalent to the previous verbose rule.

```
"A==a & B~=b => X=x, Y~=y (1)"
```

When you specify a rule using a text description, `fisrule` sets the `Description`, `Weight`, and `Connection` properties of the rule based on the description.

ruleValues — Numeric rule description

row vector | numeric array

Numeric rule description, specified as one of the following:

- Row vector to specify a single fuzzy rule
- Array, where each row of `ruleValues` specifies one rule

For each row, the numeric rule description has $M+N+2$ columns, where M is the number of input variables and N is the number of output variables. Each column contains the following information:

- The first M columns specify input membership function indices and correspond to the `Antecedent` property of the rule. To indicate a NOT condition, specify a negative value. If a rule does not use a given input, set the corresponding index to \emptyset . For each rule, at least one input membership function index must be nonzero.
- The next N columns specify output membership function indices and correspond to the `Consequent` property of the rule. To indicate a NOT condition for Mamdani systems, specify a negative value. NOT conditions are not supported for Sugeno outputs. If a rule does not use a given output, set the corresponding index to \emptyset . For each rule, at least one output membership function index must be nonzero.
- Column $M+N+1$ specifies the rule weight and corresponds to the `Weight` property of the rule.

- The final column specifies the antecedent fuzzy operator and corresponds to the `Connection` property of the rule.

When you specify a rule using `ruleValues`, `fisrule` sets the `Description` property using default variable and membership function names.

numInputs — Number of input variables

positive integer

Number of input variables for the rule, specified as a positive integer. If you specify the rule description using `ruleValues`, you must also specify the number of input variables. `fisrule` parses the rule antecedent values into the membership function indices for the input and output variables using `numInputs`.

Properties

Description — Text rule description

string | character vector

Text rule description, specified as a string or character vector. The rule description is stored as a symbolic expression no matter how you specify the rule. For example, if you specify the following verbose rule using `ruleText`:

```
"IF A IS a AND B IS NOT b THEN X IS x, Y IS NOT y (1)"
```

The stored rule is:

```
"A==a & B~=b => X=x, Y~=y (1)"
```

For more information on the verbose and symbolic rule formats, see the `ruleText` input argument.

When you specify a rule using `ruleValues`, `fisrule` sets the `Description` property using default variable and membership function names. Before using the rule in a fuzzy system, you must update the description to use the variable and membership function names from that fuzzy system using the `update` function.

Antecedent — Rule antecedent

numeric vector

Rule antecedent, specified as a numeric vector of length M , where M is the number of input variables. Each element of `Antecedent` contains one of the following values:

- Positive integer — The index of an input membership function, which represents an IS condition.
- Negative integer — The negative of an input membership function, which represents an IS NOT condition.
- 0 — A *don't care* condition, which means that the rule does not use the corresponding input variable.

This value is set when you create a fuzzy rule using `ruleValues`. If you create a fuzzy rule using `ruleText`, before using the rule in a fuzzy system, you must populate the Antecedent property using the update function.

If you update the indices in the rule antecedent using dot notation, the `Description` property is not updated to reflect the changes. To update the rule description, use the update function.

Consequent — Rule consequent

numeric vector

Rule consequent, specified as a numeric vector of length N , where N is the number of output variables. Each element of `Consequent` contains one of the following values:

- Positive integer — The index of an output membership function, which represents an IS condition.
- Negative integer — The negative of an output membership function, which represents an IS NOT condition.
- 0 — A *don't care* condition, which means that the rule does not use the corresponding output variable.

This value is set when you create a fuzzy rule using `ruleValues`. If you create a fuzzy rule using `ruleText`, before using the rule in a fuzzy system, you must populate the Consequent property using the update function.

If you update the indices in the rule consequent using dot notation, the `Description` property is not updated to reflect the changes. To update the rule description, use the update function.

Weight — Rule weight

1 (default) | positive numeric scalar

Rule weight, specified as a positive numeric scalar in the range [0 1].

If you update the rule weight using dot notation, the weight value in the `Description` property text is also updated.

Connection — Rule antecedent connection

1 | 2

Rule antecedent connection, specified as one of the following:

- 1 — Evaluate rule antecedents using the AND operator.
- 2 — Evaluate rule antecedents using the OR operator.

If you update the rule connection using dot notation, the antecedent operators in the `Description` property text are also updated.

Object Functions

`update` Update fuzzy rule using fuzzy inference system

Examples

Create Fuzzy Rule

Create a default fuzzy rule.

```
rule = fisrule

rule =
    fisrule with properties:

        Description: "input1==mf1 => output1=mf1 (1)"
        Antecedent: 1
        Consequent: 1
        Weight: 1
        Connection: 1
```

To modify the rule properties, use dot notation. For example, specify a rule weight of 0.5.

```
rule.Weight = 0.5;
```

Create Fuzzy Rule Using Text Description

Create a fuzzy rule using a verbose text description.

```
rule = fisrule("if service is poor and food is delicious then tip is average (1)");
```

Alternatively, you can specify the same rule using a symbolic text description.

```
rule = fisrule("service==poor & food==delicious => tip=average")
```

```
rule =  
    fisrule with properties:  
  
    Description: "service==poor & food==delicious => tip=average (1)"  
    Antecedent: []  
    Consequent: []  
    Weight: 1  
    Connection: 1
```

Before using rule with a fuzzy system, update the rule Antecedent and Consequent properties using the update function.

```
fis = readfis("tipper");  
rule = update(rule, fis)
```

```
rule =  
    fisrule with properties:  
  
    Description: "service==poor & food==delicious => tip=average (1)"  
    Antecedent: [1 2]  
    Consequent: 2  
    Weight: 1  
    Connection: 1
```

Create Fuzzy Rule Using Numeric Description

Create a fuzzy rule using a numeric description. Specify that the rule has two input variables.

```
rule = fisrule([1 2 2 0.5 1],2)
rule =
  fisrule with properties:
    Description: "input1==mf1 & input2==mf2 => output1=mf2 (0.5)"
    Antecedent: [1 2]
    Consequent: 2
    Weight: 0.5000
    Connection: 1
```

