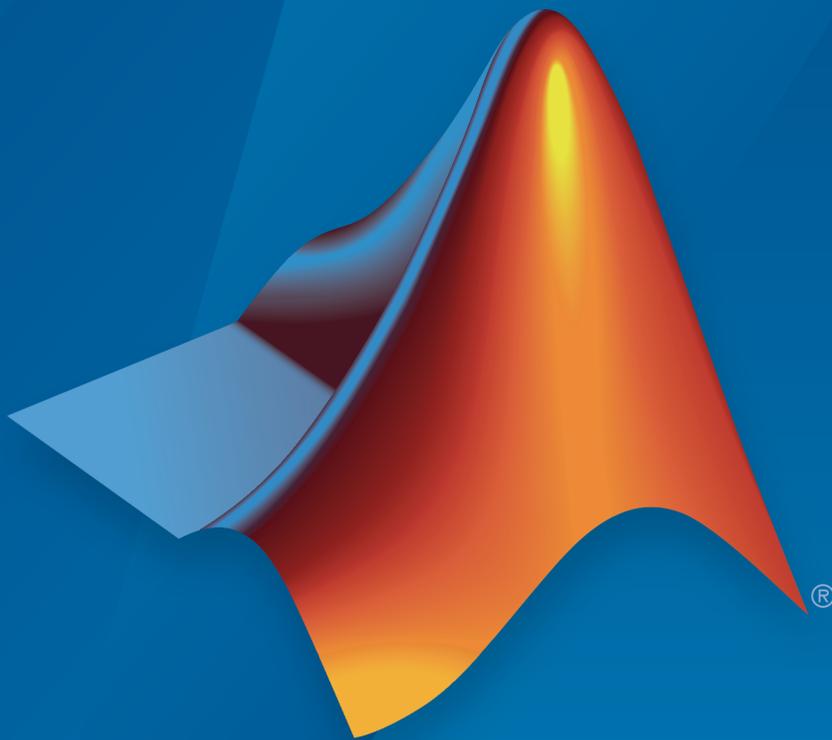


Fuzzy Logic Toolbox™

User's Guide



MATLAB®

R2017a

 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 1995–2017 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)

Getting Started

1

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
Fuzzy vs. Nonfuzzy Logic	1-9

Tutorial

2

Foundations of Fuzzy Logic	2-2
Overview	2-2
Fuzzy Sets	2-3
Membership Functions	2-6
Logical Operations	2-11
If-Then Rules	2-15
References	2-18
Types of Fuzzy Inference Systems	2-20
Fuzzy Inference Process	2-22
Step 1. Fuzzify Inputs	2-23
Step 2. Apply Fuzzy Operator	2-24
Step 3. Apply Implication Method	2-25
Step 4. Aggregate All Outputs	2-25

Step 5. Defuzzify	2-27
Fuzzy Inference Diagram	2-27
What Is Mamdani-Type Fuzzy Inference?	2-30
Build Mamdani Systems Using Fuzzy Logic Designer	2-31
Fuzzy Logic Toolbox Graphical User Interface Tools	2-31
The Basic Tipping Problem	2-33
The Fuzzy Logic Designer	2-34
The Membership Function Editor	2-39
The Rule Editor	2-47
The Rule Viewer	2-50
The Surface Viewer	2-52
Importing and Exporting Fuzzy Inference Systems	2-54
Build Mamdani Systems Using Custom Functions	2-55
Build Fuzzy Inference Systems Using Custom Functions in Fuzzy Logic Designer	2-55
Specifying Custom Membership Functions	2-56
Specifying Custom Inference Functions	2-62
Build Mamdani Systems at the Command Line	2-69
Simulate Fuzzy Inference Systems in Simulink	2-81
Build Your Own Fuzzy Simulink Models	2-88
About the Fuzzy Logic Controller Block	2-88
About the Fuzzy Logic Controller with Ruleviewer Block ...	2-89
Initializing Fuzzy Logic Controller Blocks	2-89
Example: Cart and Pole Simulation	2-90
What Is Sugeno-Type Fuzzy Inference?	2-92
Comparison of Sugeno and Mamdani Systems	2-100
Advantages of the Sugeno Method	2-100
Advantages of the Mamdani Method	2-100

Neuro-Adaptive Learning and ANFIS	3-2
When to Use Neuro-Adaptive Learning	3-2
Model Learning and Inference Through ANFIS	3-3
References	3-5
Comparison of anfis and Neuro-Fuzzy Designer	
Functionality	3-7
Training Data	3-7
Input FIS Structure	3-7
Training Options	3-8
Display Options	3-9
Method	3-9
Output FIS Structure for Training Data	3-10
Training Error	3-10
Step-Size	3-10
Checking Data	3-11
Output FIS Structure for Checking Data	3-11
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	3-15
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
Modeling Inverse Kinematics in a Robotic Arm	3-51
Adaptive Noise Cancellation Using ANFIS	3-62

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
Model Suburban Commuting Using Subtractive Clustering	4-12
Data Clustering Using the Clustering Tool	4-24
Load and Plot the Data	4-25
Perform the Clustering	4-25
Save the Cluster Centers	4-26

5

Fuzzy Inference Engine	5-2
Compile and Evaluate Fuzzy Systems on Windows Platforms	5-3
Compile and Evaluate Fuzzy Systems on UNIX Platforms ..	5-6

Apps — Alphabetical List

6

Functions — Alphabetical List

7

Blocks — Alphabetical List

8

Bibliography

A

Glossary

Getting Started

- “Fuzzy Logic Toolbox Product Description” on page 1-2
- “What Is Fuzzy Logic?” on page 1-3
- “Fuzzy vs. Nonfuzzy Logic” on page 1-9

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?

In this section...

“Description of Fuzzy Logic” on page 1-3

“Why Use Fuzzy Logic?” on page 1-6

“When Not to Use Fuzzy Logic” on page 1-7

“What Can Fuzzy Logic Toolbox Software Do?” on page 1-8

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 2-2. 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.

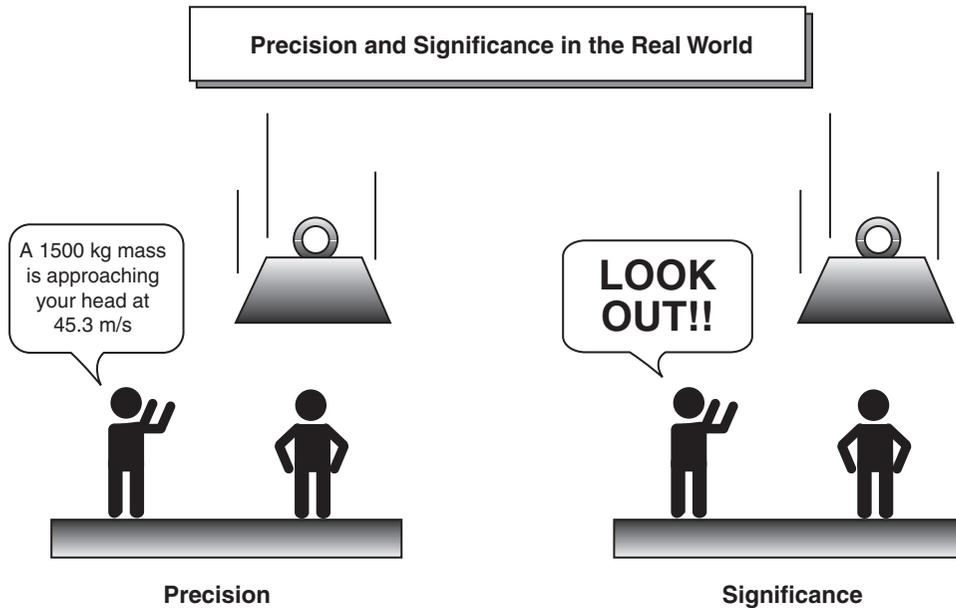
A trend that is growing in visibility relates to the use of fuzzy logic in combination with neurocomputing and genetic algorithms. More generally, fuzzy logic, neurocomputing, and genetic algorithms may be viewed as the principal constituents of what might be called soft computing. Unlike the traditional, 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.

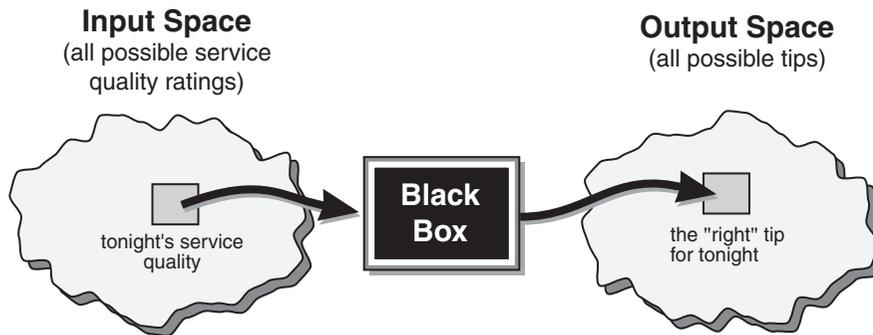
In this sense, fuzzy logic is both old and new because, although the modern and methodical science of fuzzy logic is still young, the concepts of fuzzy logic relies on age-old skills of human reasoning.



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?”

To determine 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

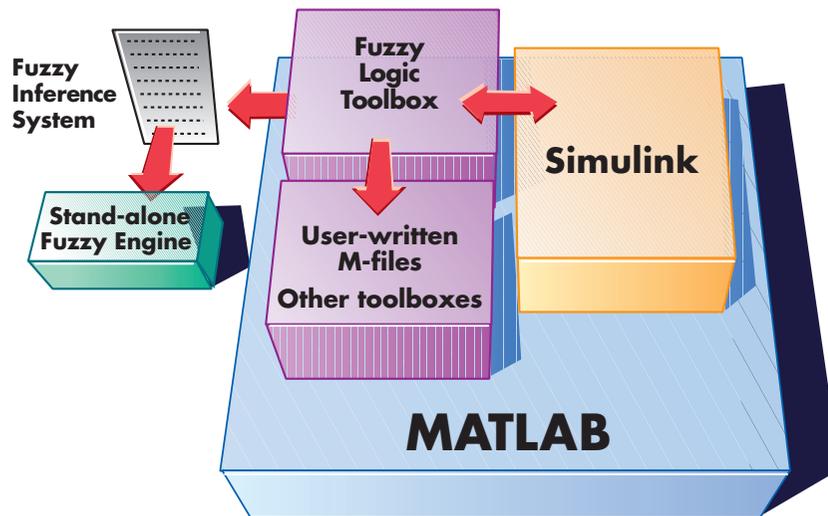
Fuzzy logic is not a cure-all. When should you not use fuzzy logic? The safest statement is the first one made in this introduction: fuzzy logic is a convenient way to map an input space to an output space. If you find it's not convenient, try something else. If a simpler solution already exists, use 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[™], Neural Network Toolbox[™], or Optimization Toolbox[™] software.

More About

- “Foundations of Fuzzy Logic” on page 2-2
- “Fuzzy vs. Nonfuzzy Logic” on page 1-9

Fuzzy vs. Nonfuzzy Logic

The 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.

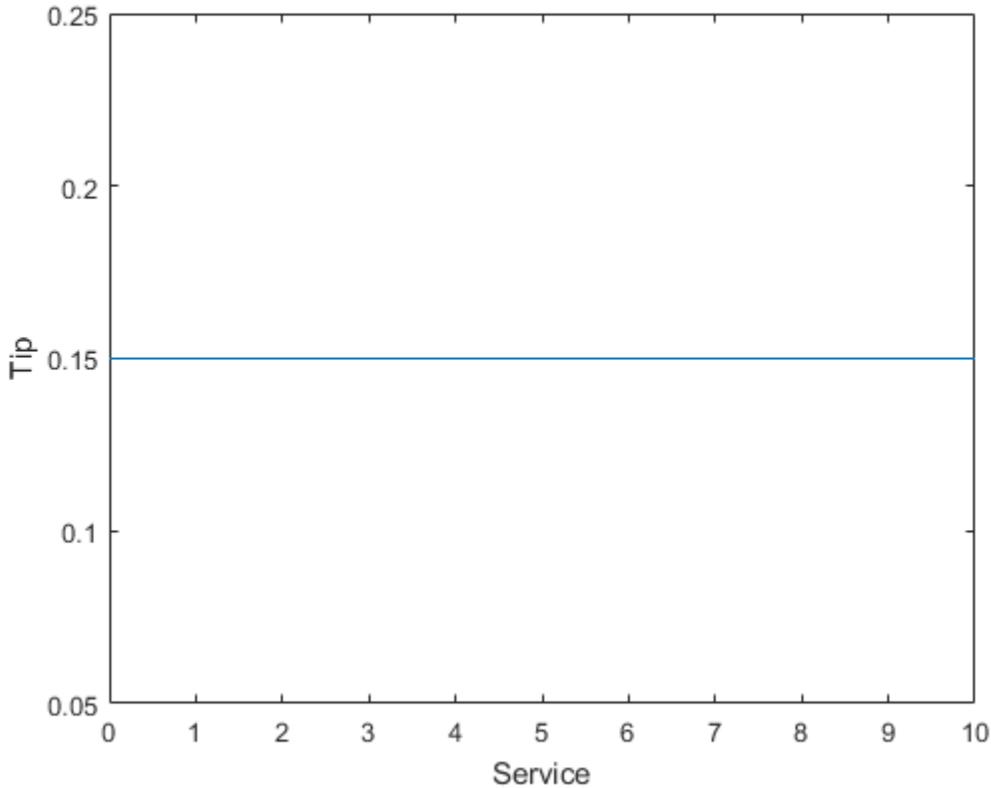
The Basic Tipping Problem. Given a number between 0 and 10 that represents the quality of service at a restaurant (where 10 is excellent), what should the tip be?

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

The Nonfuzzy Approach

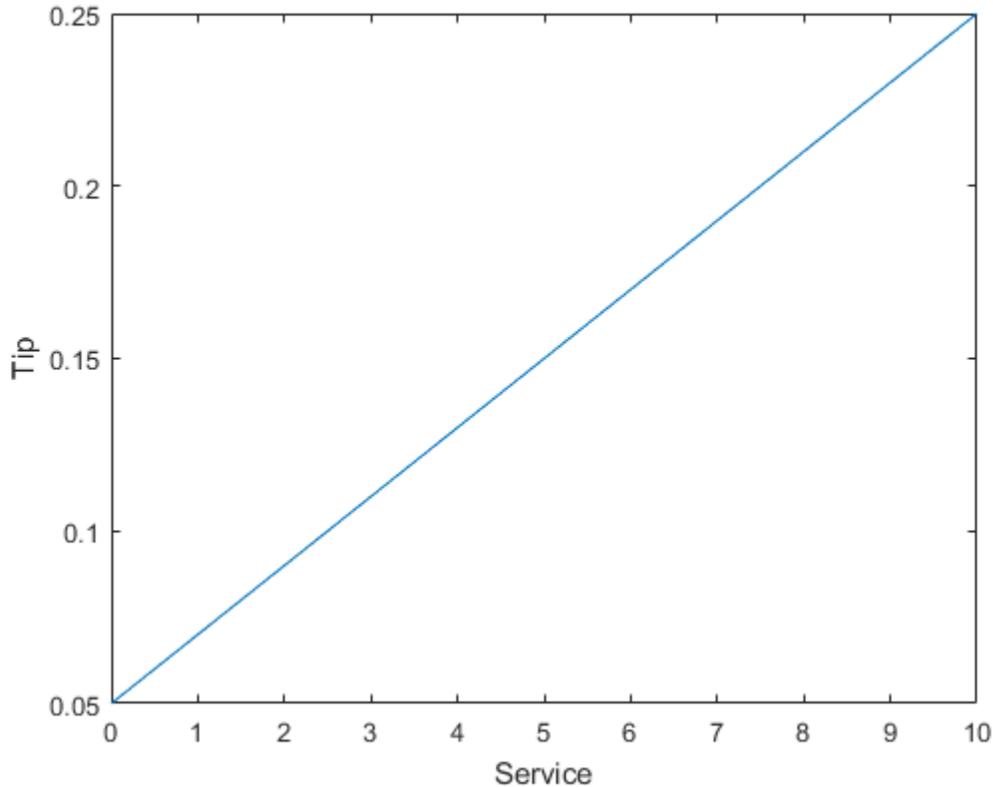
Begin with the simplest possible relationship. Suppose that the tip always equals 15% of the total bill.

```
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 take into account the quality of the service, so you need to add a new term to the equation. Because service is rated on a scale of 0 to 10, you might have the tip go 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])
```



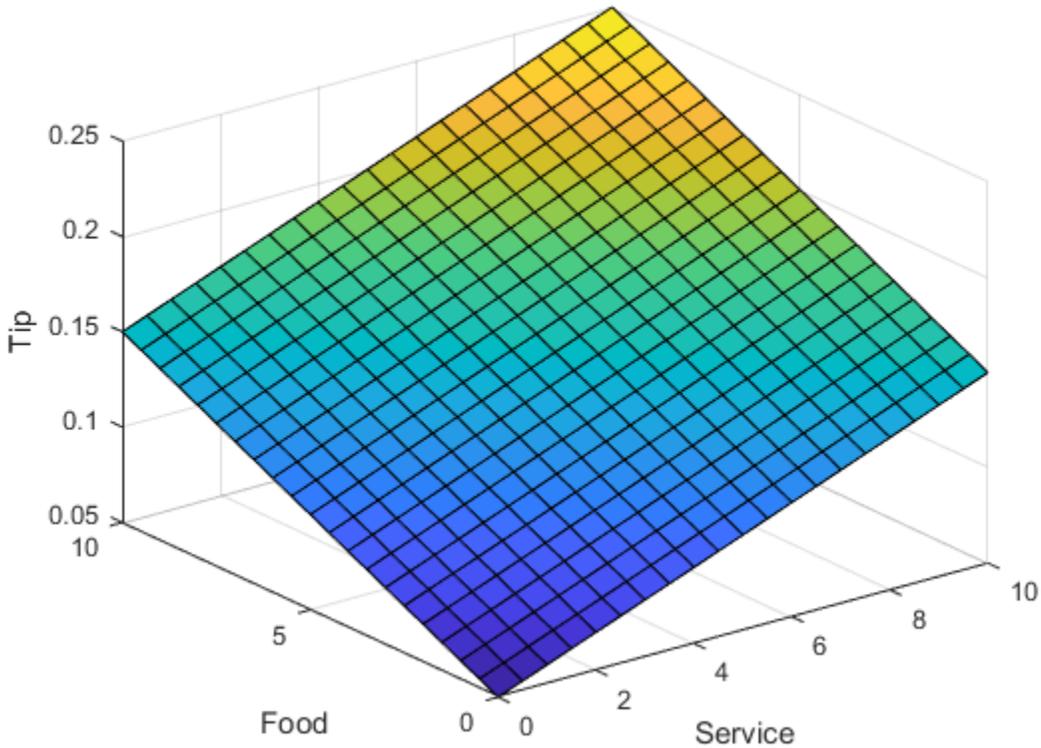
The formula does what you want it to do, and is straight forward. However, you may want the tip to reflect the quality of the food as well. This extension of the problem is defined as follows.