Before using rule with a fuzzy system, update the rule Description property using the update function.

```
fis = readfis("tipper");
rule = update(rule,fis)
rule =
  fisrule with properties:
    Description: "service==poor & food==delicious => tip=average (0.5)"
    Antecedent: [1 2]
    Consequent: 2
    Weight: 0.5000
    Connection: 1
```

Create Multiple Fuzzy Rules

Create a string array of text rule descriptions.

```
rules1 = ["if service is poor or food is rancid then tip is cheap (0.5)"...
          "if service is excellent and food is not rancid then tip is generous (0.75)"]
```

Create an array of fuzzy rules using these descriptions.

```
fuzzyRules1 = fisrule(rules1)
fuzzyRules1 =
  1x2 fisrule array with properties:
    Description
```

Antecedent
Consequent
Weight
Connection

Details:

Description

```
1 "service==poor | food==rancid => tip=cheap (0.5)"
2 "service==excellent & food~=rancid => tip=generous (0.75)"
```

Alternatively, you can specify multiple rules using an array of numeric rule descriptions.

```
rules2 = [1 1 1 0.5 2;
          2 -1 3 0.75 1];
fuzzyRules2 = fisrule(rules2,2)
```

```
fuzzyRules2 =
  1x2 fisrule array with properties:
```

Description
Antecedent
Consequent
Weight
Connection

Details:

Description

```
1 "input1==mf1 | input2==mf1 => output1=mf1 (0.5)"
2 "input1==mf2 & input2~=mf1 => output1=mf3 (0.75)"
```

See Also

fismf | fisvar | mamfis | sugfis

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

fisvar

Fuzzy variable

Description

Use `fisvar` objects to represent the input and output variables in a fuzzy inference system (FIS). For more information on creating fuzzy inference systems, see `mamfis` and `sugfis`.

Creation

Syntax

```
var = fisvar
var = fisvar(range)
var = fisvar('Name',name)
var = fisvar(range,'Name',name)
```

Description

`var = fisvar` creates a fuzzy variable with a default name, default range, and no membership functions. To change the variable properties, use dot notation.

`var = fisvar(range)` sets the Range property.

`var = fisvar('Name',name)` sets the Name property.

`var = fisvar(range,'Name',name)` sets both the Range and Name properties.

Properties

Name — Variable name

"var" (default) | string | character vector

Variable name, specified as a string or character vector.

Range — Variable range

[0 1] (default) | two-element vector

Variable range, specified as a two-element element vector where the first element is less than the second element. The first element specifies the lower bound of the range, and the second element specifies the upper bound of the range.

MembershipFunctions — Membership functions

[] (default) | vector of `fismf` objects

Membership functions, specified as a vector of `fismf` objects. To add membership functions to a fuzzy variable:

- Use the `addMF` function.
- Create a vector of `fismf` objects and assign it to `MembershipFunctions`.

You can modify the properties of the membership functions using dot notation.

Object Functions

`addMF` Add membership function to fuzzy variable

`removeMF` Remove membership function from fuzzy variable

Examples

Create Fuzzy Variable

Create a fuzzy variable with default properties.

```
var = fisvar;
```

To modify the properties of a `fisvar` object, use dot notation. For example, specify the range of the fuzzy variable to be from -5 to 5.

```
var.Range = [-5 5];
```

Create Fuzzy Variable with Specified Range

Create a fuzzy variable with an input range from -10 to 10.

```
var = fisvar([-10 10]);
```

Create Fuzzy Variable with Specified Name

Create a fuzzy variable with the name "speed".

```
var = fisvar("Name", "speed");
```

See Also

fismf | fisrule | mamfis | sugfis

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

mamfis

Mamdani fuzzy inference system

Description

Use a `mamfis` object to represent a Mamdani fuzzy inference system (FIS). For more information on Mamdani systems, see “What Is Mamdani-Type Fuzzy Inference?” on page 2-4.

As an alternative to Mamdani systems, you can create a Sugeno FIS using a `sugfis` object. For a comparison of Mamdani and Sugeno systems, see “Comparison of Sugeno and Mamdani Systems” on page 2-12.

Creation

To create a Mamdani FIS object, use one of the following methods:

- The `mamfis` function (described here).
- If you have input/output training data, you can use the `genfis` function with the FCM clustering method.

```
opt = genfisOptions('FCMClustering','FISType','mamdani');  
fis = genfis(inputDate,outputData,opt);
```

- If you have a `.fis` file for a Mamdani system, you can use the `readfis` function.

Syntax

```
fis = mamfis  
fis = mamfis(Name,Value)
```

Description

`fis = mamfis` creates a Mamdani FIS with default property values. To modify the properties of the fuzzy system, use dot notation.

`fis = mamfis(Name, Value)` specifies FIS configuration information or sets object properties using name-value pair arguments. You can specify multiple name-value pairs. Enclose names in quotes.

Input Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'NumInputs', 2` configures the fuzzy system to have two input variables

NumInputs — Number of FIS inputs

0 (default) | nonnegative integer

Number of FIS inputs, specified as the comma-separated pair consisting of `'NumInputs'` and a nonnegative integer.

NumInputMFs — Number of membership functions for each FIS input

3 (default) | positive integer

Number of membership functions for each FIS input, specified as the comma-separated pair consisting of `'NumInputMFs'` and a positive integer. To specify the type of membership

NumOutputs — Number of FIS outputs

0 (default) | nonnegative integer

Number of FIS outputs, specified as the comma-separated pair consisting of `'NumOutputs'` and a nonnegative integer.

NumOutputMFs — Number of membership functions for each FIS output

3 (default) | positive integer

Number of membership functions for each FIS output, specified as the comma-separated pair consisting of `'NumOutputMFs'` and a positive integer.

MFType — Membership function type

`"trimf"` (default) | `"gaussmf"`

Membership function type for both input and output variables, specified as the comma-separated pair consisting of "MFType" and either "trimf" (triangular MF) or "gaussmf" (Gaussian MF). For each input and output variable, the membership functions are uniformly distributed over the variable range with approximately 80% overlap in the MF supports.

AddRules — Flag for automatically adding rules

"allcombinations" (default) | "none"

Flag for automatically adding rules, specified as the comma-separated pair consisting of "AddRules" and one of the following:

- "allcombinations" — If both NumInputs and NumOutputs are greater than zero, create rules with antecedents that contain all input membership function combinations. Each rule consequent contains all the output variables and uses the first membership function of each output.
- "none" — Create a FIS without any rules.

Properties

Name — FIS name

"fis" (default) | string | character vector

FIS name, specified as a string or character vector.

AndMethod — AND operator method

"min" (default) | "prod" | string | character vector | function handle

AND operator method for combining fuzzified input values in a fuzzy rule antecedent, specified as one of the following:

- "min" — Minimum of fuzzified input values
- "prod" — Product of fuzzified input values
- String or character vector — Name of a custom AND function in the current working folder or on the MATLAB path
- Function handle — Custom AND function in the current working folder or on the MATLAB path

For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on fuzzy operators and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

OrMethod — OR operator method

"max" (default) | "probor" | string | character vector | function handle

OR operator method for combining fuzzified input values in a fuzzy rule antecedent, specified as one of the following:

- "max" — Maximum of fuzzified input values
- "probor" — Probabilistic OR of fuzzified input values. For more information, see `probor`.
- String or character vector — Name of a custom OR function in the current working folder or on the MATLAB path
- Function handle — Custom OR function in the current working folder or on the MATLAB path

For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on fuzzy operators and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

ImplicationMethod — Implication method

"min" (default) | "prod" | string | character vector | function handle

Implication method for computing the consequent fuzzy set, specified as one of the following:

- "min" — Truncate the consequent membership function at the antecedent result value
- "prod" — Scale the consequent membership function by the antecedent result value
- String or character vector — Name of a custom implication function in the current working folder or on the MATLAB path
- Function handle — Custom implication function in the current working folder or on the MATLAB path

For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on implication and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

AggregationMethod – Aggregation method

"max" (default) | "sum" | "probor" | string | character vector | function handle

Aggregation method for combining rule consequents, specified as one of the following:

- "max" — Maximum of consequent fuzzy sets
- "sum" — Sum of consequent fuzzy sets
- "probor" — Probabilistic OR of consequent fuzzy sets. For more information, see `probor`.
- String or character vector — Name of a custom aggregation function in the current working folder or on the MATLAB path
- Function handle — Custom aggregation function in the current working folder or on the MATLAB path

For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on aggregation and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

DefuzzificationMethod – Defuzzification method

"centroid" (default) | "bisector" | "mom" | "lom" | "som" | string | character vector | function handle

Defuzzification method for computing crisp output values from the aggregated output fuzzy set, specified as one of the following:

- "centroid" — Centroid of the area under the output fuzzy set
- "bisector" — Bisector of the area under the output fuzzy set
- "mom" — Mean of the values for which the output fuzzy set is maximum
- "lom" — Largest value for which the output fuzzy set is maximum
- "som" — Smallest value for which the output fuzzy set is maximum
- String or character vector — Name of a custom defuzzification function in the current working folder or on the MATLAB path
- Function handle — Custom defuzzification function in the current working folder or on the MATLAB path

For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on defuzzification and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

Inputs — FIS input variables

vector of `fisvar` objects

FIS input variables, specified as a vector of `fisvar` objects. To add and remove input variables, use `addInput` and `removeInput`, respectively.

You can also create a vector of `fisvar` objects and assign it to `Inputs` using dot notation.

You can add membership functions to input variables using the `addMF` function.

Outputs — FIS output variables

vector of `fisvar` objects

FIS output variables, specified as a vector of `fisvar` objects. To add and remove output variables, use `addOutput` and `removeOutput`, respectively.

You can also create a vector of `fisvar` objects and assign it to `Outputs` using dot notation.

You can add membership functions to output variables using the `addMF` function.

Rules — FIS rules

vector of `fisrule` objects

FIS input variables, specified as a vector of `fisrule` objects. To add fuzzy rules, use the `addRule` function.

You can also create a vector of `fisrule` objects and assign it to `Rules` using dot notation.

To remove a rule, set the corresponding rule vector element to `[]`. For example, to remove the tenth rule from the rule list, type:

```
fis.Rules(10) = [];
```

DisableStructuralChecks — Flag for disabling consistency checks

false (default) | true

Flag for disabling consistency checks when property values change, specified as a logical value.

By default, when you change the value of a property of a `mamfis` object, the software verifies whether the new property value is consistent with the other object properties. These checks can affect performance, particularly when creating and updating fuzzy systems within loops.

To disable these checks, which results in faster FIS construction, set `DisableStructuralChecks` to `true`.

Note Disabling structural checks can result in an invalid `mamfis` object.

To reenable the consistency checks, first verify that the changes you made to the FIS are consistent and produce a valid `mamfis` object. Then, set `DisableStructuralChecks` to `false`. If the `mamfis` object is invalid, reenabling the consistency checks generates an error.

Object Functions

<code>addInput</code>	Add input variable to fuzzy inference system
<code>removeInput</code>	Remove input variable from fuzzy inference system
<code>addOutput</code>	Add output variable to fuzzy inference system
<code>removeOutput</code>	Remove output variable from fuzzy inference system
<code>addRule</code>	Add rule to fuzzy inference system
<code>addMF</code>	Add membership function to fuzzy variable
<code>removeMF</code>	Remove membership function from fuzzy variable
<code>evalfis</code>	Evaluate fuzzy inference system
<code>writeFIS</code>	Save fuzzy inference system to file

Examples

Create Mamdani Fuzzy Inference System

Create a Mamdani fuzzy inference system with default property values.

```
fis = mamfis;
```

Modify the system properties using dot notation. For example, configure `fis` to use centroid defuzzification.

```
fis.DefuzzificationMethod = "centroid";
```

Alternatively, you can specify one of more FIS properties when you create a fuzzy system. For example, create a Mamdani fuzzy system with specified AND and OR methods.

```
fis = mamfis("AndMethod", "prod", "OrMethod", "probor");
```

Specify Number of Inputs and Outputs for Mamdani System

Create a Mamdani fuzzy inference system with three inputs and one output.

```
fis = mamfis("NumInputs",3,"NumOutputs",1);
```

Alternative Functionality

App

You can interactively create a Mamdani FIS using the **Fuzzy Logic Designer** app. You can then export the system to the MATLAB workspace.