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

See how the formula is affected now that you have added another variable. Try the following equation:

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

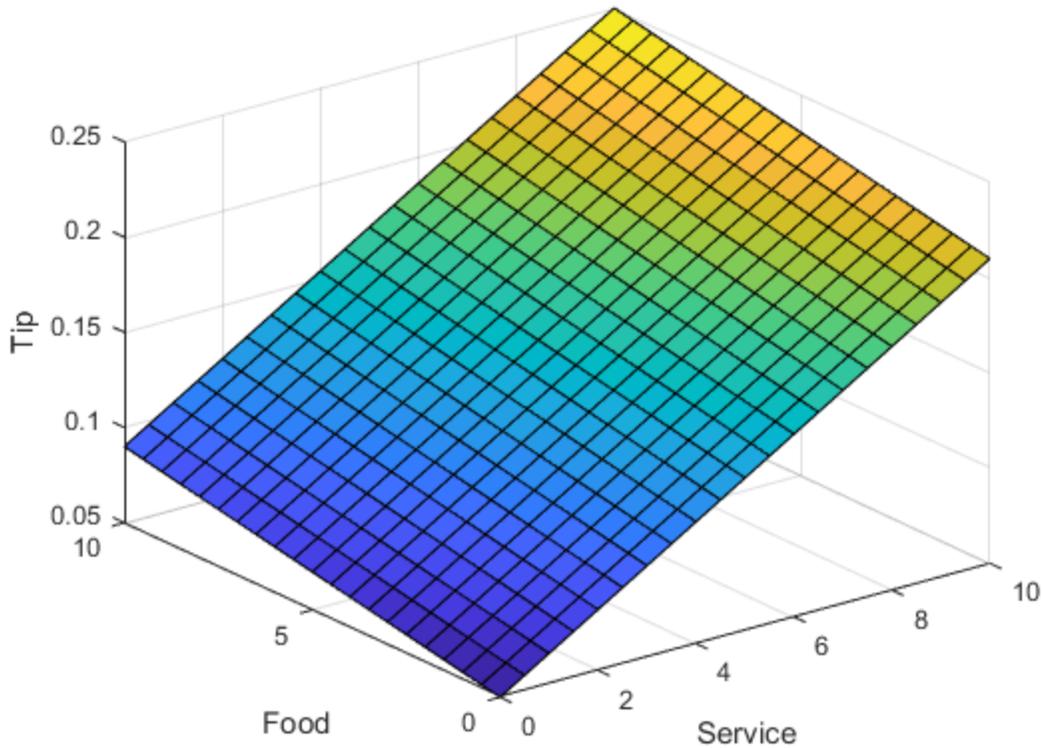
```
surf(S,F,tip)
xlabel('Service')
ylabel('Food')
zlabel('Tip')
```



In this case, the results look satisfactory, but when you look at them closely, they do not seem quite right. Suppose 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%. Try this equation:

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

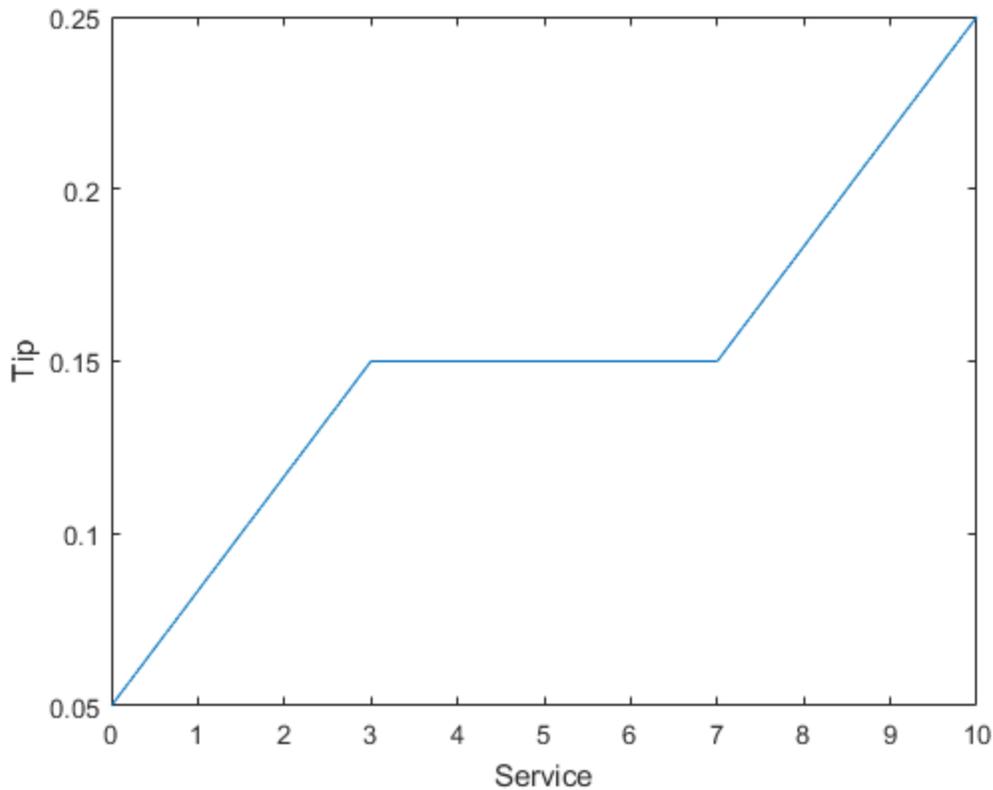
```
xlabel('Service')  
ylabel('Food')  
zlabel('Tip')
```



The response is still somewhat too uniformly linear. Suppose you want more of a flat response in the middle, i.e., you want to give a 15% tip in general, 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])
```



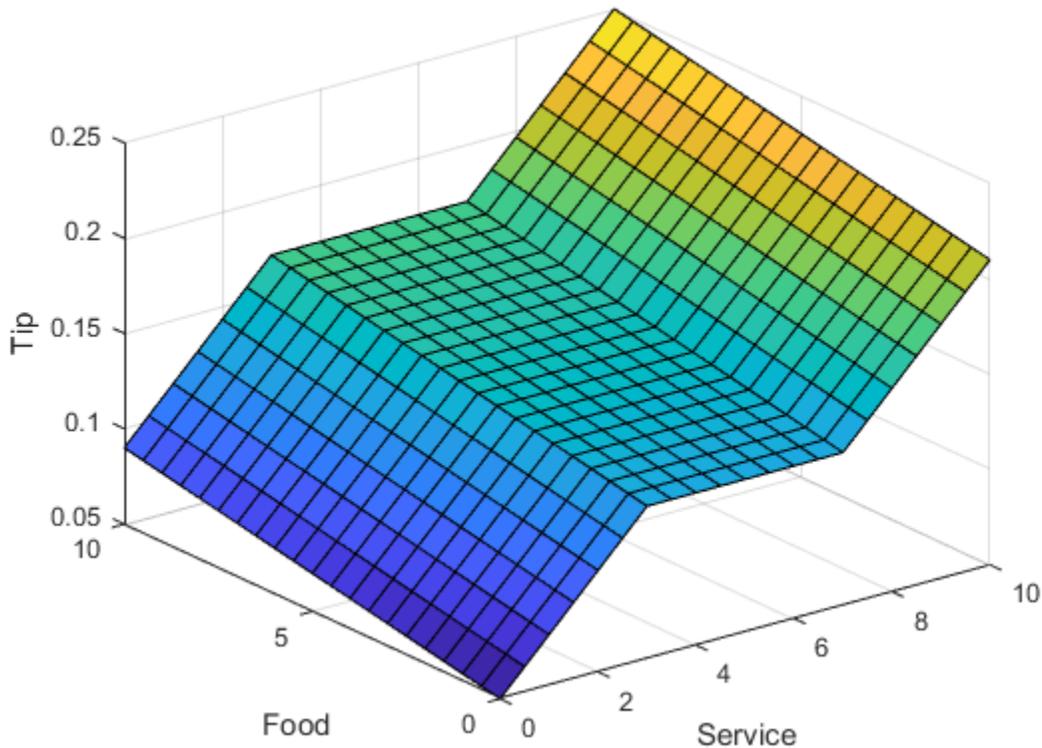
Suppose you extend this to two dimensions, where you take food into account again.

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

```

(1-servRatio)*(0.20/10*F(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 was a little difficult to code this correctly, and it is definitely not easy to modify this code in the future. Moreover, it is even less apparent how the algorithm works to someone who did not see the original design process.

The Fuzzy Logic Approach

You need 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 might 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. If you want to include the food's effect 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 tight 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. Coincidentally, you have just defined 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:

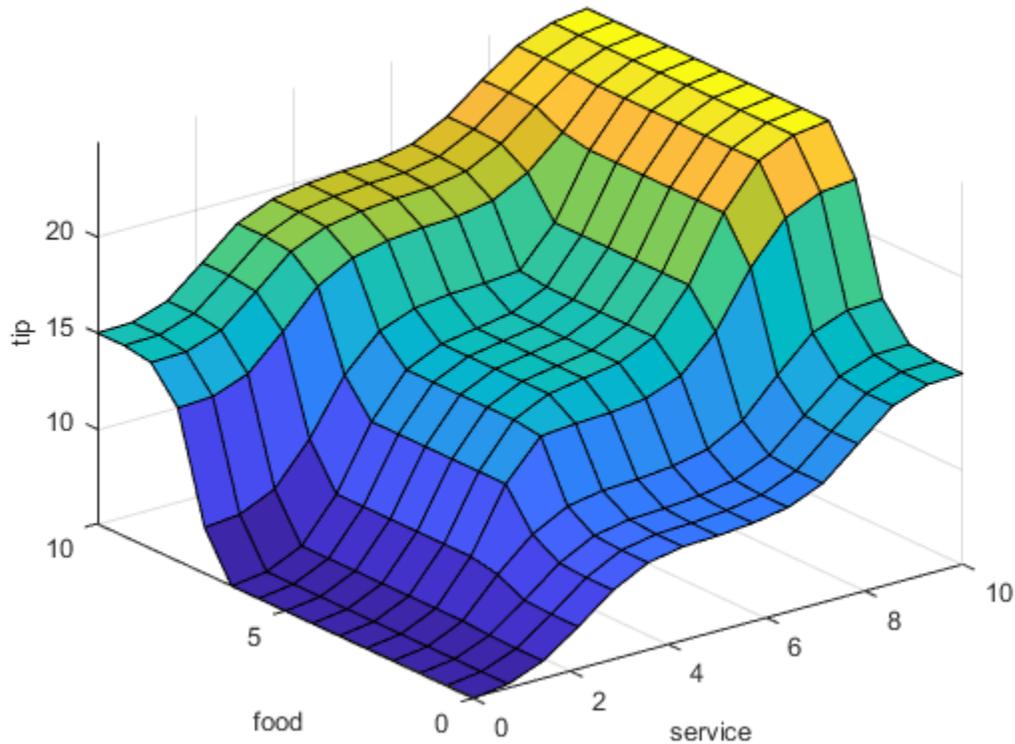
- How are the rules all combined?
- How do I define mathematically what an average tip is?

See other sections of the documentation for detailed answers to these questions. The details of the method don't really change much from problem to problem - the mechanics of fuzzy logic aren't terribly complex. What matters is that you understand that fuzzy logic is adaptable, simple, and easily applied.

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. The mechanics of how fuzzy inference works is explained in the Overview section of Foundations of Fuzzy Logic topic. In the topic, Build Mamdani Systems (GUI), the entire tipping problem is worked through using the Fuzzy Logic Toolbox (TM) apps.

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 very 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 bottom of the list that influenced the shape of the overall output without needing to undo what had already been done. Making the subsequent modification was relatively easy.

Moreover, by using fuzzy logic rules, the maintenance of the structure of the algorithm decouples along fairly clean lines. The notion of an average tip might change from day to day, city to city, country to country, but the underlying logic is the same: if the service is good, the tip should be average.

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 that recalibration will not be so quick and simple.