See Also

`fismf` | `fisrule` | `fisvar` | `sugfis`

Topics

“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

sugfis

Sugeno fuzzy inference system

Description

Use a `sugfis` object to represent a Sugeno fuzzy inference system (FIS). For more information on Sugeno systems, see “What Is Sugeno-Type Fuzzy Inference?” on page 2-5.

As an alternative to Sugeno systems, you can create a Mamdani FIS using a `mamfis` object. For a comparison of Sugeno and Mamdani systems, see “Comparison of Sugeno and Mamdani Systems” on page 2-12.

Creation

To create a Sugeno FIS object, use one of the following methods:

- The `sugfis` function (described here).
- If you have input/output data, you can use the `genfis` function.
- If you have a `.fis` file for a Sugeno system, you can use the `readfis` function.
- Convert an existing Mamdani FIS to a Sugeno FIS using `convertToSugeno`.

Syntax

```
fis = sugfis  
fis = sugfis(Name,Value)
```

Description

`fis = sugfis` creates a Sugeno FIS with default property values. To modify the properties of the fuzzy system, use dot notation.

`fis = sugfis(Name, Value)` specifies FIS configuration information or sets object properties using name-value pair arguments. You can specify multiple name-value pairs. Enclose names in quotes.

Input Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'NumInputs', 2` configures the fuzzy system to have two input variables

NumInputs — Number of FIS inputs

0 (default) | nonnegative integer

Number of FIS inputs, specified as the comma-separated pair consisting of `'NumInputs'` and a nonnegative integer.

NumInputMFs — Number of membership functions for each FIS input

3 (default) | positive integer

Number of membership functions for each FIS input, specified as the comma-separated pair consisting of `'NumInputMFs'` and a positive integer. To specify the type of membership

NumOutputs — Number of FIS outputs

0 (default) | nonnegative integer

Number of FIS outputs, specified as the comma-separated pair consisting of `'NumOutputs'` and a nonnegative integer.

NumOutputMFs — Number of membership functions for each FIS output

3 (default) | positive integer

Number of membership functions for each FIS output, specified as the comma-separated pair consisting of `'NumOutputMFs'` and a positive integer.

MFType — Membership function type

"trimf" (default) | "gaussmf"

Membership function type for input variables, specified as the comma-separated pair consisting of 'MFType' and either "trimf" (triangular MF) or "gaussmf" (Gaussian MF). For each input variable, the membership functions are uniformly distributed over the variable range with approximately 80% overlap in the MF supports.

Output membership functions are set to "constant" and uniformly distributed over the output variable ranges.

AddRules — Flag for automatically adding rules

"allcombinations" (default) | "none"

Flag for automatically adding rules, specified as the comma-separated pair consisting of "AddRules" and one of the following:

- "allcombinations" — If both NumInputs and NumOutputs are greater than zero, create rules with antecedents that contain all input membership function combinations. Each rule consequent contains all the output variables and uses the first membership function of each output.
- "none" — Create a FIS without any rules.

Properties

Name — FIS name

"fis" (default) | string | character vector

FIS name, specified as a string or character vector.

AndMethod — AND operator method

"prod" (default) | "min" | string | character vector | function handle

AND operator method for combining fuzzified input values in a fuzzy rule antecedent, specified as one of the following:

- "prod" — Product of fuzzified input values
- "min" — Minimum of fuzzified input values
- String or character vector — Name of a custom AND function in the current working folder or on the MATLAB path
- Function handle — Custom AND function in the current working folder or on the MATLAB path

For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on fuzzy operators and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

OrMethod — OR operator method

"probor" (default) | "max" | string | character vector | function handle

OR operator method for combining fuzzified input values in a fuzzy rule antecedent, specified as one of the following:

- "probor" — Probabilistic OR of fuzzified input values. For more information, see `probor`.
- "max" — Maximum of fuzzified input values
- String or character vector — Name of a custom OR function in the current working folder or on the MATLAB path
- Function handle — Custom OR function in the current working folder or on the MATLAB path

For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on fuzzy operators and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

ImplicationMethod — Implication method

"prod" (default)

Implication method for computing consequent fuzzy set, specified as "prod". Sugeno systems always use the "prod" implication method, which scales the consequent membership function by the antecedent result value.

For more information on implication and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

AggregationMethod — Aggregation method

"sum" (default)

Aggregation method for combining rule consequents, specified as "sum". Sugeno systems always use the "sum" aggregation method, which is the sum of the consequent fuzzy sets.

For more information on aggregation and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

DefuzzificationMethod — Defuzzification method

"wtaver" (default) | "wtsum"

Defuzzification method for computing crisp output values from the aggregated output fuzzy set, specified as one of the following:

- "wtaver" — Weighted average of all rule outputs
- "wtsum" — Weighted sum of all rule outputs

For more information on using custom functions, see “Build Fuzzy Systems Using Custom Functions” on page 2-50.

For more information on defuzzification and the fuzzy inference process, see “Fuzzy Inference Process” on page 1-28.

Inputs — FIS input variables

vector of `fisvar` objects

FIS input variables, specified as a vector of `fisvar` objects. To add and remove input variables, use `addInput` and `removeInput`, respectively. You can modify the properties of the input variables using dot notation.

You can also create a vector of `fisvar` objects and assign it to `Inputs` using dot notation.

You can add membership functions to input variables using the `addMF` function.

Outputs — FIS output variables

vector of `fisvar` objects

FIS output variables, specified as a vector of `fisvar` objects. To add and remove output variables, use `addOutput` and `removeOutput`, respectively.

You can also create a vector of `fisvar` objects and assign it to `Outputs` using dot notation.

You can add membership functions to output variables using the `addMF` function.

Rules — FIS rules

vector of `fisrule` objects

FIS input variables, specified as a vector of `fisrule` objects. To add fuzzy rules, use the `addRule` function.

You can also create a vector of `fisrule` objects and assign it to `Rules` using dot notation.

To remove a rule, set the corresponding rule vector element to `[]`. For example, to remove the tenth rule from the rule list, type:

```
fis.Rules(10) = [];
```

DisableStructuralChecks — Flag for disabling consistency checks

false (default) | true

Flag for disabling consistency checks when property values change, specified as a logical value.

By default, when you change the value of a property of a `sugfis` object, the software verifies whether the new property value is consistent with the other object properties. These checks can affect performance, particularly when creating and updating fuzzy systems within loops.

To disable these checks, which results in faster FIS construction, set `DisableStructuralChecks` to `true`.

Note Disabling structural checks can result in an invalid `sugfis` object.

To reenble the consistency checks, first verify that the changes you made to the FIS are consistent and produce a valid `sugfis` object. Then, set `DisableStructuralChecks` to `false`. If the `sugfis` object is invalid, reenabling the consistency checks generates an error.

Object Functions

<code>addInput</code>	Add input variable to fuzzy inference system
<code>removeInput</code>	Remove input variable from fuzzy inference system
<code>addOutput</code>	Add output variable to fuzzy inference system
<code>removeOutput</code>	Remove output variable from fuzzy inference system
<code>addRule</code>	Add rule to fuzzy inference system

addMF	Add membership function to fuzzy variable
removeMF	Remove membership function from fuzzy variable
evalfis	Evaluate fuzzy inference system
writeFIS	Save fuzzy inference system to file

Examples

Create Sugeno Fuzzy Inference System

Create a Sugeno fuzzy inference system with default property values.

```
fis = sugfis;
```

Modify the system properties using dot notation. For example, configure `fis` to use weighted-sum defuzzification.

```
fis.DefuzzificationMethod = "wtsum";
```

Alternatively, you can specify one or more FIS properties when you create a fuzzy system. For example, create a Sugeno fuzzy system with specified AND and OR methods.

```
fis = sugfis("AndMethod","min","OrMethod","max");
```

Specify Number of Inputs and Outputs for Sugeno System

Create a Sugeno fuzzy inference system with three inputs and one output.

```
fis = sugfis("NumInputs",3,"NumOutputs",1);
```

Alternative Functionality

App

You can interactively create a Sugeno FIS using the **Fuzzy Logic Designer** or **Neuro-Fuzzy Designer** apps. You can then export the system to the MATLAB workspace.

See Also

fismf | fisrule | fisvar | mamfis

Topics

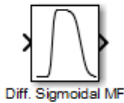
“Build Fuzzy Systems at the Command Line” on page 2-38

Introduced in R2018b

Blocks — Alphabetical List

Diff. Sigmoidal MF

Difference of two sigmoids membership function in Simulink software



Description

The Diff. Sigmoidal MF block implements a membership function in Simulink based on the difference between two sigmoids. The two sigmoid curves are given by

$$f_k(x) = \frac{1}{1 + \exp(-a_k(x - c_k))}$$

where $k=1,2$. The parameters a_1 and a_2 control the slopes of the left and right curves. The parameters c_1 and c_2 control the points of inflection for the left and right curves. The parameters a_1 and a_2 should be positive.

See Also

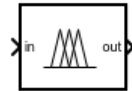
`dsigmf`

Introduced before R2006a

Fuzzy Logic Controller

Evaluate fuzzy inference system

Library: Fuzzy Logic Toolbox



Description

The Fuzzy Logic Controller block implements a fuzzy inference system (FIS) in Simulink. You specify the FIS to evaluate using the **FIS name** parameter.

For more information on fuzzy inference, see “Fuzzy Inference Process” on page 1-28.

To display the fuzzy inference process in the Rule Viewer during simulation, use the Fuzzy Logic Controller with Ruleviewer block.

Ports

Input

in — Input signal

scalar | vector

For a single-input fuzzy inference system, the input is a scalar signal. For a multi-input fuzzy system, combine the inputs into a vector signal using blocks such as:

- Mux
- Vector Concatenate
- Bus Creator

Output

out — Defuzzified output signal

scalar | vector

For a single-output fuzzy inference system, the output is a scalar signal. For a multi-output fuzzy system, the output is a vector signal. To split system outputs into scalar signals, use the Demux block.

fi — Fuzzified input values

matrix

Fuzzified input values, obtained by evaluating the input membership functions of each rule at the current input values.

fi is an N_R -by- N_U matrix signal, where N_R is the number of FIS rules and N_U is the number of FIS inputs. Element (i,j) of **fi** is the value of the input membership function for the j th input in the i th rule.

For more information on fuzzifying input values, see “Fuzzify Inputs” on page 1-29.

Dependencies

To enable this port, select the **Fuzzified inputs (fi)** parameter.

rfs — Rule firing strengths

column vector

Rule firing strengths, obtained by evaluating the antecedent of each rule; that is, applying the fuzzy operator to the values of the fuzzified inputs.

rfs is a column vector signal of length N_R , where element i is the firing strength of the i th rule.

For more information on applying fuzzy operators, see “Apply Fuzzy Operator” on page 1-30.

Dependencies

To enable this port, select the **Rule firing strengths (rfs)** parameter.

ro — Rule outputs

matrix

Rule outputs, obtained by applying the rule firing strengths to the output membership functions using the implication method specified in the FIS.

For a Mamdani system, each rule output is a fuzzy set. In this case, **ro** is an N_S -by- $(N_R N_Y)$ matrix signal. Here N_S is the number of sample points used for evaluating output variable ranges, and N_Y is the number of output variables. Each column of **ro** contains the output fuzzy set for one rule. The first N_R columns contain the rule outputs for the first output variable, the next N_R columns correspond to the second output variable, and so on.

For a Sugeno system, each rule output is a scalar value. In this case, **ro** is an N_R -by- N_Y matrix signal. Element (j,k) of **ro** is the value of the k th output variable for the j th rule.

For more information on fuzzy implication, see “Apply Implication Method” on page 1-31 and “What Is Sugeno-Type Fuzzy Inference?” on page 2-5

Dependencies

- To enable this port, select the **Rule outputs (ro)** parameter.
- To specify N_S , use the **Number of samples for output discretization** parameter.

ao — Aggregated output

matrix | row vector

Aggregate output for each output variable, obtained by combining the corresponding outputs from all the rules using the aggregation method specified in the FIS.