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

```
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

If, as with a fuzzy system, the comment is identical with the code, think how much more likely your code is to have comments. Fuzzy logic uses language that is clear to you, high level comments, and that also has meaning to the machine, which is why it is a very 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 really no longer the computer - it is your mental model of what the computer is doing. Computers have the ability to make things hopelessly complex; fuzzy logic reclaims the middleground and lets the machine work with your preferences rather than the other way around.

Related Examples

- “Build Mamdani Systems at the Command Line” on page 2-69
- “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31

Tutorial

- “Foundations of Fuzzy Logic” on page 2-2
- “Types of Fuzzy Inference Systems” on page 2-20
- “Fuzzy Inference Process” on page 2-22
- “What Is Mamdani-Type Fuzzy Inference?” on page 2-30
- “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31
- “Build Mamdani Systems Using Custom Functions” on page 2-55
- “Build Mamdani Systems at the Command Line” on page 2-69
- “Simulate Fuzzy Inference Systems in Simulink” on page 2-81
- “Build Your Own Fuzzy Simulink Models” on page 2-88
- “What Is Sugeno-Type Fuzzy Inference?” on page 2-92
- “Comparison of Sugeno and Mamdani Systems” on page 2-100

Foundations of Fuzzy Logic

In this section...

“Overview” on page 2-2

“Fuzzy Sets” on page 2-3

“Membership Functions” on page 2-6

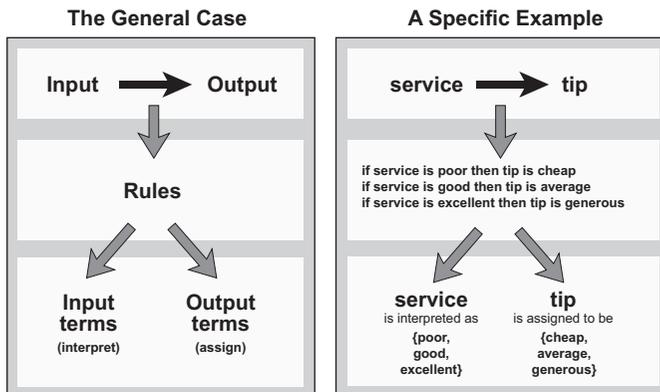
“Logical Operations” on page 2-11

“If-Then Rules” on page 2-15

“References” on page 2-18

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.



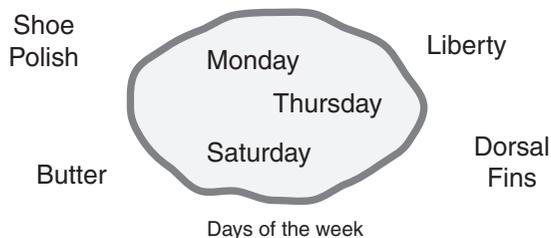
To summarize the concept of fuzzy inference depicted in this figure, *fuzzy inference is a method that interprets the values in the input vector and, based on some set of rules, assigns values to the output vector.*

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.

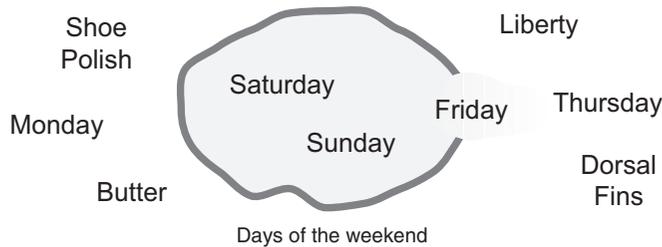


This type of set is called a classical set because it has been around for a long time. It was Aristotle who first formulated the Law of the Excluded Middle, which says X must either be in set A or in set not-A. Another version of this law is:

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.

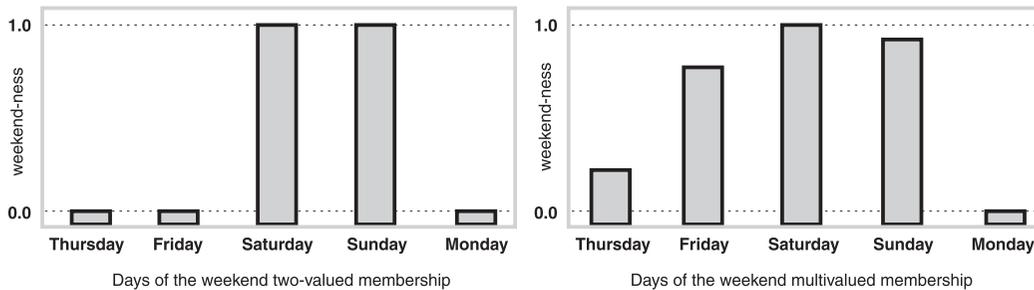
Any statement can be fuzzy. The major advantage that fuzzy reasoning offers is the ability to reply to a yes-no question with a not-quite-yes-or-no answer. Humans do this kind of thing all the time (think how rarely you get a straight answer to a seemingly simple question), but it is a rather new trick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

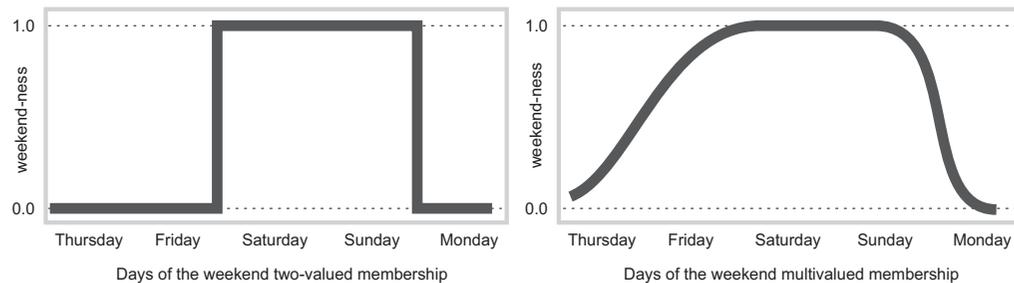
Q: Is Friday a weekend day?
 A: 0.8 (for the most part yes, but not completely)
 Q: Is Sunday a weekend day?
 A: 0.95 (yes, but not quite as much as Saturday).

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.

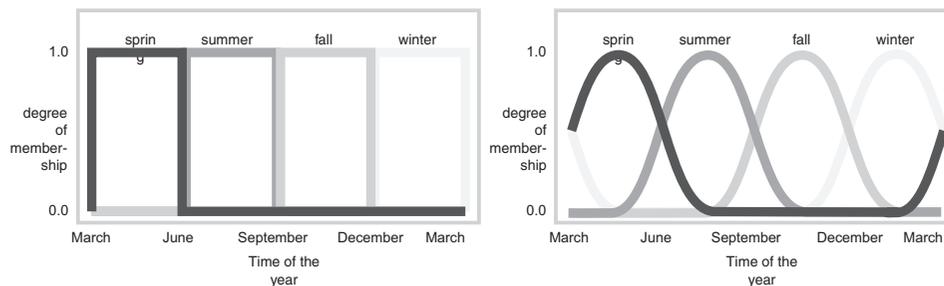


By making the plot continuous, you are defining the degree to which any given instant belongs in the weekend rather than an entire day. In the plot on the left, notice that at

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 2-6 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).

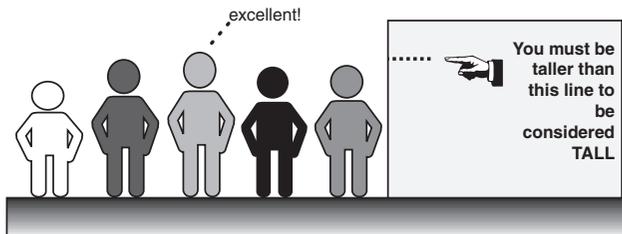


Membership Functions

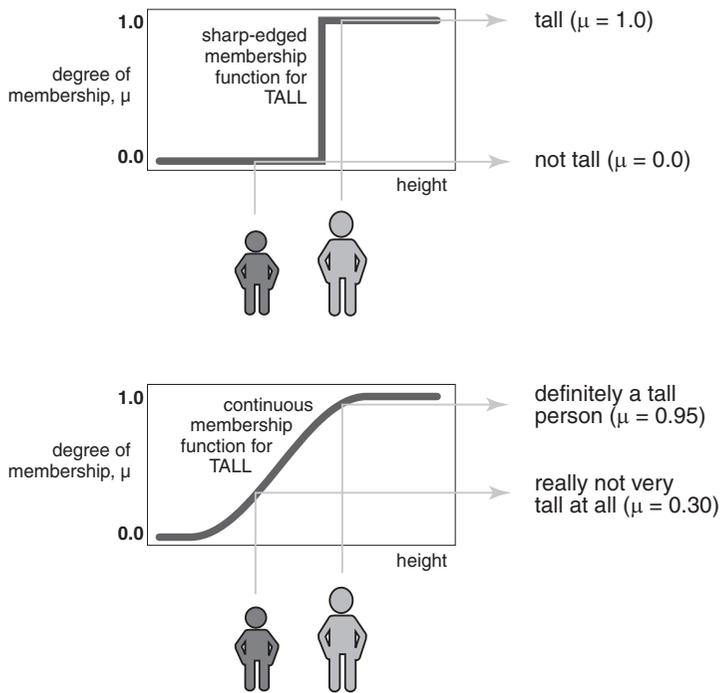
A *membership function* (MF) is a curve that defines how each point in the input space is mapped to a membership value (or degree of membership) between 0 and 1. The input space is sometimes referred to as the *universe of discourse*, a fancy name for a simple concept.

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.

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

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

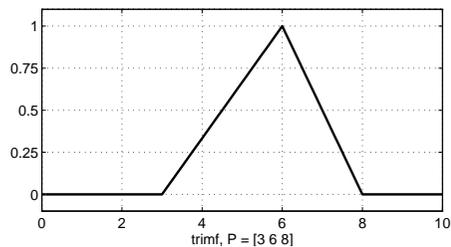
$\mu_A(x)$ is called the membership function (or MF) of x in A . The membership function maps each element of X to a membership value between 0 and 1.

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

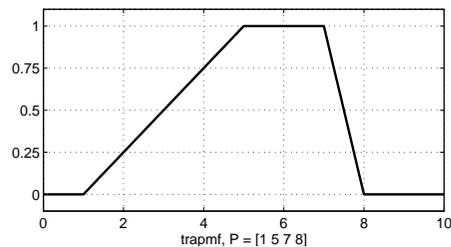
- Piece-wise linear functions
- Gaussian distribution function
- Sigmoid curve
- Quadratic and cubic polynomial curves

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.



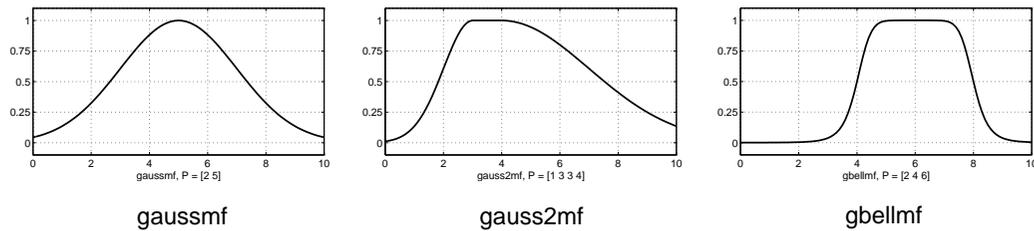
`trimf`



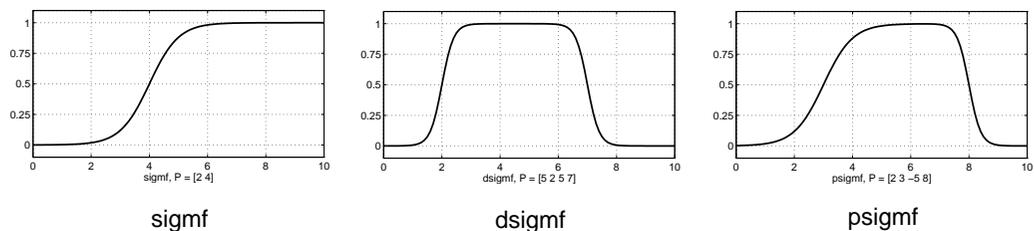
`trapmf`

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`.

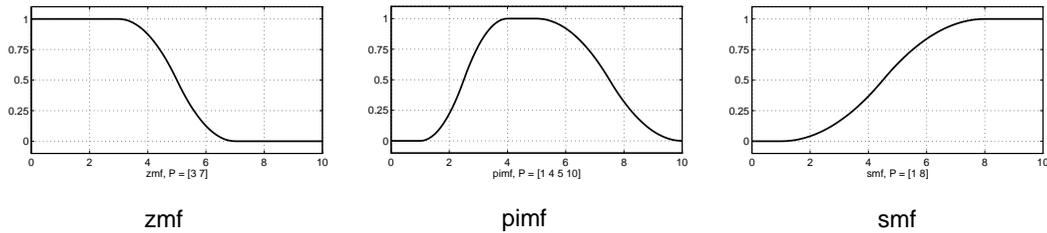
The *generalized bell* membership function is specified by three parameters and has the function name `gbellmf`. The bell membership function has one more parameter than the Gaussian membership function, so it can approach a non-fuzzy set if the free parameter is tuned. Because of their smoothness and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



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`.



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

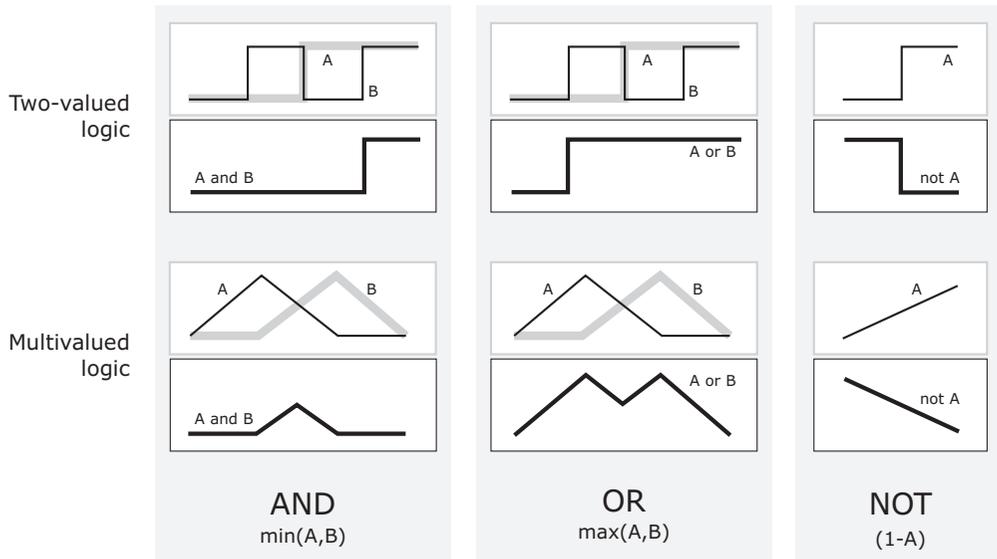
OR

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 $T(.,.)$ 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 $S(.,.)$ 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.

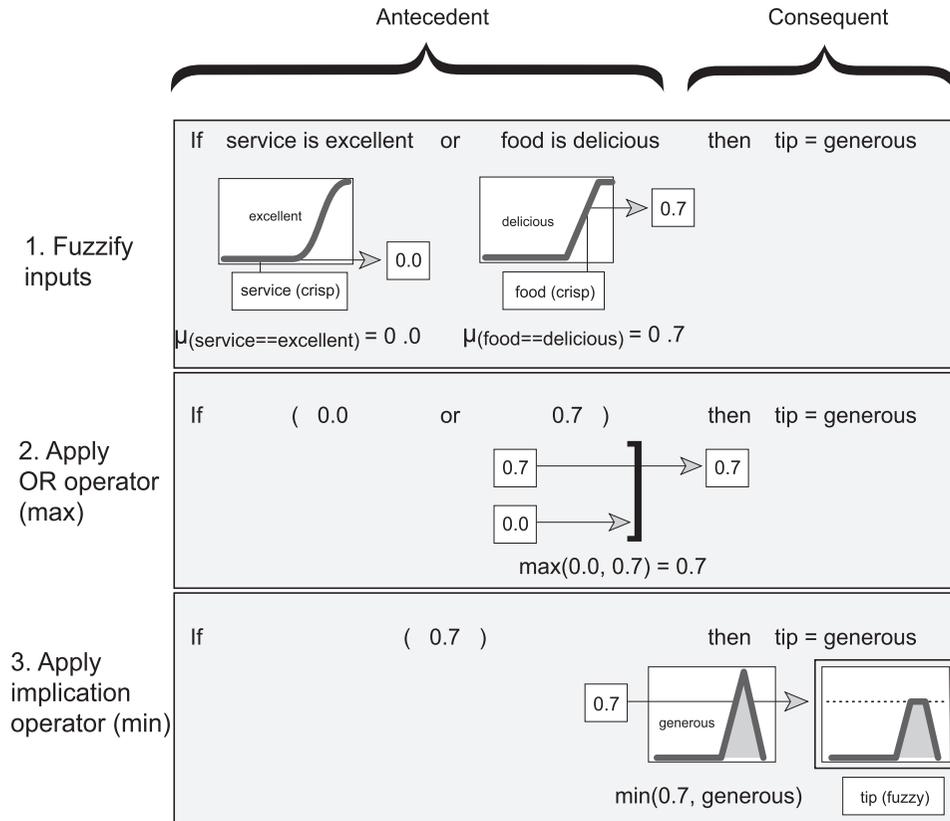
The antecedent of a rule can have multiple parts.

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 to 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 Mamdani Systems Using Fuzzy Logic Designer" on page 2-31 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.

More About

- "Fuzzy Inference Process" on page 2-22
- "Fuzzy vs. Nonfuzzy Logic" on page 1-9

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.

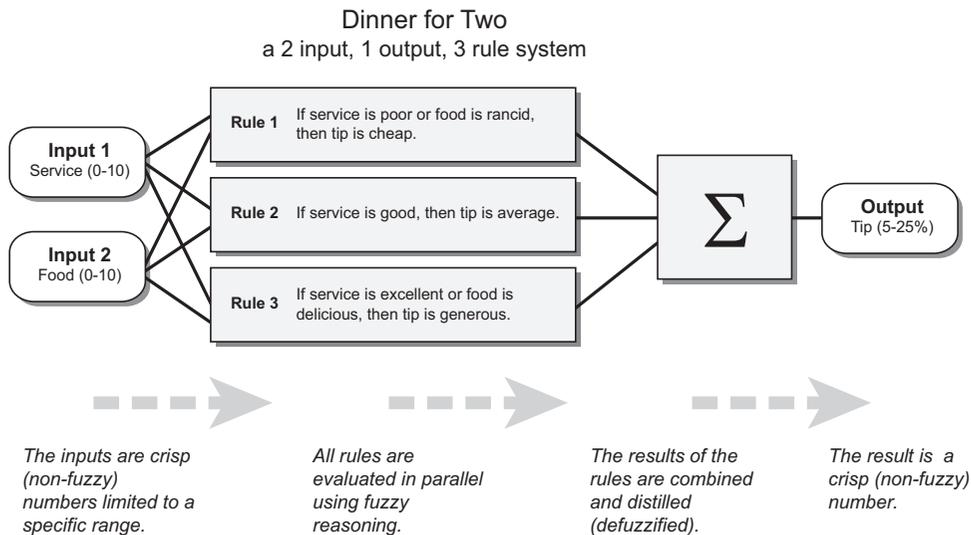
More About

- "Fuzzy Inference Process" on page 2-22
- "What Is Mamdani-Type Fuzzy Inference?" on page 2-30
- "What Is Sugeno-Type Fuzzy Inference?" on page 2-92
- "Comparison of Sugeno and Mamdani Systems" on page 2-100

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 of the pieces that are described in “Membership Functions” on page 2-6, “Logical Operations” on page 2-11, and “If-Then Rules” on page 2-15.

This section describes the fuzzy inference process and uses the example of the two-input, one-output, three-rule tipping problem “The Basic Tipping Problem” on page 2-33 that you saw in the introduction in more detail. 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 one of the more important aspects of fuzzy logic systems. Instead of sharp switching between modes based on breakpoints, logic flows smoothly from regions where the system's behavior is dominated by either one rule or another.

Fuzzy inference process comprises of five parts:

- Fuzzification of the input variables
- Application of the fuzzy operator (AND or OR) in the antecedent

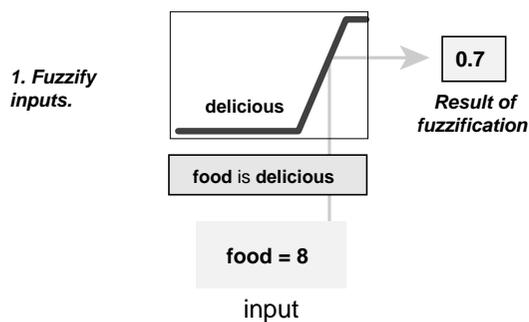
- Implication from the antecedent to the consequent
- Aggregation of the consequents across the rules
- Defuzzification

A fuzzy inference diagram displays all parts of the fuzzy inference process — from fuzzification through defuzzification.

Step 1. Fuzzify Inputs

The first step is to take the inputs and determine the degree to which they belong to each of the appropriate fuzzy sets via membership functions. In 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 between 0 and 10) and the output is a fuzzy degree of membership in the qualifying linguistic set (always the interval between 0 and 1). Fuzzification of the input amounts to either a table lookup or a function evaluation.

This example is built on three rules, and each of the rules depends on resolving the inputs into a number of different fuzzy linguistic sets: service is poor, service is good, food is rancid, food is delicious, and so on. Before the rules can be evaluated, the inputs must be fuzzified according to each of these linguistic sets. For example, to what extent is the food really delicious? The following figure shows how well the food at the hypothetical restaurant (rated on a scale of 0 to 10) qualifies, (via its membership function), as the linguistic variable delicious. In this case, we rated the food as an 8, which, given your 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.

Step 2. Apply Fuzzy Operator

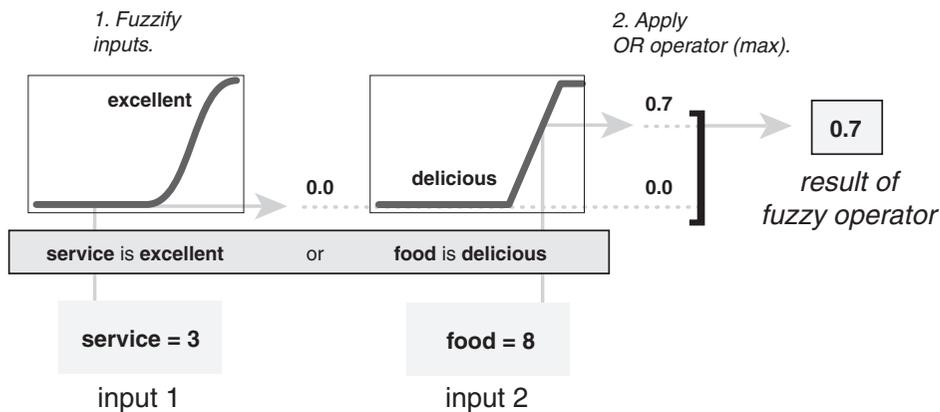
After the inputs are fuzzified, you know the degree to which each part of the antecedent is satisfied for each rule. If the antecedent of a given rule has more than one part, the fuzzy operator is applied to obtain one number that represents the result of the antecedent for that rule. This number is 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 2-11 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

$$\text{probor}(a,b) = a + b - ab$$

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.

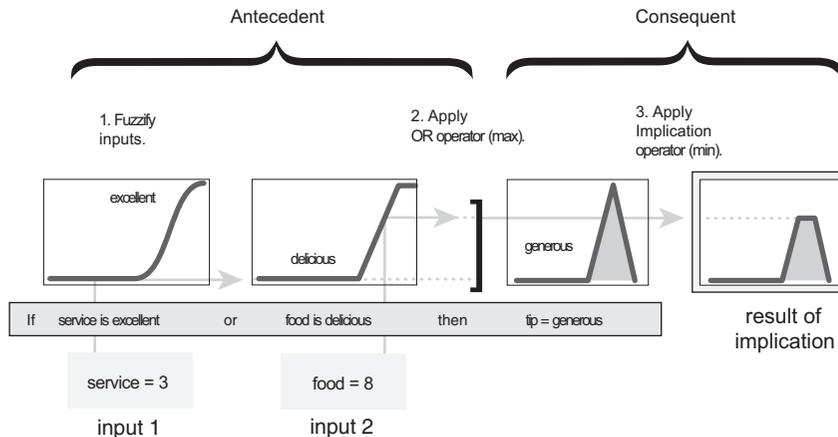
The following figure shows the OR operator *max* at work, evaluating the antecedent of the rule 3 for the tipping calculation. The two different pieces of the antecedent (service is excellent and food is delicious) yielded the fuzzy membership values 0.0 and 0.7 respectively. The fuzzy OR operator simply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



Step 3. Apply Implication Method

Before applying the implication method, you must determine the rule's weight. Every rule has a *weight* (a number between 0 and 1), which is applied to the number given by the antecedent. Generally, this weight is 1 (as it is for this example) and thus has no effect at all on the implication process. From time to time you may want to weight one rule relative to the others by changing its weight value to something other than 1.

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.



Step 4. Aggregate All Outputs

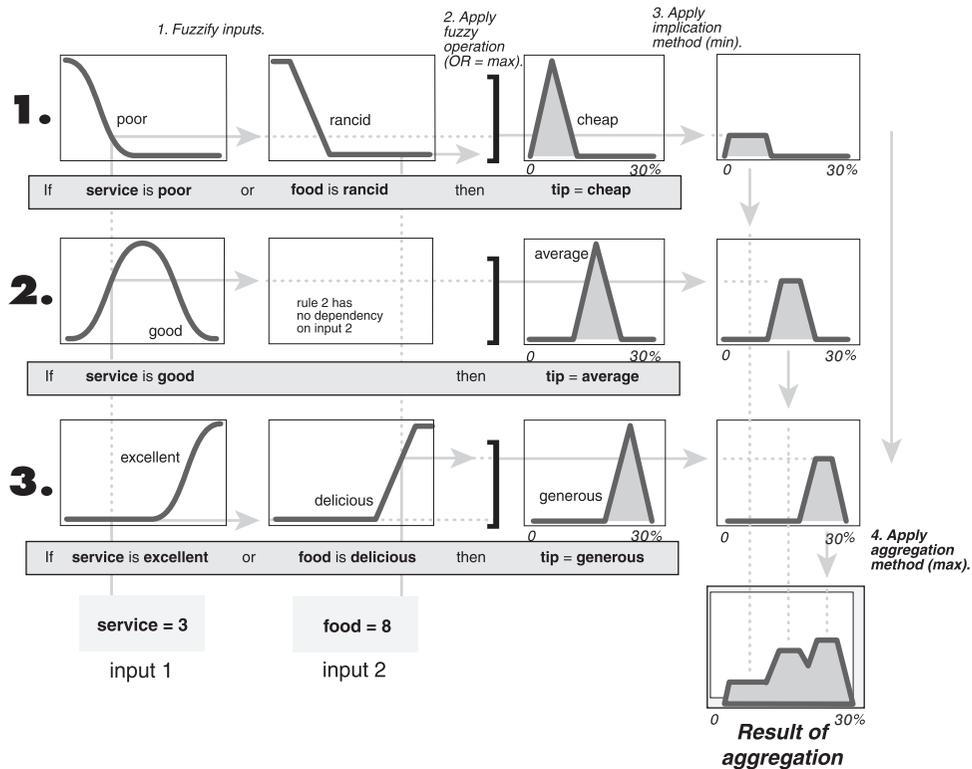
Because decisions are based on the testing of all of the rules in a FIS, the rules must be combined in some manner in order to make a decision. Aggregation is the process by which the fuzzy sets that represent the outputs of each rule are combined into a single fuzzy set. Aggregation only occurs once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

truncated output functions returned by the implication process for each rule. The output of the aggregation process is one fuzzy set for each output variable.

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

- max (maximum)
- probor (probabilistic OR)
- sum (simply the sum of each rule's output set)

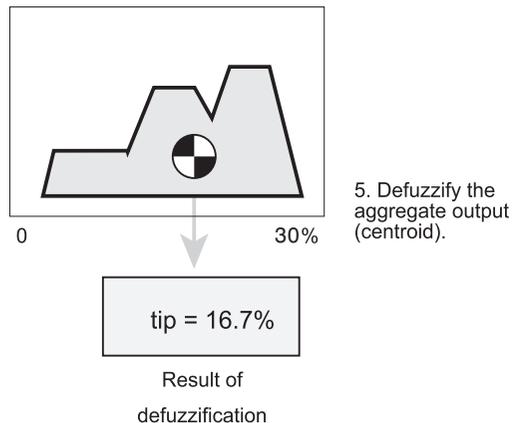
In the following diagram, all three rules have been placed together to show how the output of each rule is combined, or aggregated, into a single fuzzy set whose membership function assigns a weighting for every output (tip) value.



Step 5. Defuzzify

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

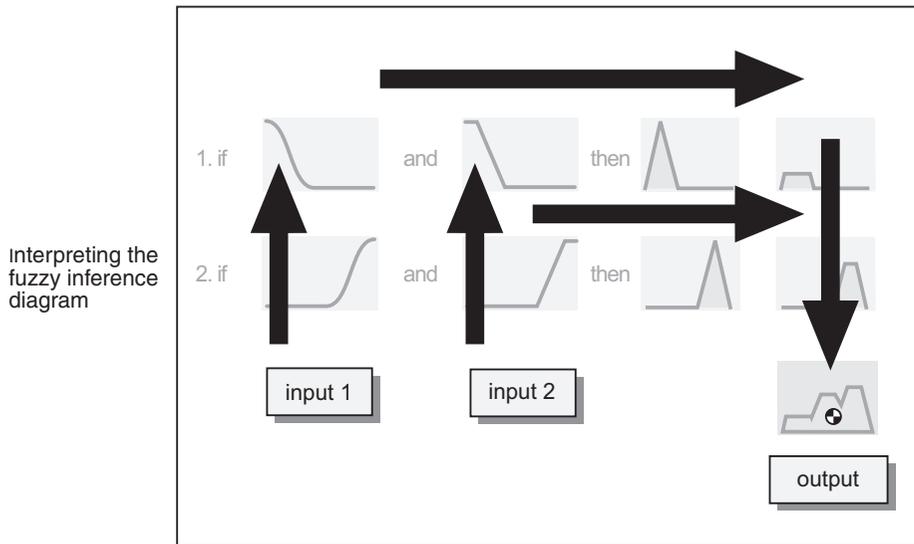
There are five built-in defuzzification methods supported: centroid, bisector, middle of maximum (the average of the maximum value of the output set), largest of maximum, and smallest of maximum. Perhaps the most popular defuzzification method is the centroid calculation, which returns the center of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



While the aggregate output fuzzy set has 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.

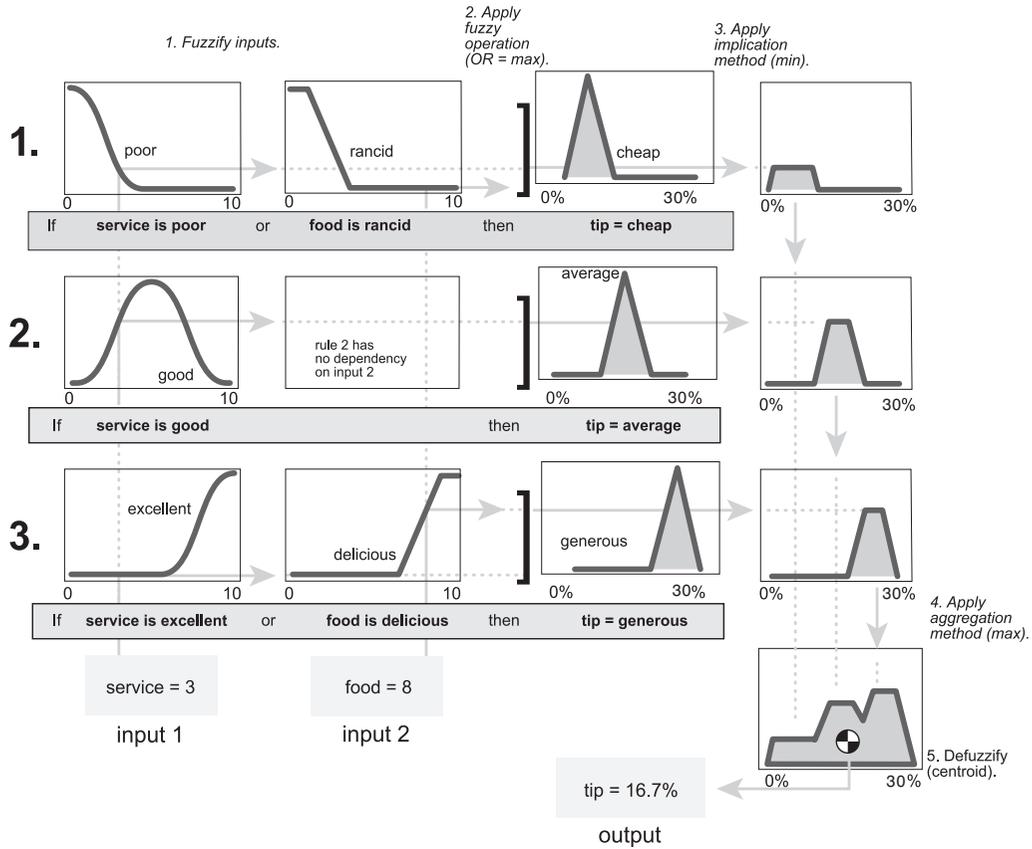
Fuzzy Inference Diagram

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, then across each row, or rule, and then down the rule outputs to finish 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. There is a lot to see in a fuzzy inference diagram, but after you become accustomed to it, you can learn a lot about a system very quickly. For instance, from this diagram with these particular inputs, you can easily see that the implication method is truncation with the *min* function. The *max* function is being used for the fuzzy OR operation. Rule 3 (the bottom-most row in the diagram shown previously) is having the strongest influence on the output. and so on. The Rule Viewer described in “The Rule Viewer” on page 2-50 is a MATLAB implementation of the fuzzy inference diagram.



More About

- “Membership Functions” on page 2-6
- “Logical Operations” on page 2-11
- “If-Then Rules” on page 2-15
- “Types of Fuzzy Inference Systems” on page 2-20

What Is Mamdani-Type Fuzzy Inference?

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.

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.

More About

- "Comparison of Sugeno and Mamdani Systems" on page 2-100
- "What Is Sugeno-Type Fuzzy Inference?" on page 2-92
- "Build Mamdani Systems Using Fuzzy Logic Designer" on page 2-31
- "Build Mamdani Systems at the Command Line" on page 2-69
- "Build Mamdani Systems Using Custom Functions" on page 2-55

Build Mamdani Systems Using Fuzzy Logic Designer

In this section...

“Fuzzy Logic Toolbox Graphical User Interface Tools” on page 2-31

“The Basic Tipping Problem” on page 2-33

“The Fuzzy Logic Designer” on page 2-34

“The Membership Function Editor” on page 2-39

“The Rule Editor” on page 2-47

“The Rule Viewer” on page 2-50

“The Surface Viewer” on page 2-52

“Importing and Exporting Fuzzy Inference Systems” on page 2-54

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 2-33, using the Fuzzy Logic Toolbox UI tools.

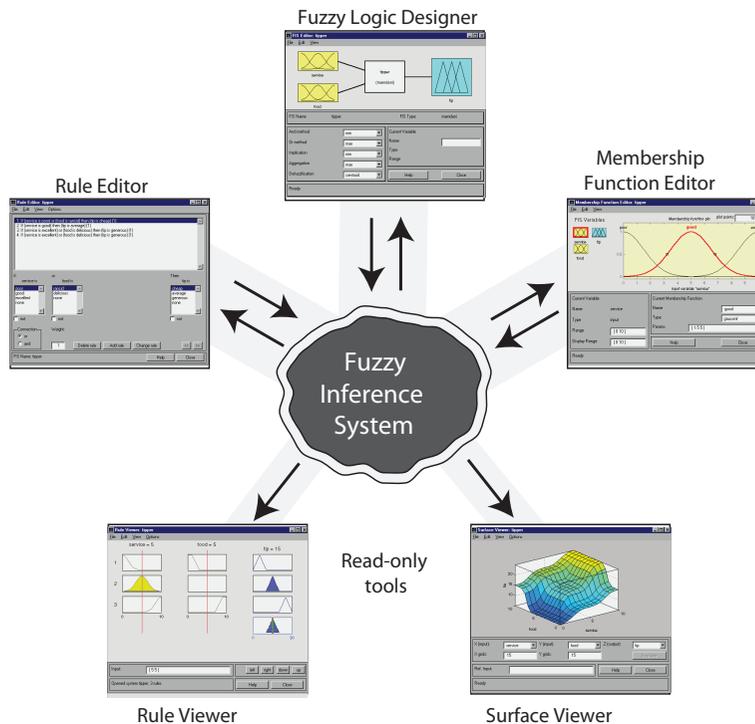
You use the following tools to build, edit, and view fuzzy inference systems:

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

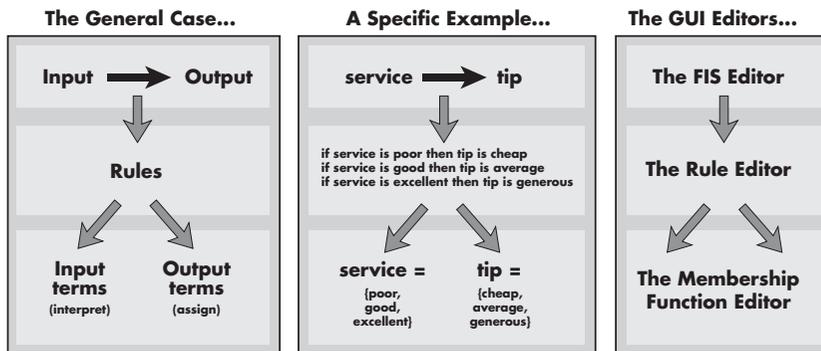
Fuzzy Logic Toolbox software does not limit the number of inputs. However, the number of inputs may be limited by the available memory of your machine. If the number of inputs is too large, or the number of membership functions is too big, then it may also be difficult to analyze the FIS using the other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

- **Membership Function Editor** to define the shapes of all the membership functions associated with each variable
- **Rule Editor** to edit the list of rules that defines the behavior of the system.
- **Rule Viewer** 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** 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 FIS 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 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 2-34.
- 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 Mamdani Systems at the Command Line” on page 2-69.

The Basic Tipping Problem

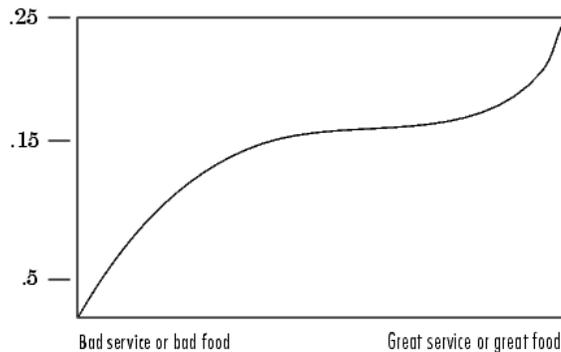
This example uses a two-input, one-output tipping problem based on tipping practices in the U.S.

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.

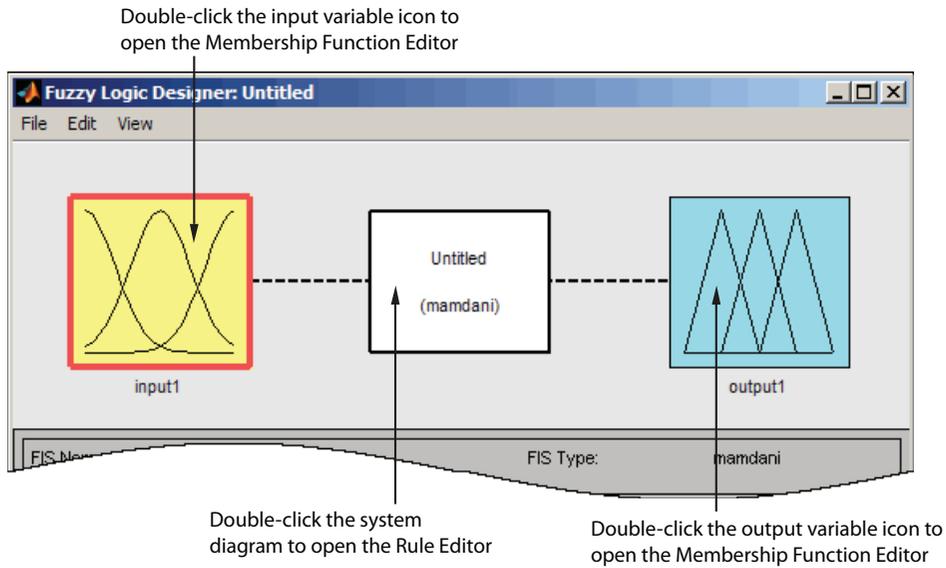
Now that you know the rules and have an idea of what the output should look like, use the UI tools to construct a fuzzy inference system for this decision process.

The Fuzzy Logic Designer

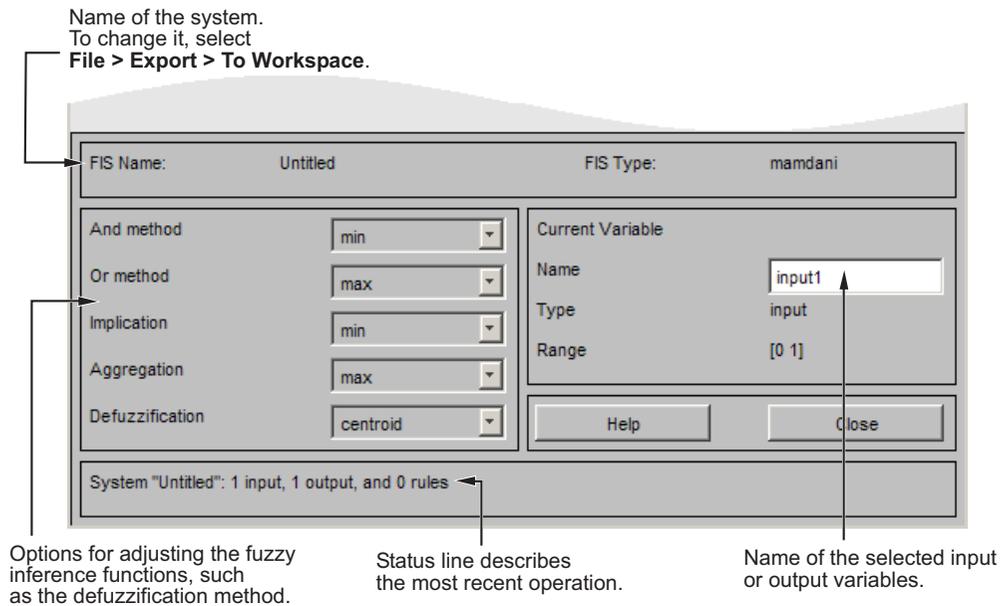
The **Fuzzy Logic Designer** displays information about a fuzzy inference system. To open the **Fuzzy Logic Designer**, type the following command at the MATLAB prompt:

```
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 2-92.

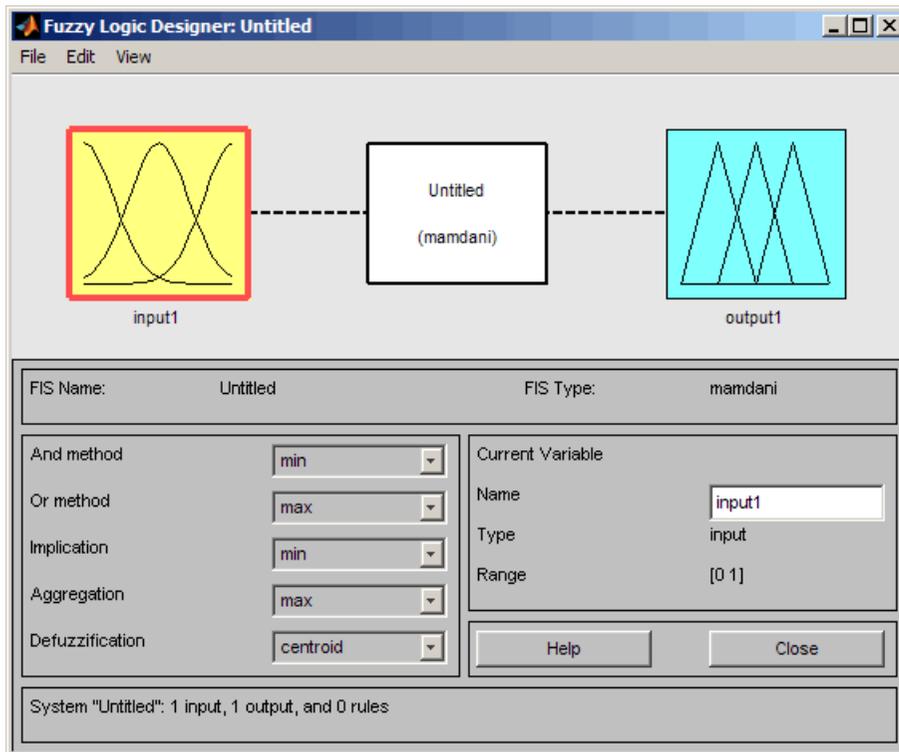
In the **Fuzzy Logic Designer**:

- 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 2-33 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
```

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

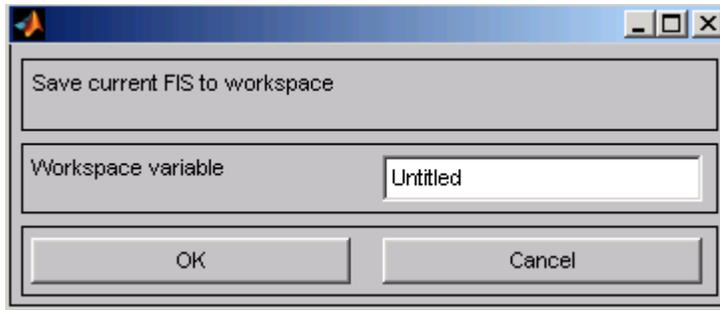
In this example, you construct a two-input, one output system. The two inputs are **service** and **food**. The one output is **tip**.

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

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

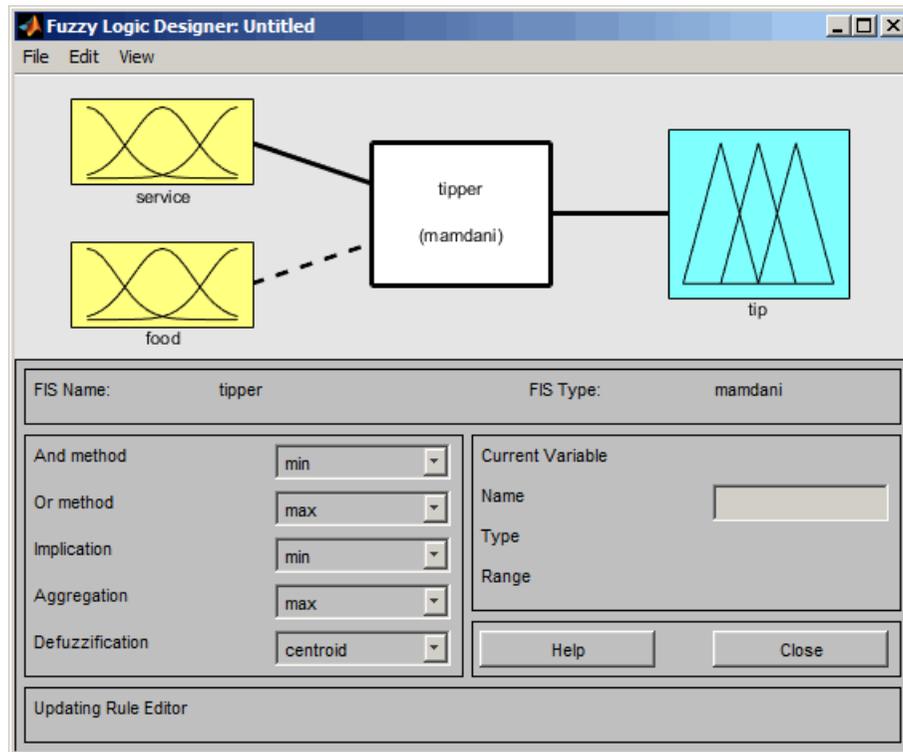
A second yellow box labeled **input2** appears.

- 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`.

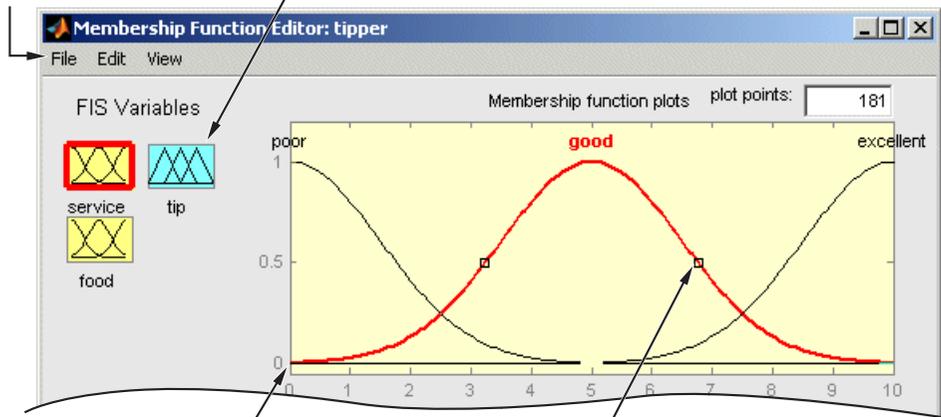
The Membership Function Editor

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.

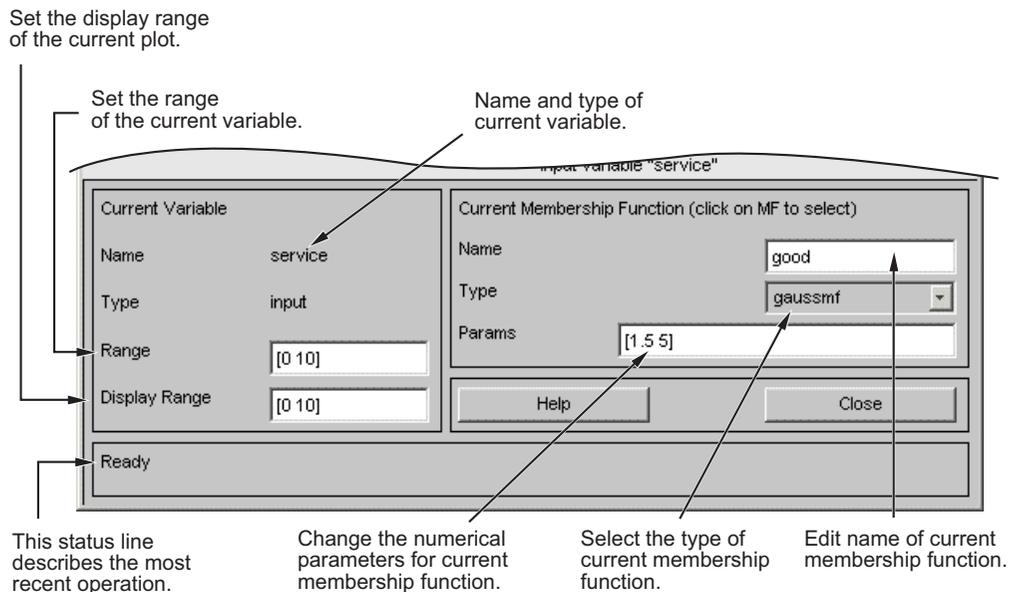
Menu commands for saving, opening, and editing a fuzzy system.

"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.

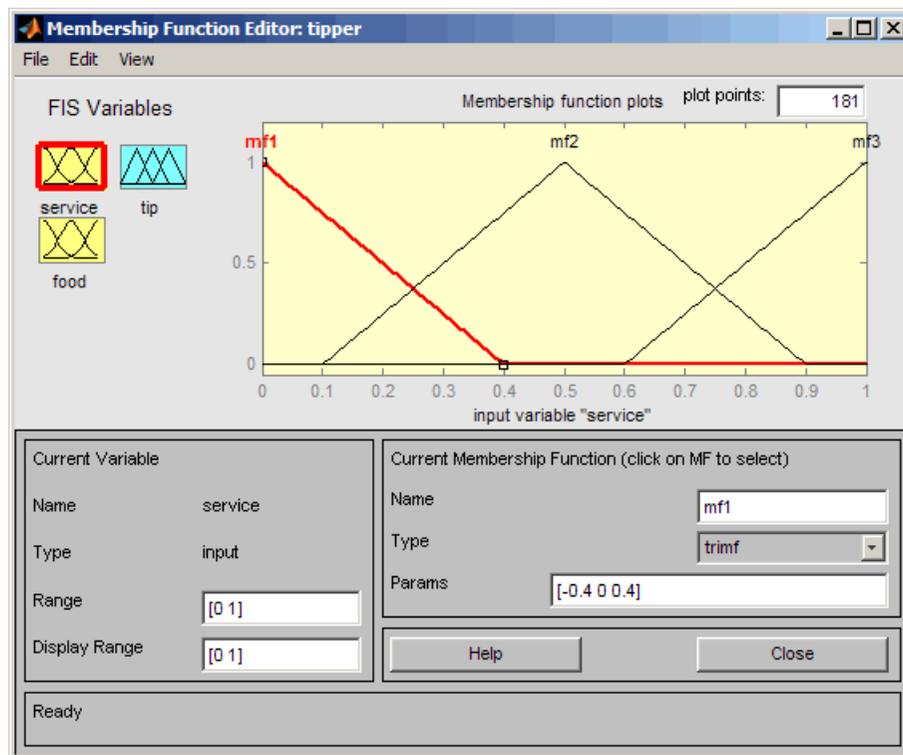
The membership functions from the current variable are displayed in the main graph. These membership functions can be manipulated in two ways. You can first use the mouse to select a particular membership function associated with a given variable quality, (such as poor, for the variable, service), and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

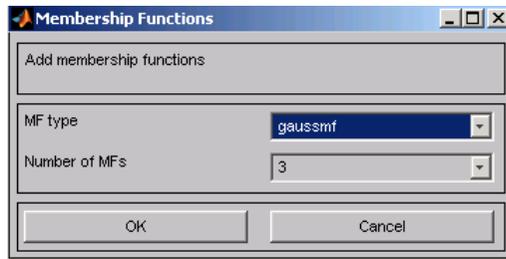
Below the Variable Palette is some information about the type and name of the current variable. There is a text field in this region that lets you change the limits of the current variable's range (universe of discourse) and another that lets you set the limits of the current plot (which has no real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

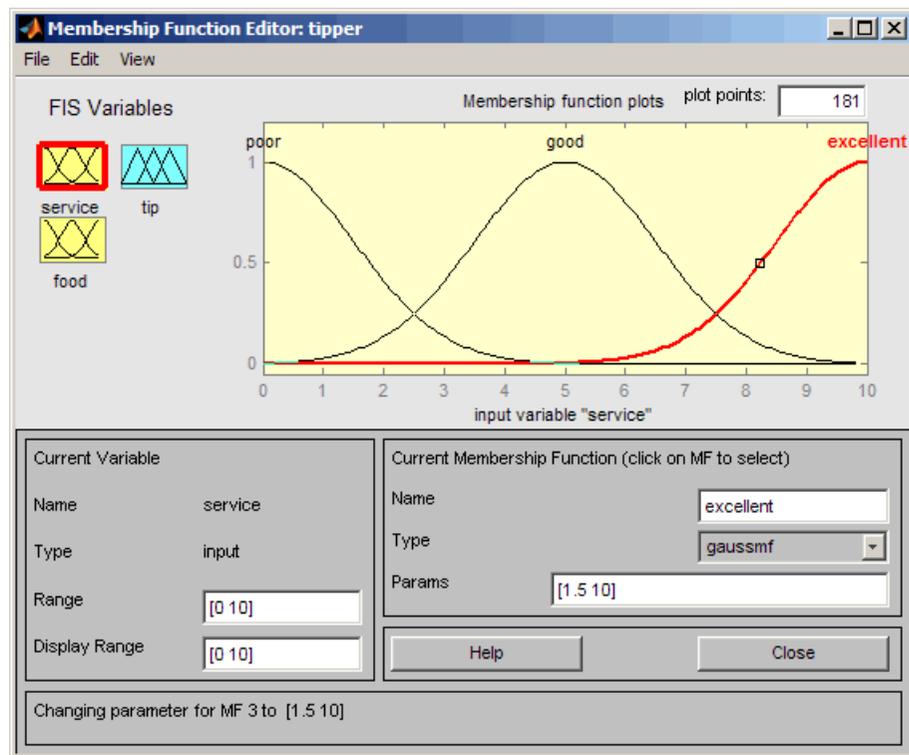
The two inputs of **Params** represent the standard deviation and center for the Gaussian curve.

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`.

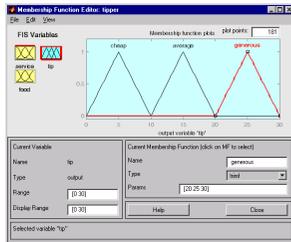
- b Select **Edit > Add MFs** to open the Membership Functions dialog box.
 - c In the Membership Functions dialog box, select `trapmf` as the **MF Type**.
 - d Select 2 in the **Number of MFs** drop-down list.
 - e Click **OK** to add two trapezoidal curves to the input variable `food`.
- 8 Rename the 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:

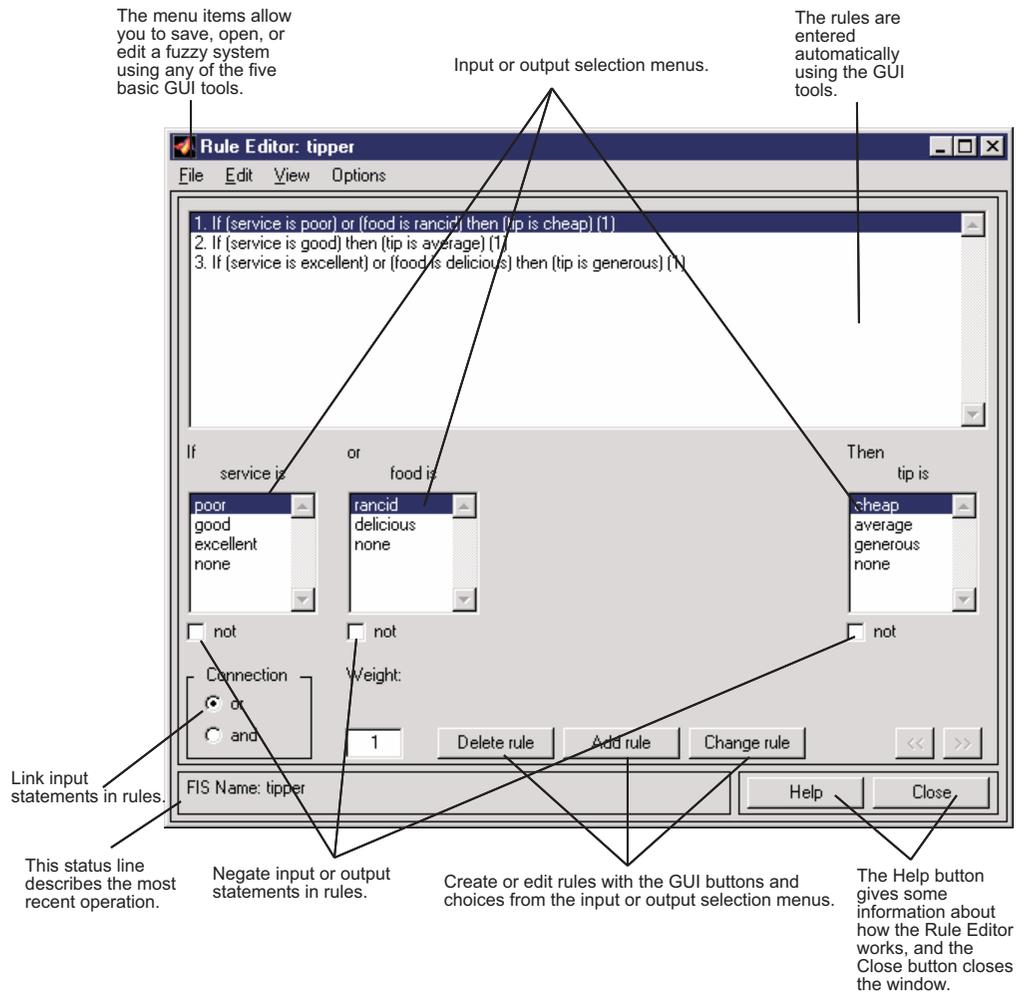
- In the **Name** field, enter **generous**.
- In the **Params** field, enter **[20 25 30]**.

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



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.

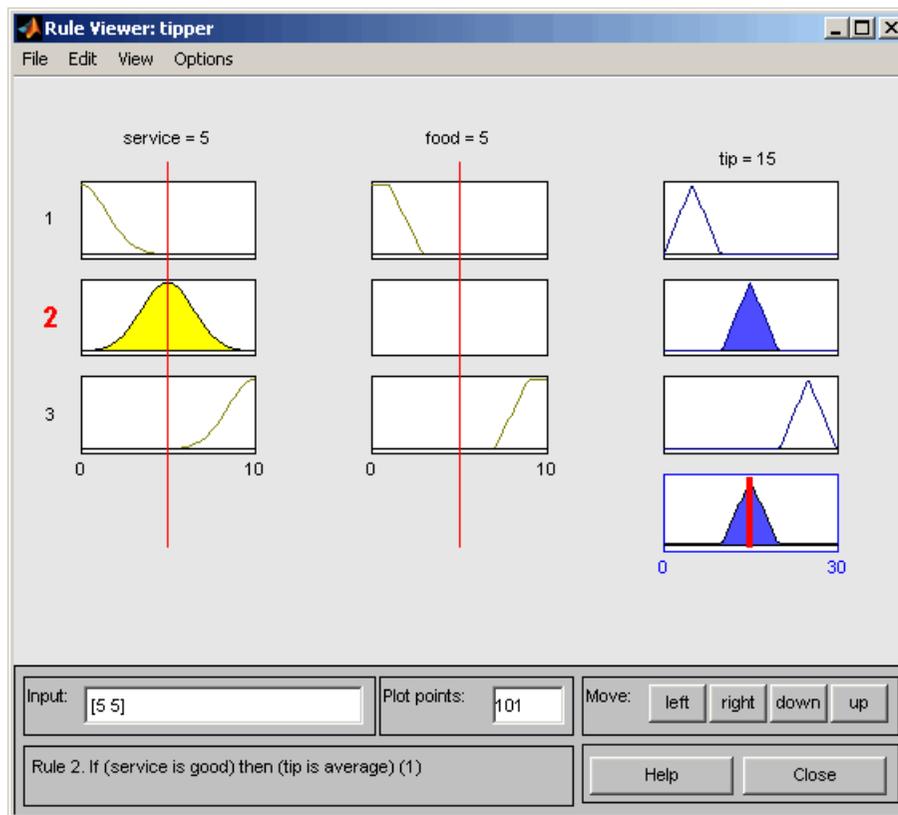
- The first column in this structure corresponds to the input variables.
- The second column corresponds to the output variable.
- The third column displays the weight applied to each rule.
- The fourth column is shorthand that indicates whether this is an OR (2) rule or an AND (1) rule.
- The numbers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

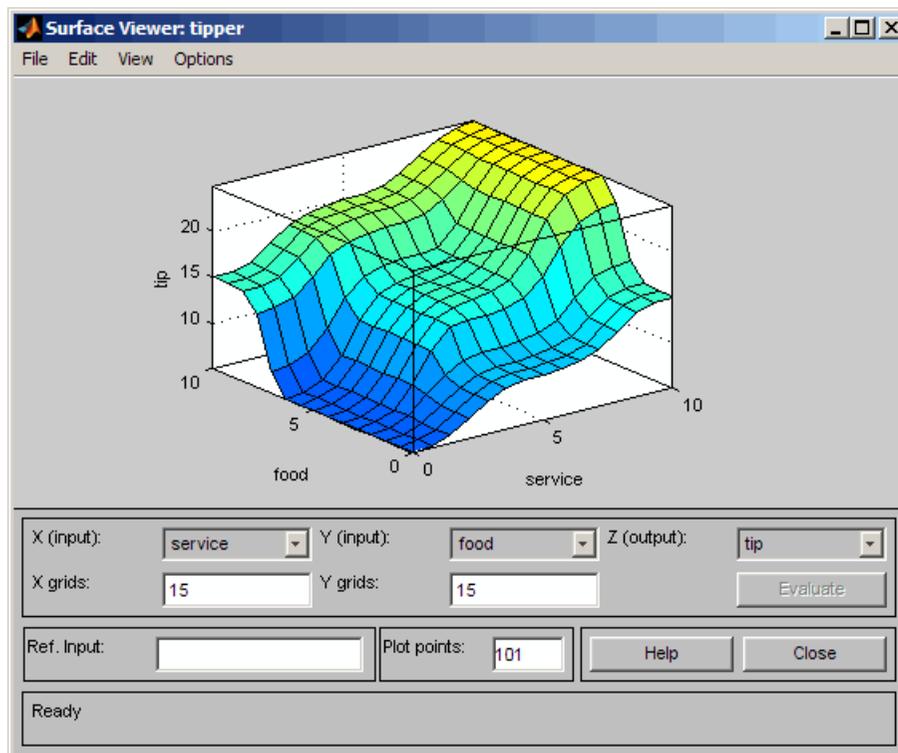
You can shift the plots using **left**, **right**, **down**, and **up**. The menu items allow you to save, open, or edit a fuzzy system using any of the five basic UI tools.

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.

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.

Clicking **Evaluate** initiates the calculation, and the plot is generated after the calculation is complete. To change the x-axis or y-axis grid after the surface is in view, change the appropriate input field, and press **Enter**. The surface plot is updated to reflect the new grid settings.

The Surface Viewer has a special capability that is very helpful in cases with two (or more) inputs and one output: you can grab the axes, using the mouse and reposition them to get a different three-dimensional view on the data.

The **Ref. Input** field is used in situations when there are more inputs required by the system than the surface is mapping. 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. A NaN is the IEEE[®] symbol for Not a Number.

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**.

This concludes the quick walk-through of each of the main UI tools. For the tipping problem, the output of the fuzzy system matches your original idea of the shape of the

fuzzy mapping from service to tip fairly well. In hindsight, you might say, “Why bother? I could have just drawn a quick lookup table and been done an hour ago!” However, if you are interested in solving an entire class of similar decision-making problems, fuzzy logic may provide an appropriate tool for the solution, given its ease with which a system can be quickly modified.

Importing and Exporting 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`. This text file can be edited and modified and is simple to understand. When you save your fuzzy system to the MATLAB workspace, you are creating a variable (whose name you choose) that acts as a MATLAB structure for the FIS system. FIS files and FIS structures represent the same system.

Note If you do not save your FIS to 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 2-30
- “Build Mamdani Systems at the Command Line” on page 2-69
- “Simulate Fuzzy Inference Systems in Simulink” on page 2-81

Build Mamdani Systems Using Custom Functions

In this section...

“Build Fuzzy Inference Systems Using Custom Functions in Fuzzy Logic Designer” on page 2-55

“Specifying Custom Membership Functions” on page 2-56

“Specifying Custom Inference Functions” on page 2-62

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 2-22, you can replace the built-in membership functions or inference functions, or both with custom functions. In this section, you learn how to build a fuzzy inference system using custom functions in the UI.

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

- 1 Open the **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 2-34.
- 3 Create custom membership functions, and replace the built-in membership functions with them, as described in “Specifying Custom Membership Functions” on page 2-56.

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

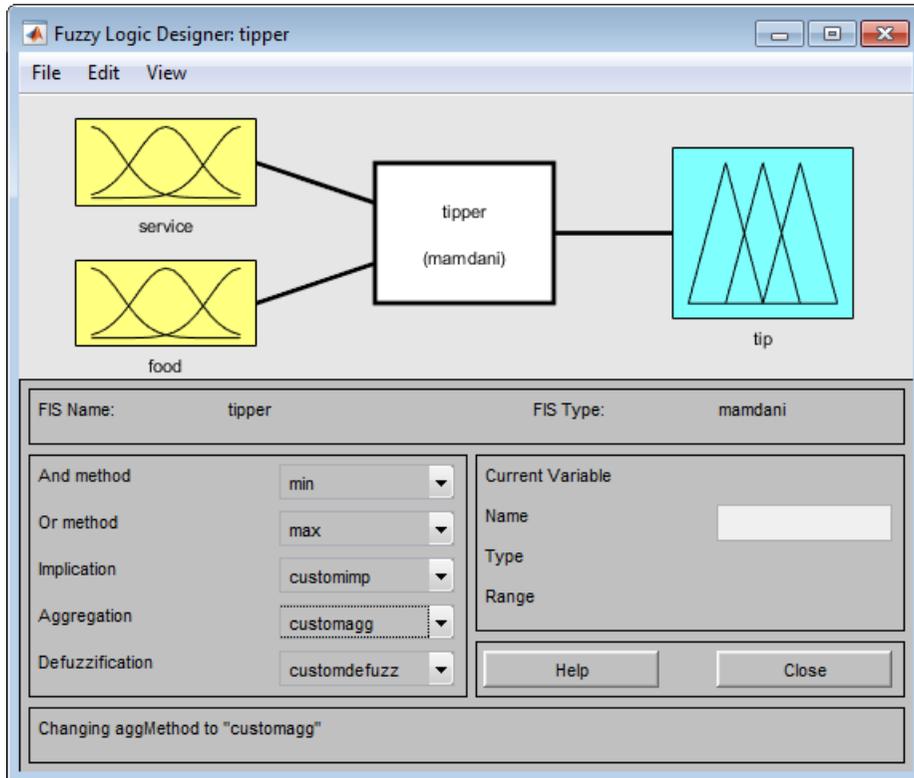
- 4 Create rules using the Rule Editor, as described in “The Rule Editor” on page 2-47.

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 “Specifying Custom Inference Functions” on page 2-62.

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 2-52.

Specifying 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. Save the custom membership functions in your current working folder. To learn how to build fuzzy systems using custom membership functions, see “Build Fuzzy Inference Systems Using Custom Functions in Fuzzy Logic Designer” on page 2-55.

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'; % scaling the output to lie between 0 and 1
```

Note: Custom membership functions can include a maximum of 16 parameters for the `params` input argument.

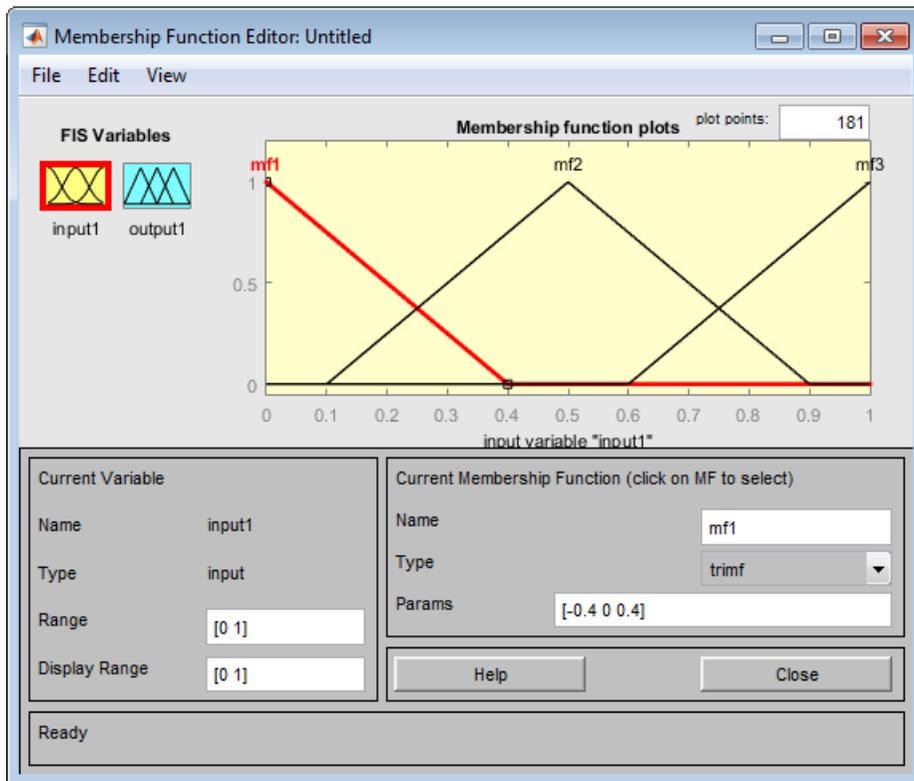
- 2 Open the **Fuzzy Logic Designer** by typing the following at the MATLAB prompt if you have not done so already:

```
fuzzyLogicDesigner
```

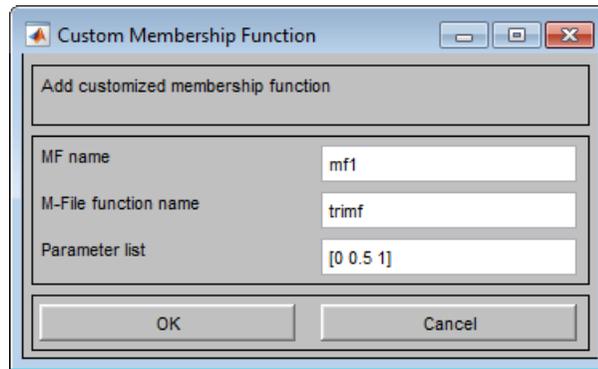
The **Fuzzy Logic Designer** opens with the default FIS name, **Untitled**, and contains one input, **input1**, and one output, **output1**.

- 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:
- Specify a name for the custom membership function in the **MF name** field.

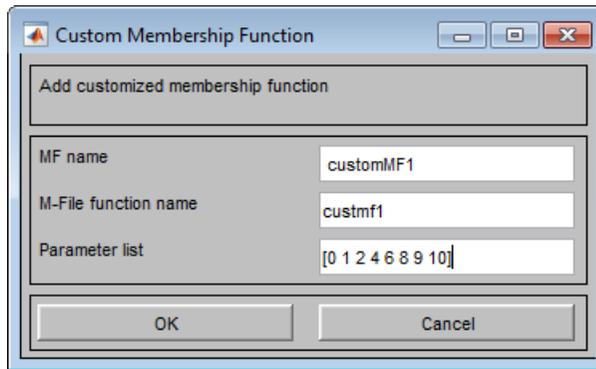
Note: When adding additional custom membership functions, specify a different **MF name** for each function.

- Specify the name of the custom membership function file in the **M-file function name** field.
- Specify a vector of parameters in the **Parameter list** field.

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

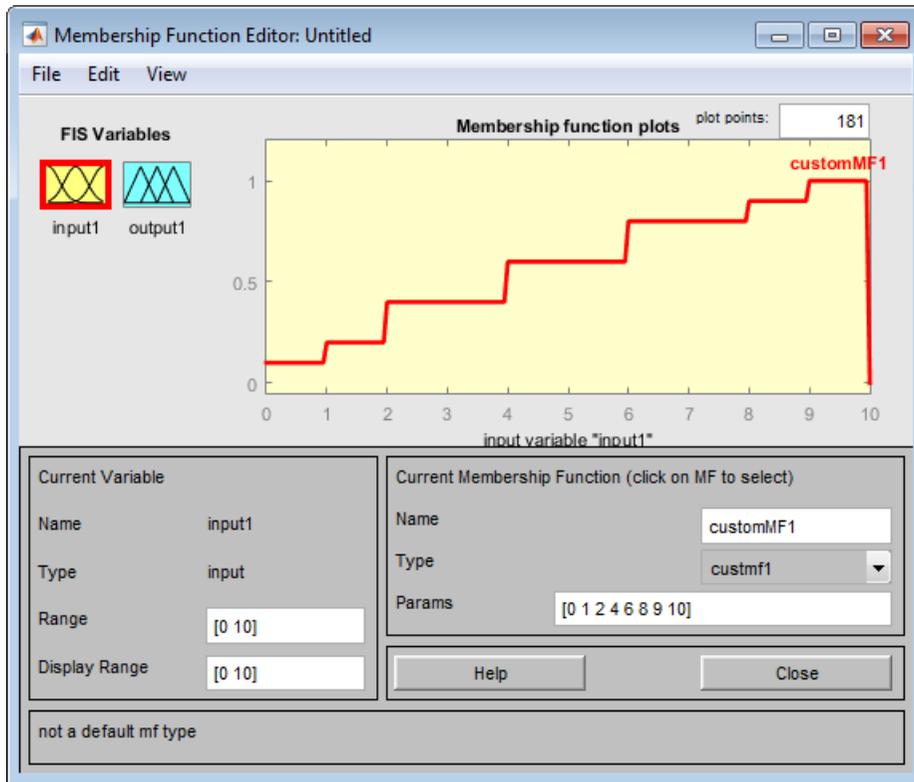
Note: The length of the parameter vector must be greater than or equal to the number of parameters in the custom membership function.

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

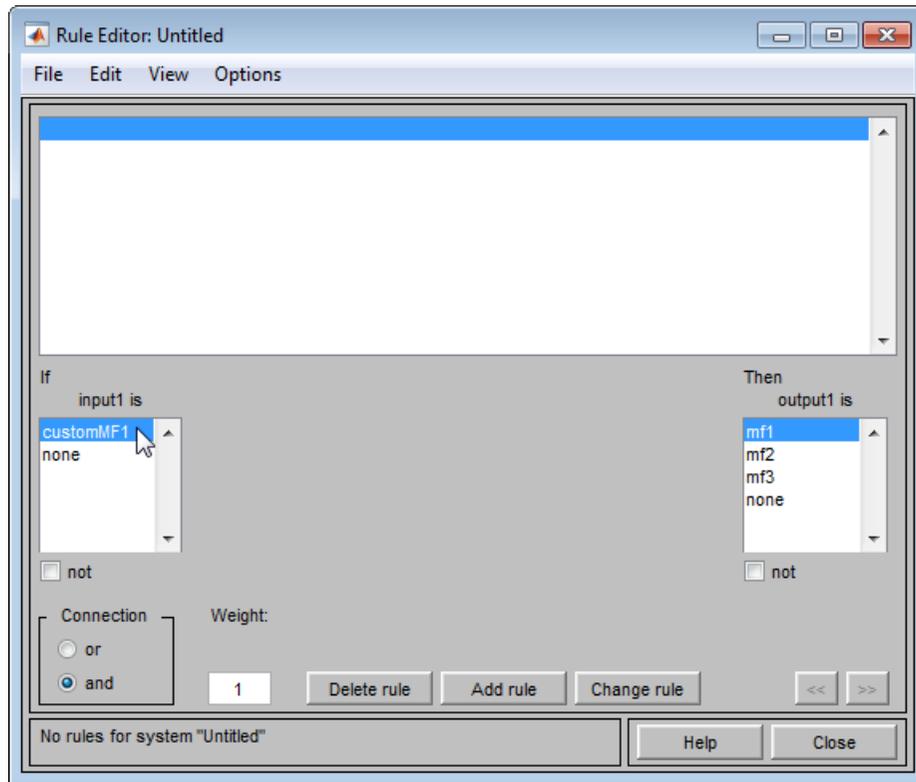


- d** Click **OK** to add the custom membership function.
- e** Specify both the **Range** and **Display Range** to be [0 10] to match the range of the custom membership function.

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 an FIS at the MATLAB command line. For example, to add `custmf1` to the first input variable of the FIS, `a`, type the following:

```
a = addmf(a, 'input', 1, 'customMF1', custmf1', [0 1 2 4 6 8 9 10]);
```

Specifying 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 2-55 section.

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

- “Creating Custom AND and OR Functions” on page 2-63
- “Creating Custom Implication Functions” on page 2-64
- “Creating Custom Aggregation Functions” on page 2-65
- “Guidelines for Creating Custom Defuzzification Functions” on page 2-65
- “Steps for Specifying Custom Inference Functions” on page 2-65

Creating 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 matrix `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)`.

Creating Custom Implication Functions

Custom implication functions must operate in the same way as the MATLAB functions `max`, `min`, or `prod`.

The following is an example of a custom implication function:

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

if nargin==1
    y = prod(x1).^2;
else
    y = (x1.*x2).^2;
end
```

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.

Creating 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 $y = \text{customagg}(x)$.

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 custom aggregation function:

```
function aggfun = customagg(x)
aggfun = (sum(x)/2).^0.5;
```

Guidelines for Creating 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 .

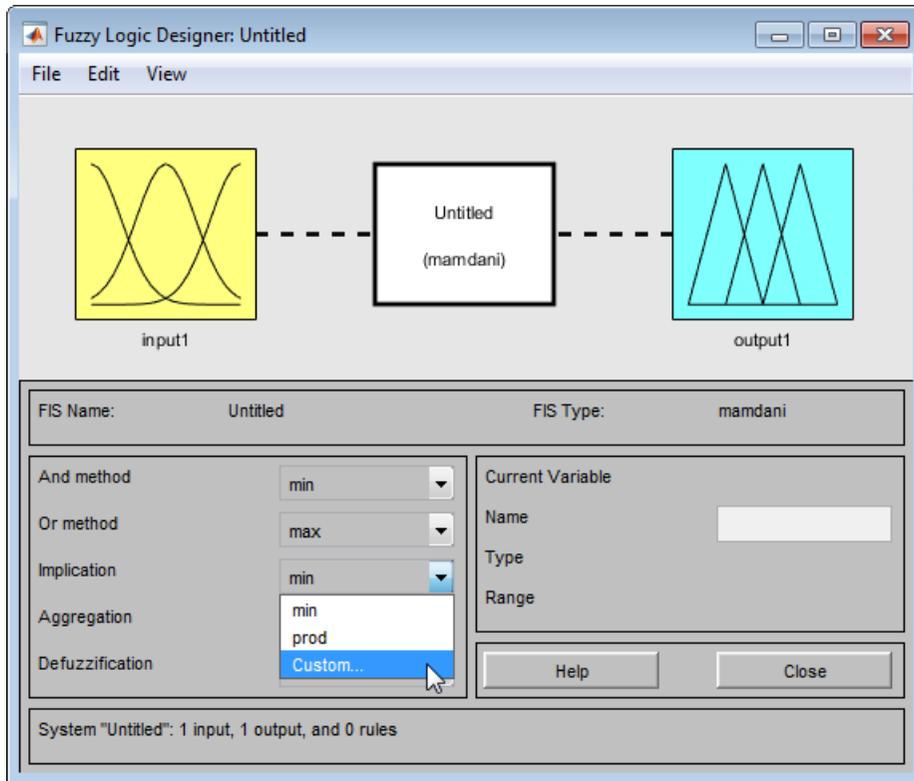
The following is an example of a custom defuzzification function:

```
function defuzzfun = customdefuzz(xmf, ymf)
total_area = sum(ymf);
defuzzfun = sum(ymf.*xmf)/total_area;
```

Steps for Specifying Custom Inference Functions

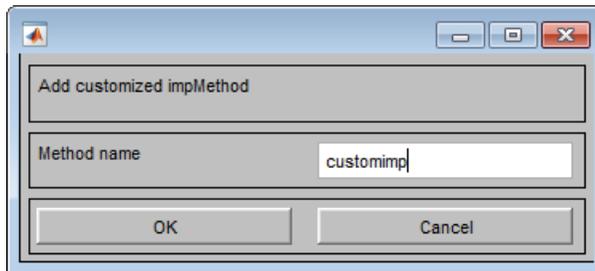
After you create and save a custom inference function, use the following steps to specify the function in the fuzzy inference process:

- 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.



This action 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 an FIS at the MATLAB command line. For example, to add a custom:

- Defuzzification method, type

```
a.defuzzMethod = 'customdefuzz';
```

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

- Implication method, type

```
a.impMethod = 'customimp';
```

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

- Aggregation method, type

```
a.aggMethod = 'customagg';
```

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

See Also

Fuzzy Logic Designer

Related Examples

- “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31
- “Build Mamdani Systems at the Command Line” on page 2-69

Build Mamdani Systems at the Command Line

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

Tipping 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')

fis =

    struct with fields:

        name: 'tipper'
        type: 'mamdani'
        andMethod: 'min'
        orMethod: 'max'
        defuzzMethod: 'centroid'
        impMethod: 'min'
        aggMethod: 'max'
        input: [1×2 struct]
        output: [1×1 struct]
        rule: [1×3 struct]
```

This command returns a MATLAB structure that contains the properties of the fuzzy system.

The `getfis` command displays a list of fuzzy system properties, some of which are not fields in the FIS structure.

```
getfis(fis)

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
```

To access these properties, use the `getfis` command, specifying the property name.

```
getfis(fis, 'InLabels')
```

```
ans =
```

```
2x7 char array
```

```
'service'
'food   '
```

Similarly, you can also obtain the FIS structure information using `getfis`.

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

```
ans =
```

```
struct with fields:
```

```
    Name: 'tip'
  NumMFs: 3
    mf1: 'cheap'
    mf2: 'average'
    mf3: 'generous'
   range: [0 30]
```

You can also access FIS properties using dot notation. For example, to view the `type` of the fuzzy system, type:

```
fis.type
```

```
ans =  
    'mamdani'
```

To set the properties of your fuzzy system, use the `setfis` command. For example, to change the name of the FIS, type:

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

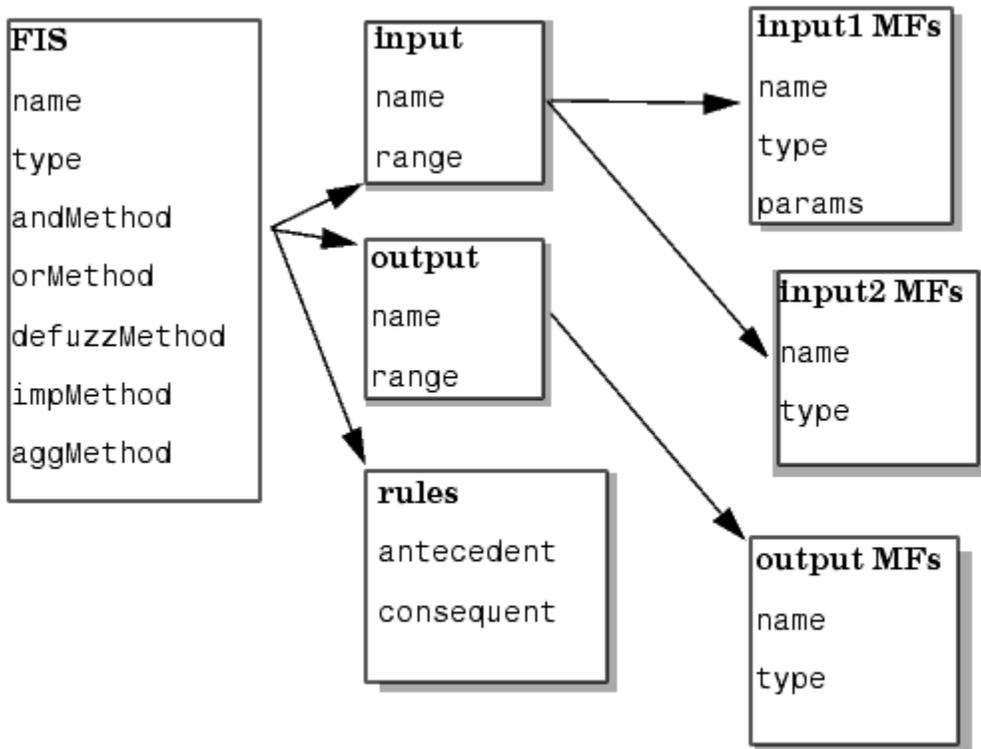
```
fis =  
    struct with fields:  
        name: 'gratuity'  
        type: 'mamdani'  
        andMethod: 'min'  
        orMethod: 'max'  
        defuzzMethod: 'centroid'  
        impMethod: 'min'  
        aggMethod: 'max'  
        input: [1×2 struct]  
        output: [1×1 struct]  
        rule: [1×3 struct]
```

Alternatively, you can use dot notation. For example:

```
fis.name = 'gratuity';
```

FIS Structure

The FIS structure contains all the fuzzy inference system information, including the variable names, membership function definitions, and fuzzy inference methods. This structure is itself a hierarchy of structures, as shown in the following:



To view all the information about an FIS, use the `showfis` command.

```
showfis(fis)
```

```

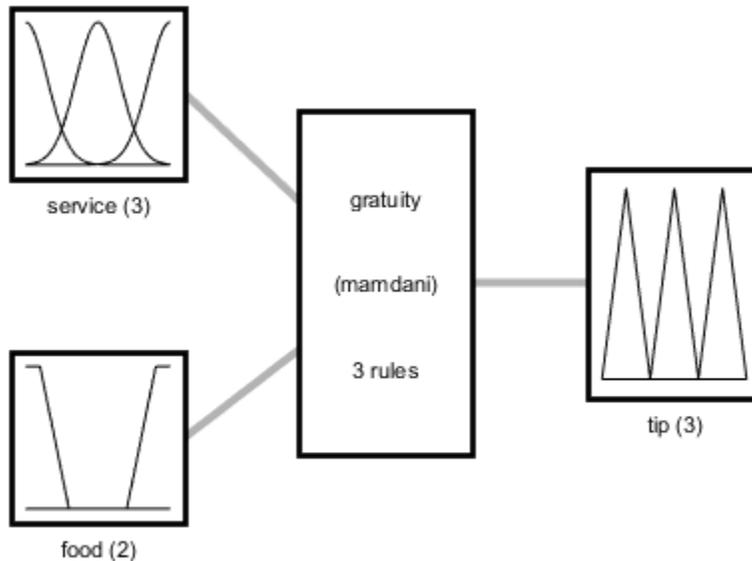
1. Name          gratuity
2. Type          mamdani
3. Inputs/Outputs [2 1]
4. NumInputMFs   [3 2]
5. NumOutputMFs  3
6. NumRules      3
7. AndMethod     min
8. OrMethod      max
9. ImpMethod     min
10. AggMethod    max
11. DefuzzMethod centroid
12. InLabels     service
  