For a Mamdani system, the aggregate result for each output variable is a fuzzy set. In this case, **ao** is as an N_S -by- N_Y matrix signal. Each column of **ao** contains the aggregate fuzzy set for one output variable.

For a Sugeno system, the aggregate result for each output variable is a scalar value. In this case, **ao** is a row vector signal of length N_Y , where element k is the aggregate result for the k th output variable.

For more information on fuzzy aggregation, see “Aggregate All Outputs” on page 1-31 and “What Is Sugeno-Type Fuzzy Inference?” on page 2-5

Dependencies

- To enable this port, select the **Aggregated outputs (ao)** parameter.
- To specify N_S , use the **Number of samples for output discretization** parameter.

Parameters

General

FIS name — Fuzzy inference system

`mamfisobject` | `sugfisobject` | file name

Fuzzy inference system to evaluate, specified as one of the following:

- `mamfis` or `sugfis` object — Specify the name of a FIS object in the MATLAB workspace.

To create a fuzzy inference system, you can:

- Use the **Fuzzy Logic Designer** app. For an example, see “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14.
- Automatically generate the FIS object using the `genfis` command.
- Build the system at the command line. For an example, see “Build Fuzzy Systems at the Command Line” on page 2-38.
- File name — Specify the name of a `.fis` file in the current working folder or on the MATLAB path. Including the file extension in the file name is optional.

To save a fuzzy inference system to a `.fis` file:

- In **Fuzzy Logic Designer** or **Neuro-Fuzzy Designer**, select **File > Export > To File**.
- At the command line, use `writeFIS`.

Number of samples for output discretization — Number of points in output fuzzy sets

101 (default) | integer greater than 1

Number of samples for discretizing the range of output variables, specified as an integer greater than 1. This value corresponds to the number of points in the output fuzzy set for each rule.

To reduce memory usage while evaluating a Mamdani FIS, specify a lower number of samples. Doing so sacrifices the accuracy of the defuzzified output value. Specifying a low number of samples can make the output area for defuzzification zero. In this case, the defuzzified output value is the midpoint of the output variable range.

Note The block ignores this parameter when evaluating a Sugeno FIS.

Data type — Signal data type

double (default) | single | fixed-point | expression

Signal data type, specified as one of the following:

- `double` — Double-precision signals
- `single` — Single-precision signals
- `fixdt(1,16,0)` — Fixed-point signals with binary point scaling
- `fixdt(1,16,2^0,0)` — Fixed-point signals with slope and bias scaling
- `Expression` — Expression that evaluates to one of these data types

For fixed-point data types, you can configure the signedness, word length, and scaling parameters using the **Data Type Assistant**. For more information, see “Specifying a Fixed-Point Data Type” (Simulink).

Fuzzified inputs (fi) — Enable fi output port

off (default) | on

Enable output port for accessing intermediate fuzzified input data.

Rule firing strengths (rfs) — Enable rfs output port

off (default) | on

Enable output port for accessing intermediate rule firing strength data.

Rule outputs (ro) — Enable ro output port

off (default) | on

Enable output port for accessing intermediate rule output data.

Aggregated outputs (ao) — Enable ao output port

off (default) | on

Enable output port for accessing intermediate aggregate output data.

Simulate using — Simulation mode

Interpreted execution (default) | Code generation

Simulation mode, specified as one of the following:

- **Interpreted execution** — Simulate fuzzy systems using precompiled MEX files for single and double data types. Using this option reduces the initial compilation time of the model.
- **Code generation** — Simulate fuzzy system without precompiled MEX files. Use this option when simulating fuzzy systems for code generation applications.

For fixed-point data types, the Fuzzy Logic Controller block always simulates using Code generation mode.

Diagnostics

Out of range input value — Diagnostic message behavior when an input is out of range

warning (default) | error | none

Diagnostic message behavior when an input is out of range, specified as one of the following:

- **warning** — Report the diagnostic message as a warning.
- **error** — Report the diagnostic message as an error.
- **none** — Do not report the diagnostic message.

When an input value is out of range, corresponding rules in the fuzzy system can have unexpected firing strengths.

Dependencies

- Diagnostic messages are provided only when the **Simulate using** parameter is **Interpreted execution**.

No rule fired — Diagnostic message behavior when no rules fire

warning (default) | error | none

Diagnostic message behavior when no rules fire for a given output variable, specified as one of the following:

- **warning** — Report the diagnostic message as a warning.
- **error** — Report the diagnostic message as an error.
- **none** — Do not report the diagnostic message.

When **No rule fired** is warning or none and no rules fire for a given output, the defuzzified output value is set to its mean range value.

Dependencies

- Diagnostic messages are provided only when the **Simulate using** parameter is Interpreted execution.

Empty output fuzzy set – Diagnostic message behavior when an output fuzzy set is empty

warning (default) | error | none

Diagnostic message behavior when an output fuzzy set is empty, specified as one of the following:

- warning — Report the diagnostic message as a warning.
- error — Report the diagnostic message as an error.
- none — Do not report the diagnostic message.

When **Empty output fuzzy set** is warning or none and an output fuzzy set is empty, the defuzzified value for the corresponding output is set to its mean range value.

Dependencies

- This diagnostic message applies to Mamdani systems only.
- Diagnostic messages are provided only when the **Simulate using** parameter is Interpreted execution.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.

- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Convert floating-point algorithms to fixed point using Fixed-Point Designer™.

See Also

Blocks

Fuzzy Logic Controller with Ruleviewer

Apps

Fuzzy Logic Designer | **Neuro-Fuzzy Designer**

Functions

`evalfis` | `genfis` | `mamfis` | `readfis` | `sugfis` | `writefis`

Topics

“Fuzzy Inference Process” on page 1-28

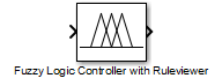
“Simulate Fuzzy Inference Systems in Simulink” on page 5-2

Introduced before R2006a

Fuzzy Logic Controller with Ruleviewer

Evaluate fuzzy inference system and view rules

Library: Fuzzy Logic Toolbox



Description

The Fuzzy Logic Controller with Ruleviewer block implements a fuzzy inference system (FIS) in Simulink and displays the fuzzy inference process in the Rule Viewer during the simulation. You specify the FIS to evaluate using the **FIS matrix** parameter. To change the time between Rule Viewer updates, specify the **Refresh rate** in seconds.