```

```
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]
45. Rule Consequent 1
46.          2
47.          3
48. Rule Weight     1
49.          1
50.          1
51. Rule Connection 2
52.          1
53.          2
```

System Display Functions

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

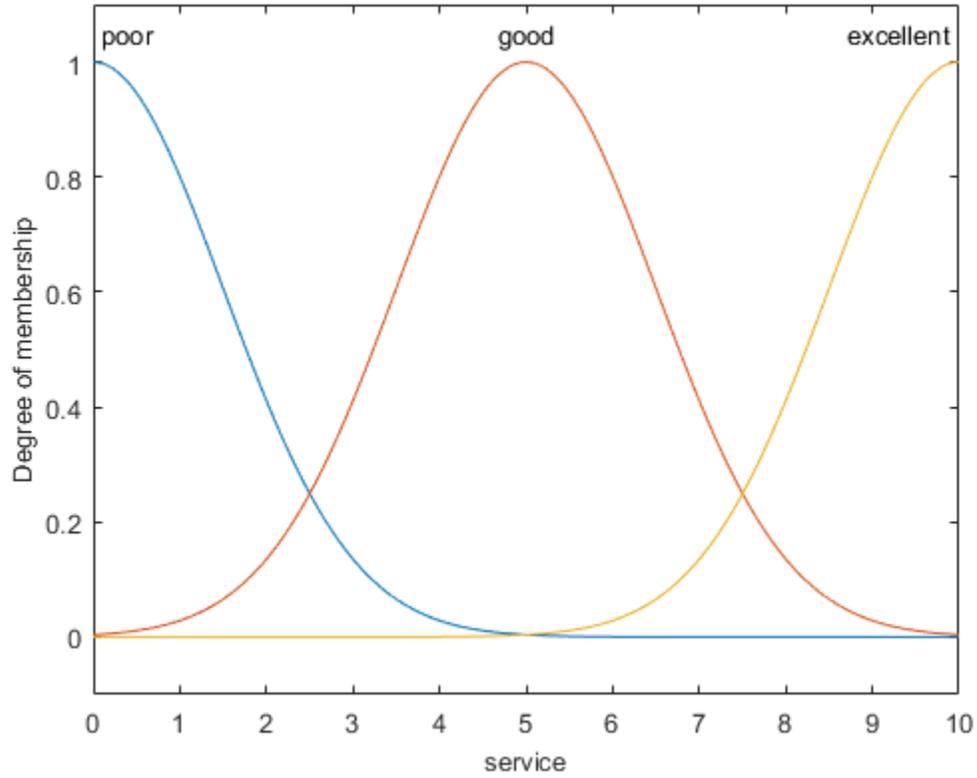
```
plotfis(fis)
```



System gratuity: 2 inputs, 1 outputs, 3 rules

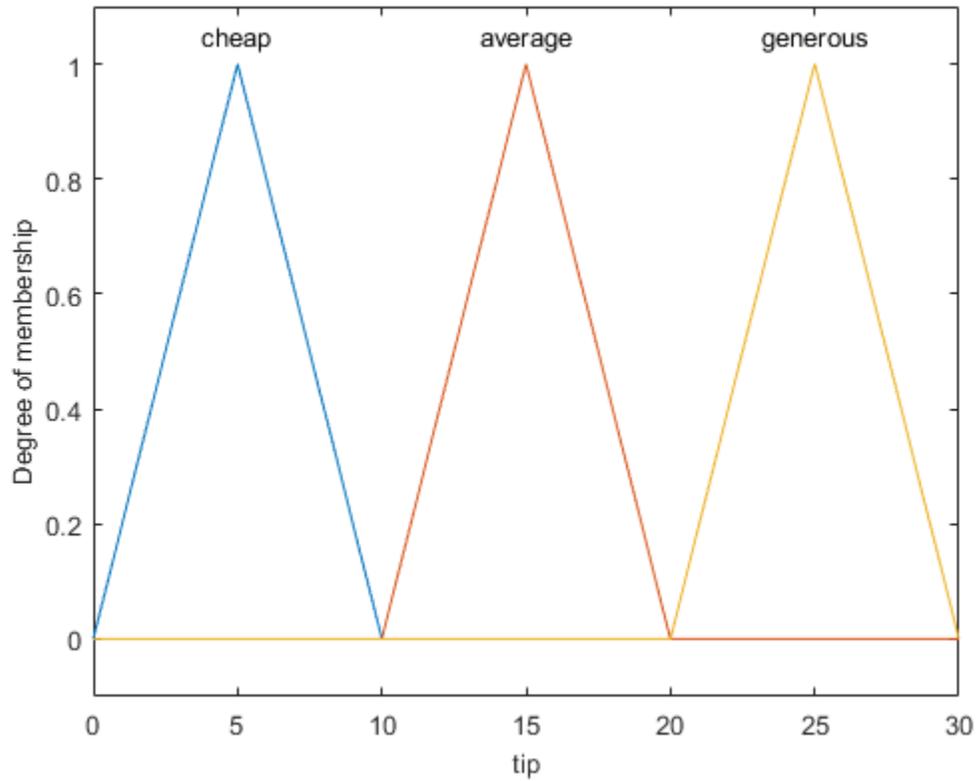
The `plotmf` command plots all the memberships functions associated with a given variable. For example, to view the membership functions for the first input variable, type:

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



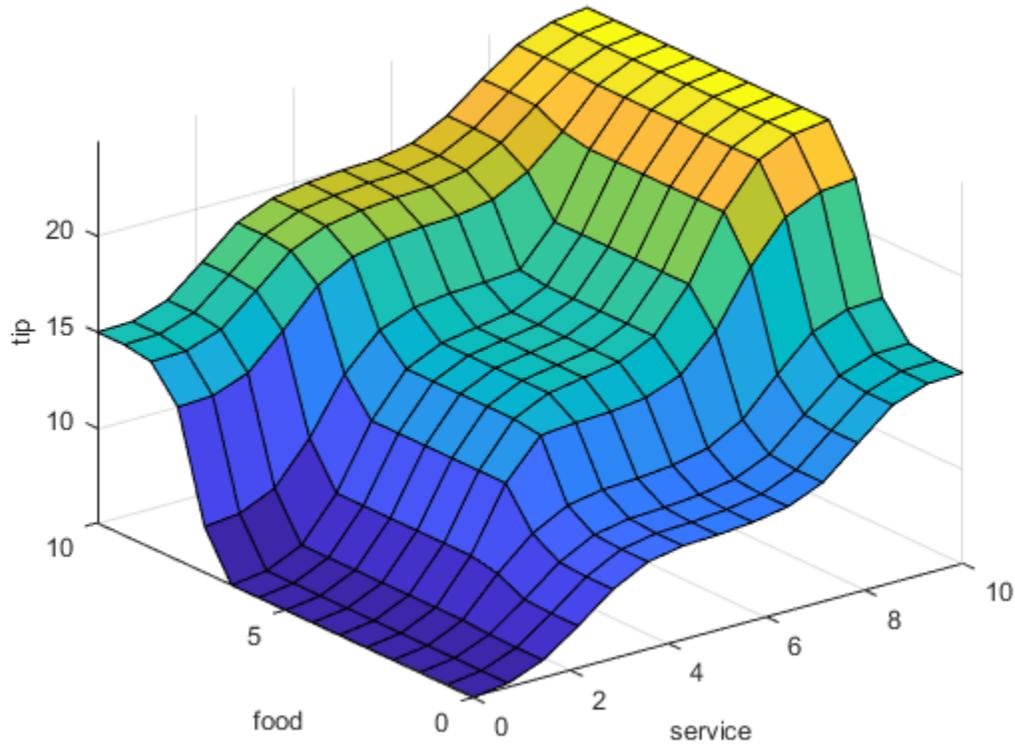
Similarly, to view the membership functions for the first output, type:

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



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

```
gensurf(fis)
```



Build FIS from Scratch

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

First, create an FIS, specifying its name.

```
fis = newfis('tipper');
```

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

```
fis = addvar(fis, 'input', 'service', [0 10]);
```

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, 'input', 1, 'poor', 'gaussmf', [1.5 0]);  
fis = addmf(fis, 'input', 1, 'good', 'gaussmf', [1.5 5]);  
fis = addmf(fis, 'input', 1, 'excellent', 'gaussmf', [1.5 10]);
```

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

```
fis = addvar(fis, 'input', 'food', [0 10]);  
fis = addmf(fis, 'input', 2, 'rancid', 'trapmf', [-2 0 1 3]);  
fis = addmf(fis, 'input', 2, 'delicious', 'trapmf', [7 9 10 12]);
```

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

```
fis = addvar(fis, 'output', 'tip', [0 30]);  
fis = addmf(fis, 'output', 1, 'cheap', 'trimf', [0 5 10]);  
fis = addmf(fis, 'output', 1, 'average', 'trimf', [10 15 20]);  
fis = addmf(fis, 'output', 1, 'generous', 'trimf', [20 25 30]);
```

Specify the following three rules for the FIS as an 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
- Column 5 - Fuzzy operator (1 for AND, 2 for OR)

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 dot notation.

```

fis = newfis('tipper');
fis.input(1).name = 'service';
fis.input(1).range = [0 10];
fis.input(1).mf(1).name = 'poor';
fis.input(1).mf(1).type = 'gaussmf';
fis.input(1).mf(1).params = [1.5 0];
fis.input(1).mf(2).name = 'good';
fis.input(1).mf(2).type = 'gaussmf';
fis.input(1).mf(2).params = [1.5 5];
fis.input(1).mf(3).name = 'excellent';
fis.input(1).mf(3).type = 'gaussmf';
fis.input(1).mf(3).params = [1.5 10];
fis.input(2).name = 'food';
fis.input(2).range = [0 10];
fis.input(2).mf(1).name = 'rancid';
fis.input(2).mf(1).type = 'trapmf';
fis.input(2).mf(1).params = [-2 0 1 3];
fis.input(2).mf(2).name = 'delicious';
fis.input(2).mf(2).type = 'trapmf';
fis.input(2).mf(2).params = [7 9 10 12];
fis.output(1).name = 'tip';
fis.output(1).range = [0 30];
fis.output(1).mf(1).name = 'cheap';
fis.output(1).mf(1).type = 'trimf';
fis.output(1).mf(1).params = [0 5 10];
fis.output(1).mf(2).name = 'average';
fis.output(1).mf(2).type = 'trimf';
fis.output(1).mf(2).params = [10 15 20];
fis.output(1).mf(3).name = 'generous';
fis.output(1).mf(3).type = 'trimf';
fis.output(1).mf(3).params = [20 25 30];
fis.rule(1).antecedent = [1 1];
fis.rule(1).consequent = 1;
fis.rule(1).weight = 1;
fis.rule(1).connection = 2;
fis.rule(2).antecedent = [2 0];
fis.rule(2).consequent = 2;
fis.rule(2).weight = 1;
fis.rule(2).connection = 1;

```

```
fis.rule(3).antecedent = [3 2];
fis.rule(3).consequent = 3;
fis.rule(3).weight = 1;
fis.rule(3).connection = 2;
```

When constructing your fuzzy system, you can also specify custom membership functions and inference functions. For more information, see “Specifying Custom Membership Functions” and “Specifying Custom Inference Functions”.

FIS Evaluation

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

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

```
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(inputs,fis)
```

```
ans =
    12.2184
     7.7885
     8.9547
```

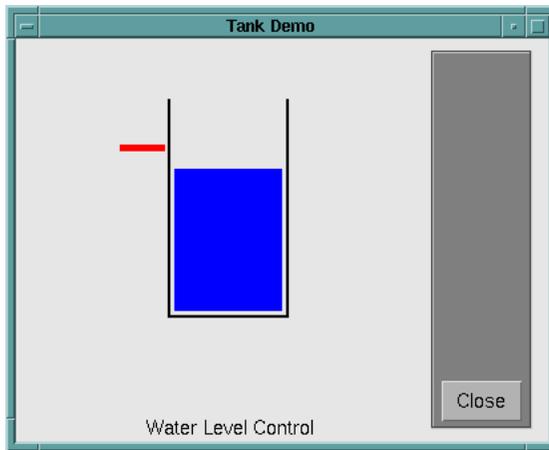
More About

- “What Is Mamdani-Type Fuzzy Inference?” on page 2-30
- “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31

Simulate Fuzzy Inference Systems in Simulink

Fuzzy Logic Toolbox software is designed to work in Simulink environment. After you have created your fuzzy system using the UI tools or some other method, you are ready to embed your system directly into a simulation.

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



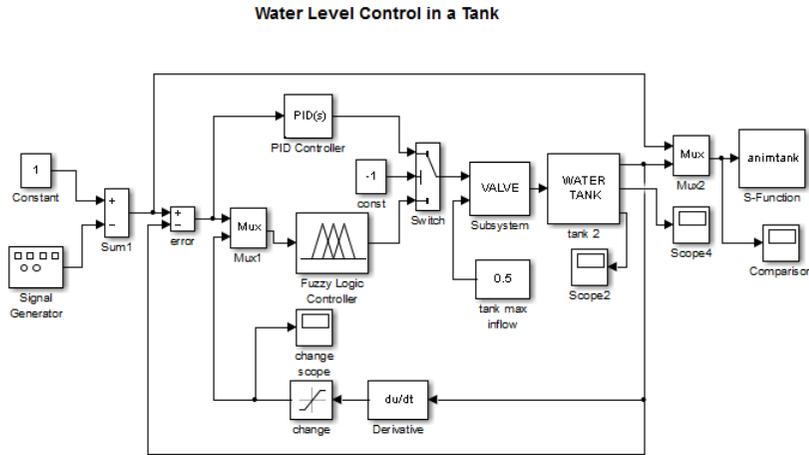
A controller for the water level in the tank needs to know the current water level and it needs to be able to set the valve. Your controller's input is the water level error (desired water level minus actual water level), and its output is the rate at which the valve is opening or closing. A first pass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

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

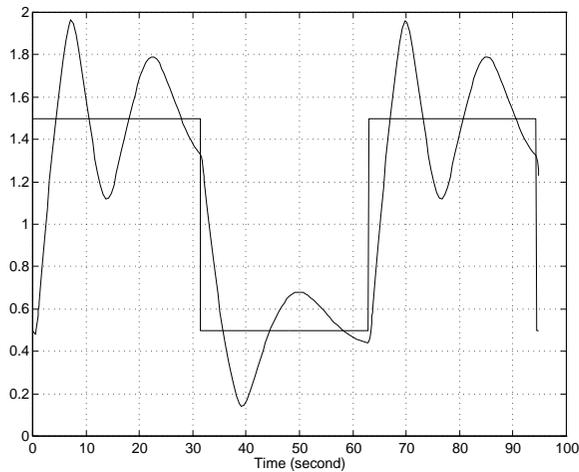
You can take fuzzy systems directly into Simulink and test them out in a simulation environment. A Simulink block diagram for this system is shown in the following figure. It contains a Simulink block called the Fuzzy Logic Controller block. The Simulink block diagram for this system is `sltank`. Typing

sltank

at the command line, causes the system to appear. At the same time, the file tank.fis is loaded into the FIS structure tank.



Some experimentation shows that three rules are not sufficient, because the water level tends to oscillate around the desired level. See the following plot:

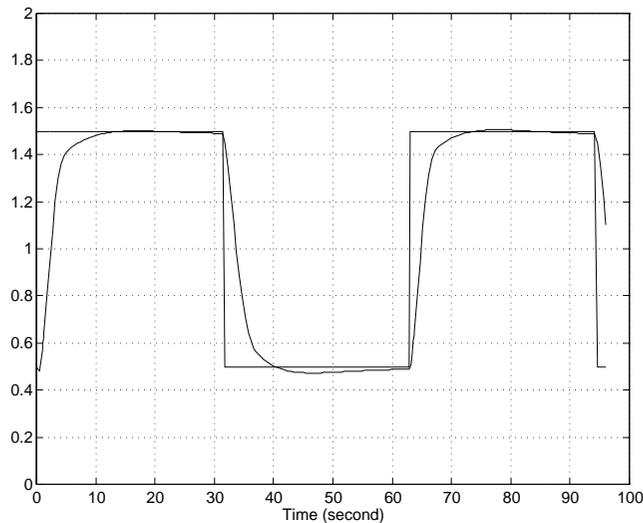


You need to add another input, the water level's rate of change, to slow down the valve movement when it gets close to the right level.

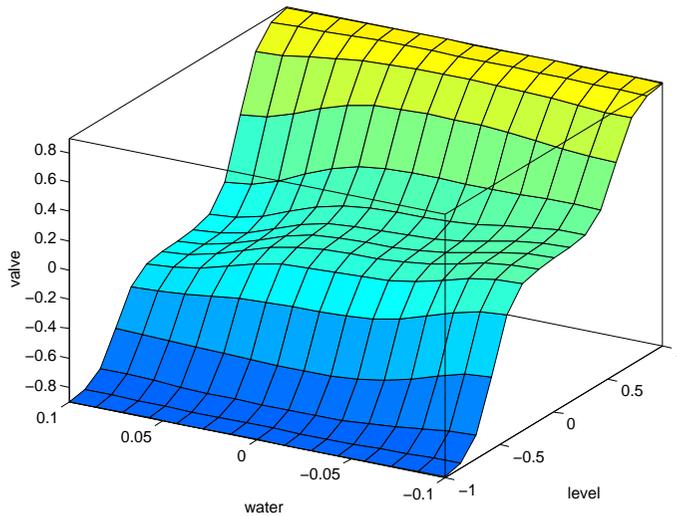
4. *If (level is good) and (rate is negative), then (valve is close_slow) (1)*

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

`sltank` is built with these five rules. With all five rules in operations, you can examine the step response by simulating this system. You do so by clicking **Simulation > Run**, and clicking the Comparison block. The result looks similar to the following plot.



One interesting feature of the water tank system is that the tank empties much more slowly than it fills up because of the specific value of the outflow diameter pipe. You can deal with this by setting the `close_slow` valve membership function to be slightly different from the `open_slow` setting. A PID controller does not have this capability. The valve command versus the water level change rate (depicted as *water*) and the relative water level change (depicted as *level*) surface looks like this. If you look closely, you can see a slight asymmetry to the plot.



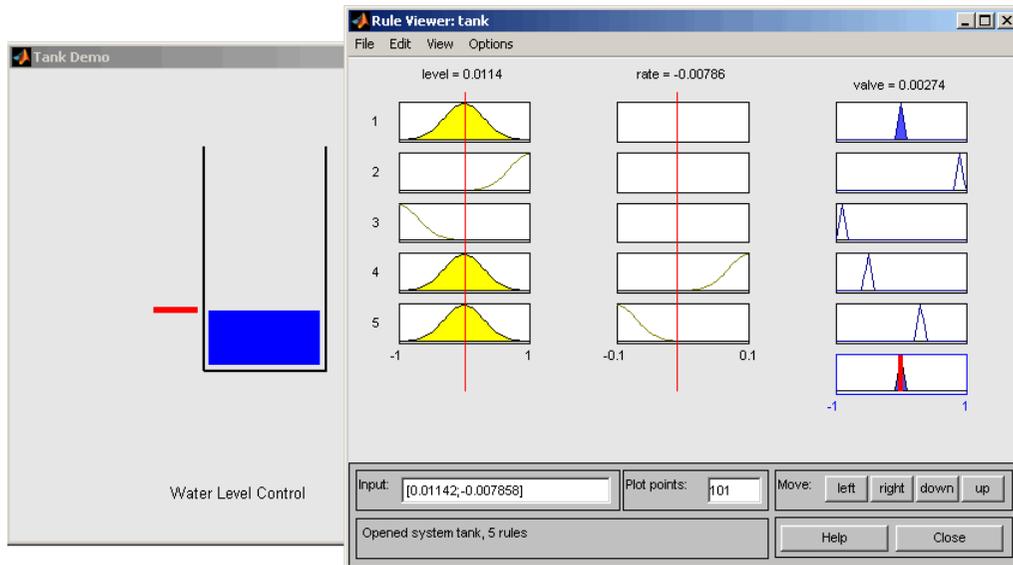
Because MATLAB software supports so many tools such as Control System Toolbox, and Neural Network Toolbox software, you can, for example, easily make a comparison of a fuzzy controller versus a linear controller or a neural network controller.

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