For more information on fuzzy inference, see “Fuzzy Inference Process” on page 1-28.

The Fuzzy Logic Controller with Ruleviewer block does not support all the features supported by the Fuzzy Logic Controller block. The Fuzzy Logic Controller with Ruleviewer block:

- Only supports double-precision data.
- Uses 101 points for discretizing output variable ranges.
- Only supports `Interpreted execution` simulation mode.
- Does not have additional output ports for accessing intermediate fuzzy inference results.

Ports

Input

Port_1(In1) — Input signal

scalar | vector

For a single-input fuzzy inference system, the input is a scalar. For a multi-input fuzzy system, combine the inputs into a vector signal using blocks such as:

- Mux
- Vector Concatenate
- Bus Creator

Output

Port_1(Out1) — Defuzzified output signal

scalar | vector

For a single-output fuzzy inference system, the output is a scalar. For a multi-output fuzzy system, the output is a vector. To split system outputs into scalar signals, use the Demux block.

Parameters

FIS matrix — Fuzzy inference system

mamfisobject | sugfisobject

Fuzzy inference system to evaluate, specified as a `mamfis` or `sugfis` object. Specify the name of a FIS object in the MATLAB workspace.

To create a fuzzy inference system, you can:

- Use the **Fuzzy Logic Designer** app. For an example, see “Build Fuzzy Systems Using Fuzzy Logic Designer” on page 2-14.
- Automatically generate the FIS object using the `genfis` command.
- Build the system at the command line. For an example, see “Build Fuzzy Systems at the Command Line” on page 2-38.

Refresh rate — Time between rule viewer updates

scalar

Time between rule viewer updates in seconds, specified as a scalar. During simulation, the Rule Viewer display updates at the specified rate to show the inference process for the latest input signal values.

Compatibility Considerations

Support for representing fuzzy inference systems as structures will be removed

Not recommended starting in R2018b

Support for representing fuzzy inference systems as structures will be removed in a future release. Use `mamfis` and `sugfis` objects instead. There are differences between these representations that require updates to your code. These differences include:

- Object property names that differ from the corresponding structure fields.
- Objects store text data as strings rather than as character vectors.

Also, all Fuzzy Logic Toolbox functions that accepted or returned fuzzy inference systems as structures now accept and return either `mamfis` or `sugfis` objects.

To convert existing fuzzy inference system structures to objects, use the `convertfis` function.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- Generating code using the Fuzzy Logic Controller with Ruleviewer block produces the same code as using the Fuzzy Logic Controller block. However, the Fuzzy Logic Controller with Ruleviewer block does not support:
 - Generating code for single-point or fixed-point data.
 - Changing the number of samples for discretizing the output variable range.

See Also

Blocks

Fuzzy Logic Controller

Apps

Fuzzy Logic Designer | Neuro-Fuzzy Designer

Functions

evalfis | mamfis | readfis | sugfis

Topics

“Fuzzy Inference Process” on page 1-28

“Simulate Fuzzy Inference Systems in Simulink” on page 5-2

Introduced before R2006a

Gaussian MF

Gaussian membership function in Simulink software



Description

The Gaussian MF block implements a membership function in Simulink based on a symmetric Gaussian. The Gaussian curve is given by

$$f(x) = \exp\left(\frac{-0.5(x-c)^2}{\sigma^2}\right)$$

where c is the mean and σ is the variance.

See Also

gaussmf

Introduced before R2006a

Gaussian2 MF

Combination of two Gaussian membership functions in Simulink software



Description

The Gaussian2 MF block implements a membership function based on a combination of two Gaussian functions. The two Gaussian functions are given by

$$f_k(x) = \exp\left(\frac{-0.5(x - c_k)^2}{\sigma_k^2}\right)$$

where $k=1,2$. The parameters c_1 and σ_1 are the mean and variance defining the left-most curve. The parameters c_2 and σ_2 are the mean and variance defining the right-most curve.

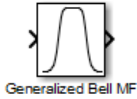
See Also

gauss2mf

Introduced before R2006a

Generalized Bell MF

Generalized bell membership function in Simulink software



Description

The Generalized Bell MF block implements a membership function in Simulink based on a generalized bell-shaped curve. The generalized bell-shaped curve is given by

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

where the parameters a and b vary the width of the curve and the parameter c locates the center of the curve. The parameter b should be positive.

See Also

gbellmf

Introduced before R2006a

Pi-shaped MF

Pi-shaped membership function in Simulink software



Description

The Pi-shaped MF block implements a membership function in Simulink based on a spline-based curve, so named because of its Π shape. The parameters a and d locate the left and right base points or “feet” of the curve. The parameters b and c set the left and right top point or “shoulders” of the curve.

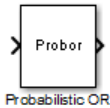
See Also

`pimf`

Introduced before R2006a

Probabilistic OR

Probabilistic OR function in Simulink software



Description

The Probabilistic OR block outputs the probabilistic OR value for the vector signal input, based on

$$y = 1 - \text{prod}(1 - x)$$

See Also

Blocks

Probabilistic Rule Agg

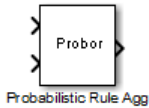
Functions

probor

Introduced before R2006a

Probabilistic Rule Agg

Probabilistic OR function, rule aggregation method



Description

The Probabilistic Rule Agg block outputs the element-wise(*) probabilistic OR value of the two inputs based on

$$y = 1 - \text{prod}(1 - [a; b])$$

The two inputs, a and b , are row vectors.

See Also

Blocks

Probabilistic OR

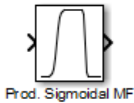
Functions

probor

Introduced before R2006a

Prod. Sigmoidal MF

Product of two sigmoid membership functions in Simulink software



Description

The Prod. Sigmoidal MF block implements a membership function based on the product of two sigmoidal curves. The two sigmoidal curves are given by

$$f_k(x) = \frac{1}{1 + \exp(-a_k(x - c_k))}$$

where $k=1,2$ The parameters a_1 and a_2 control the slopes of the left and right curves. The parameters c_1 and c_2 control the points of inflection for the left and right curves. Parameters a_1 and a_2 should be positive and negative respectively.

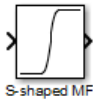
See Also

psigmf

Introduced before R2006a

S-shaped MF

S-shaped membership function in Simulink software



Description

The S-shaped MF block implements an S-shaped membership function in Simulink. Going from left to right the function increases from 0 to 1. The parameters a and b locate the left and right extremes of the sloped portion of the curve.

See Also

smf

Introduced before R2006a

Sigmoidal MF

Sigmoidal membership function in Simulink software



Description

The Sigmoidal MF block implements a sigmoidal membership function given by

$$f(x) = \frac{1}{1 + \exp(-a(x - c))}$$

When the sign of a is positive the curve increases from left to right. Conversely, when the sign of a is negative the curve decreases from left to right. The parameter c sets the point of inflection of the curve.

See Also

`sigmf`

Introduced before R2006a

Trapezoidal MF

Trapezoidal membership function in Simulink software



Description

The Trapezoidal MF block implements a trapezoidal-shaped membership function. The parameters a and d set the left and right “feet,” or base points, of the trapezoid. The parameters b and c set the “shoulders,” or top of the trapezoid.

See Also

trapmf

Introduced before R2006a

Triangular MF

Triangular membership function in Simulink software



Description

The Triangular MF block implements a triangular-shaped membership function. The parameters a and c set the left and right “feet,” or base points, of the triangle. The parameter b sets the location of the triangle peak.

See Also

`trimf`

Introduced before R2006a

Z-shaped MF

Z-shaped membership function in Simulink software



Description

The Z-shaped MF block implements a Z-shaped membership function. Going from left to right the function decreases from 1 to 0. The parameters a and b locate the left and right extremes of the sloped portion of the curve.

See Also

zmf

Introduced before R2006a

Bibliography

- [1] Bezdek, J.C., *Pattern Recognition with Fuzzy Objective Function Algorithms*, Plenum Press, New York, 1981.
- [2] Chiu, S., "Fuzzy Model Identification Based on Cluster Estimation," *Journal of Intelligent & Fuzzy Systems*, Vol. 2, No. 3, Sept. 1994.
- [3] Dubois, D. and H. Prade, *Fuzzy Sets and Systems: Theory and Applications*, Academic Press, New York, 1980.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] Jang, J.-S. R. and C.-T. Sun, "Neuro-fuzzy modeling and control," *Proceedings of the IEEE*, March 1995.
- [8] Jang, J.-S. R. and C.-T. Sun, *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*, Prentice Hall, 1997.
- [9] Kaufmann, A. and M.M. Gupta, *Introduction to Fuzzy Arithmetic*, V.N. Reinhold, 1985.
- [10] Lee, C.-C., "Fuzzy logic in control systems: fuzzy logic controller-parts 1 and 2," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 20, No. 2, pp 404-435, 1990.

- [11] 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.
- [12] Mamdani, E.H., "Advances in the linguistic synthesis of fuzzy controllers," *International Journal of Man-Machine Studies*, Vol. 8, pp. 669-678, 1976.
- [13] 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.
- [14] Schweizer, B. and A. Sklar, "Associative functions and abstract semi-groups," *Publ. Math Debrecen*, 10:69-81, 1963.
- [15] 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, NY, 1977.
- [16] Sugeno, M., *Industrial applications of fuzzy control*, Elsevier Science Pub. Co., 1985.
- [17] Wang, L.-X., *Adaptive fuzzy systems and control: design and stability analysis*, Prentice Hall, 1994.
- [18] Widrow, B. and D. Stearns, *Adaptive Signal Processing*, Prentice Hall, 1985.
- [19] Yager, R., "On a general class of fuzzy connectives," *Fuzzy Sets and Systems*, 4:235-242, 1980.
- [20] 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.
- [21] Zadeh, L.A., "Fuzzy sets," *Information and Control*, Vol. 8, pp. 338-353, 1965.
- [22] 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.
- [23] 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.
- [24] Zadeh, L.A., "Fuzzy Logic," *Computer*, Vol. 1, No. 4, pp. 83-93, 1988.

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

Glossary

Adaptive Neuro-Fuzzy Inference System	(ANFIS) A technique for automatically tuning Sugeno-type inference systems based on training data.
aggregation	The combination of the consequents of each rule in a Mamdani fuzzy inference system in preparation for defuzzification.
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 fulfillment	See firing strength
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.
firing strength	The degree to which the antecedent part of a fuzzy rule is satisfied. The firing strength may be the result of an AND or an OR operation, and it shapes the output function for the rule. Also known as <i>degree of fulfillment</i> .
fuzzification	The process of generating membership values for a fuzzy variable using membership functions.
fuzzy c-means clustering	A data clustering technique wherein each data point belongs to a cluster to a degree specified by a membership grade.
fuzzy inference system (FIS)	The overall name for a system that uses fuzzy reasoning to map an input space to an output space.
fuzzy operators	AND, OR, and NOT operators. These are also known as <i>logical connectives</i> .
fuzzy set	A set that can contain elements with only a partial degree of membership.

fuzzy singleton	A fuzzy set with a membership function that is unity at a one particular point and zero everywhere else.
implication	The process of shaping the fuzzy set in the consequent based on the results of the antecedent in a Mamdani-type FIS.
Mamdani-type inference	A type of fuzzy inference in which the fuzzy sets from the consequent of each rule are combined through the aggregation operator and the resulting fuzzy set is defuzzified to yield the output of the system.
membership function (MF)	A function that specifies the degree to which a given input belongs to a set or is related to a concept.
singleton output function	An output function that is given by a spike at a single number rather than a continuous curve. In the Fuzzy Logic Toolbox software, it is only supported as part of a zero-order Sugeno model.
subtractive clustering	A technique for automatically generating fuzzy inference systems by detecting clusters in input-output training data.
Sugeno-type inference	A type of fuzzy inference in which the consequent of each rule is a linear combination of the inputs. The output is a weighted linear combination of the consequents.
T-conorm	A two-input function that describes a superset of fuzzy union (OR) operators, including maximum, algebraic sum, and any of several parameterized T-conorms Also known as <i>S-norm</i> .
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.