```
sltankrule
```

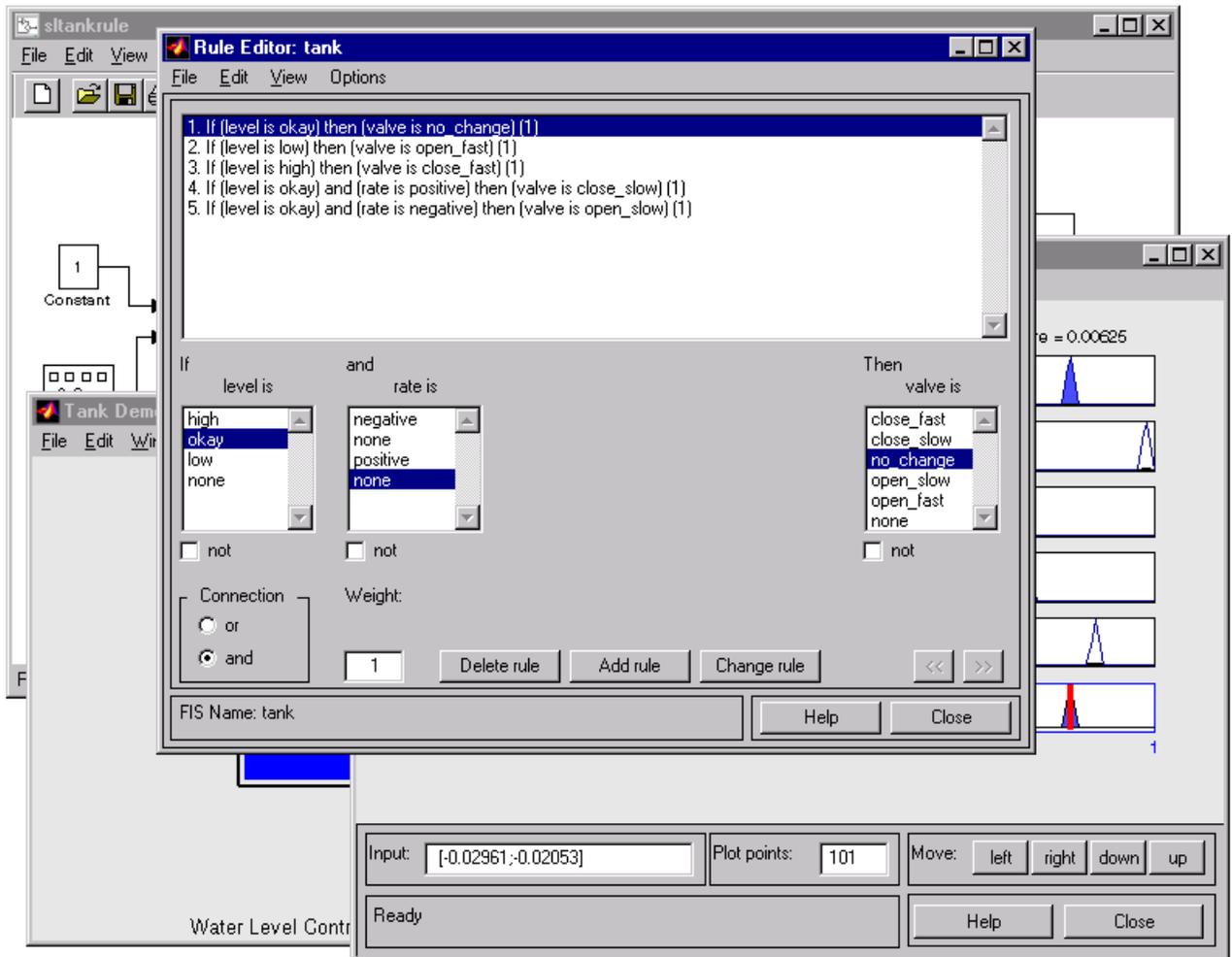
This example contains a block called the Fuzzy Controller With Rule Viewer block.

In this example, the Rule Viewer opens when you start the simulation. This Rule Viewer provides an animation of how the rules are fired during the water tank simulation. The windows that open when you simulate `sltankrule` are depicted as follows.



The Rule Viewer that opens during the simulation can be used to access the Membership Function Editor, the Rule Editor, or any of the other UIs, (see “The Membership Function Editor” on page 2-39, or “The Rule Editor” on page 2-47, for more information).

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



If you stop the simulation prior to selecting any of these editors, you should change your FIS. Remember to save any changes you make to your FIS to the workspace before you restart the simulation.

See Also

Fuzzy Logic Controller | Fuzzy Logic Controller with Ruleviewer

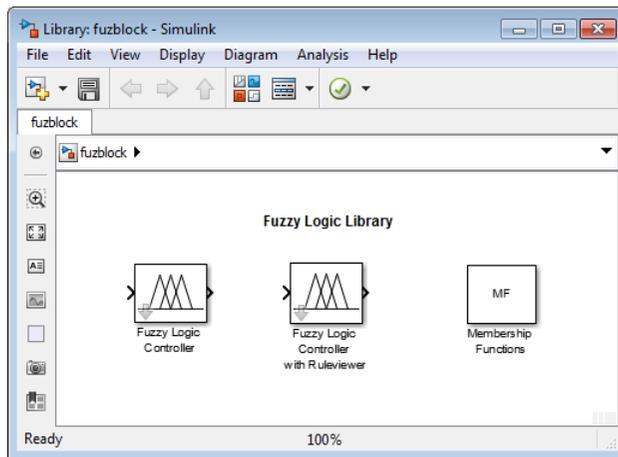
More About

- “Build Your Own Fuzzy Simulink Models” on page 2-88

Build Your Own Fuzzy Simulink Models

To build your own Simulink systems that use fuzzy logic, use the blocks in the Fuzzy Logic Toolbox library. To open the library, open the Simulink Library Browser. In the left pane, select **Fuzzy Logic Toolbox**. Alternatively enter the following at the MATLAB command line:

```
fuzblock
```



The Fuzzy Logic Toolbox library contains the Fuzzy Logic Controller and Fuzzy Logic Controller with Rule Viewer blocks. It also includes a Membership Functions sublibrary that contains Simulink blocks for the built-in membership functions.

To add a block from the library, drag the block into the Simulink model window. You can get help on a specific block by clicking **Help**.

About the Fuzzy Logic Controller Block

For most fuzzy inference systems, the Fuzzy Logic Controller block automatically generates a hierarchical block diagram representation of your FIS. This automatic model generation ability is called the *Fuzzy Wizard*. The block diagram representation only uses built-in Simulink blocks and, therefore, allows for efficient code generation.

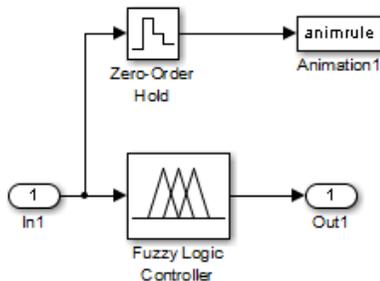
The Fuzzy Wizard cannot handle an FIS with custom membership functions or with AND, OR, IMP, and AGG functions outside of the following list:

- orMethod: max
- andMethod: min,prod
- impMethod: min,prod
- aggMethod: max

In these cases, the Fuzzy Logic Controller block uses the S-function `sffis` to simulate the FIS.

About the Fuzzy Logic Controller with Ruleviewer Block

The Fuzzy Logic Controller with Rule Viewer block is an extension of the Fuzzy Logic Controller block. It allows you to visualize how rules are fired during simulation. Right-click the Fuzzy Controller With Rule Viewer block, and select **Mask > Look Under Mask** to view the underlying subsystem.

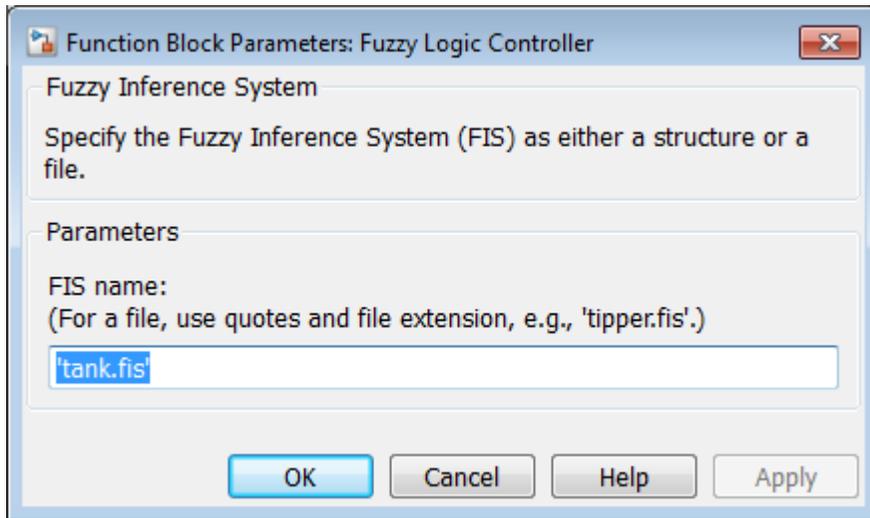


Initializing Fuzzy Logic Controller Blocks

You can initialize a Fuzzy Logic Controller or Fuzzy Logic Controller with Ruleviewer block using a fuzzy inference system saved as a `.fis` file or a structure. To learn how to save your fuzzy inference system, see “Importing and Exporting Fuzzy Inference Systems” on page 2-54.

To initialize a Fuzzy Logic Controller block, use the following steps:

- 1 Double-click the block to open the **Function Block Parameters: Fuzzy Logic Controller** dialog box.
- 2 In **FIS name**, enter the name of the structure variable or the name of the `.fis` file.



Note: When entering the name of the `.fis` file in the blocks, you must enclose it in single quotes.

You can similarly initialize the Fuzzy Logic Controller with Ruleviewer block. In this case, enter the name of the structure variable in **FIS matrix**.

Example: Cart and Pole Simulation

The cart and pole simulation is an example of a FIS model auto-generated by the Fuzzy Logic Controller block.

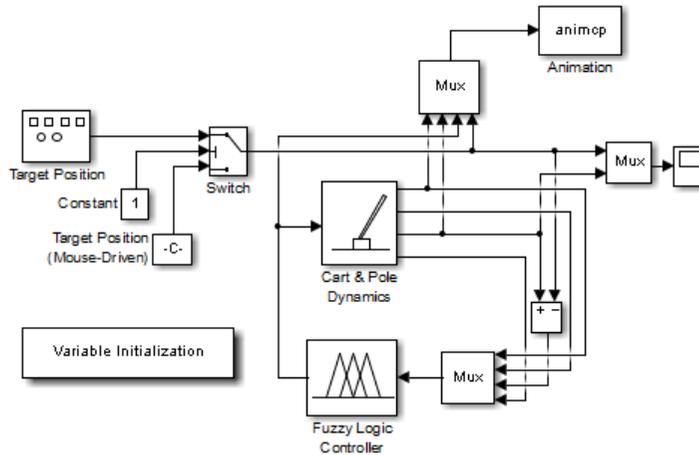
Type

```
s1cp
```

at the MATLAB prompt to open the simulation.

This model appears.

Cart and Pole System



Right-click on the Fuzzy Logic Controller block, and select **Mask > Look Under Mask** from the right-click menu. The following subsystem opens.



Follow the same procedure to look under the mask of the FIS Wizard subsystem to see the implementation of your FIS. The Fuzzy Logic Controller block uses built-in Simulink blocks to implement your FIS. Although the models can grow complex, this representation is better suited than the S-function `sffis` for efficient code generation.

See Also

Fuzzy Logic Controller | Fuzzy Logic Controller with Ruleviewer

More About

- “Simulate Fuzzy Inference Systems in Simulink” on page 2-81

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

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

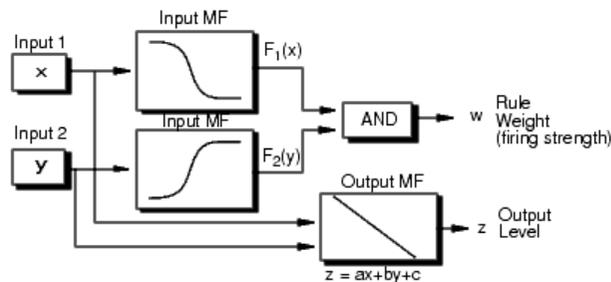
where $F_{1,2}(\cdot)$ are the membership functions for Inputs 1 and 2.

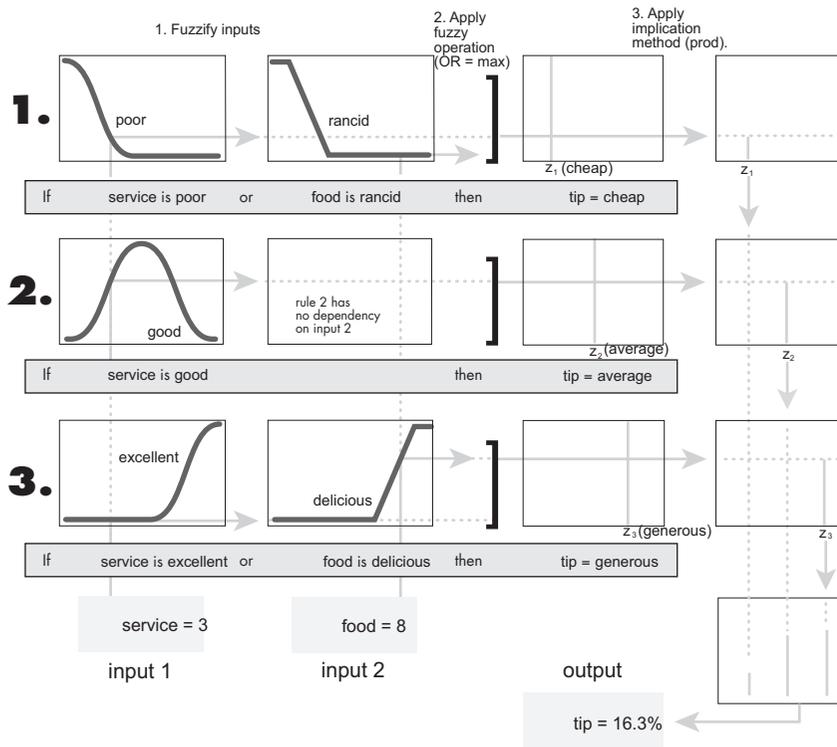
The final output of the system is the weighted average of all rule outputs, computed as

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

where N is the number of rules.

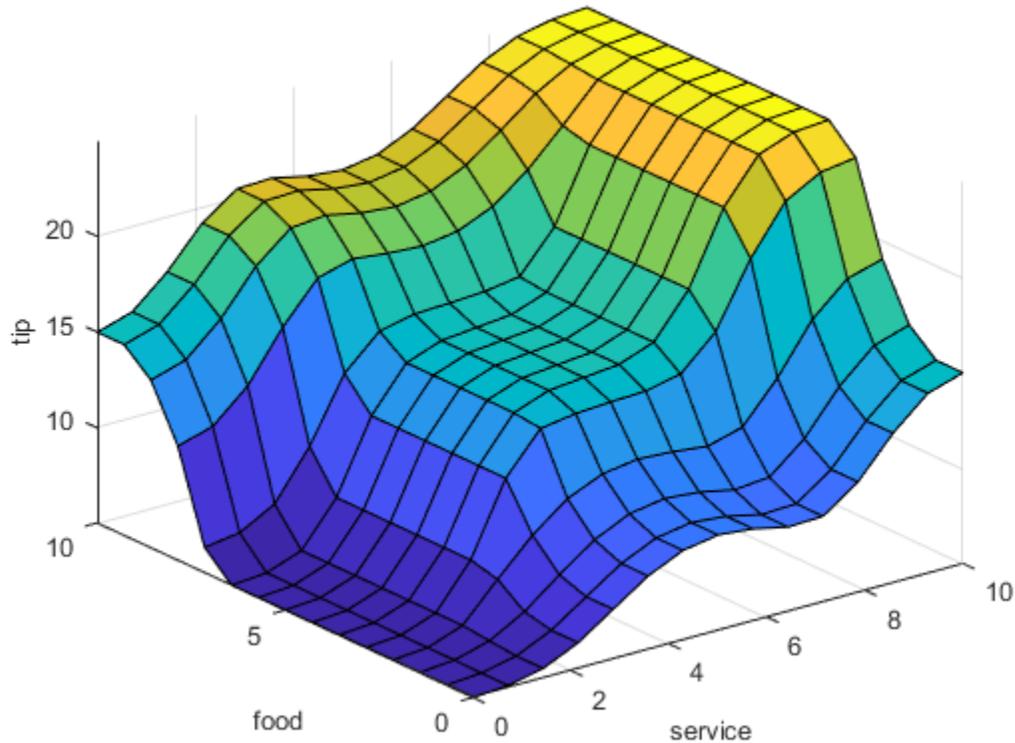
A Sugeno rule operates as shown in the following diagram.





The preceding figure shows the fuzzy tipping model developed in previous sections of this manual 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 `sugeno1.fis`. Load the system and view the properties of its output variable.

```
fismat = readfis('sugeno1');  
getfis(fismat,'output',1)
```

```
ans =
```

```
struct with fields:  
  
    Name: 'output'  
    NumMFs: 2  
    mf1: 'line1'  
    mf2: 'line2'  
    range: [0 1]
```

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

```
getfis(fismat,'output',1,'mf',1)
```

```
ans =
```

```
struct with fields:  
  
    Name: 'line1'  
    Type: 'linear'  
    params: [-1 -1]
```

View the properties of the second membership function.

```
getfis(fismat,'output',1,'mf',2)
```

```
ans =  
  
struct with fields:  
  
    Name: 'line2'  
    Type: 'linear'  
    params: [1 -1]
```

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

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

and the membership function `line2` is:

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

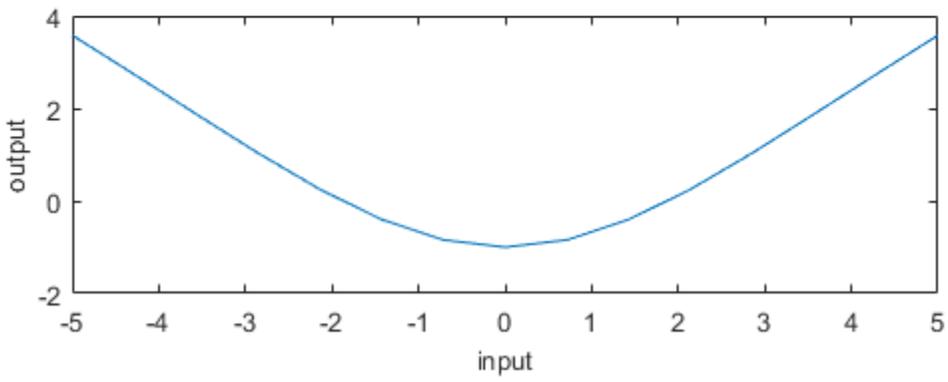
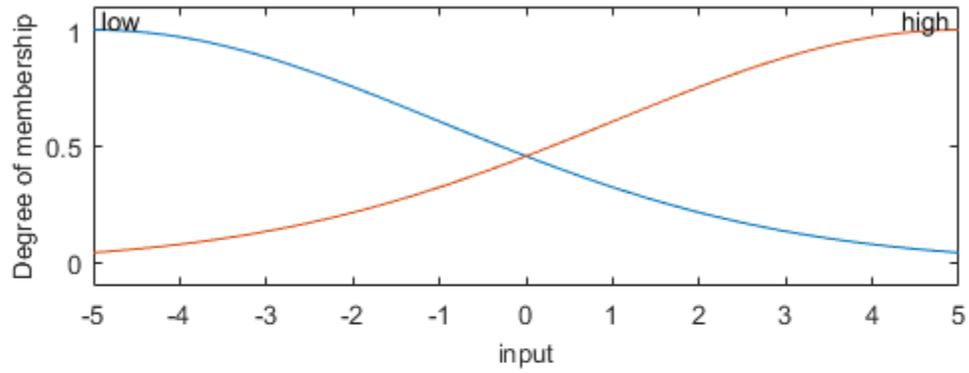
The input membership functions and rules define which of these output functions are expressed and when:

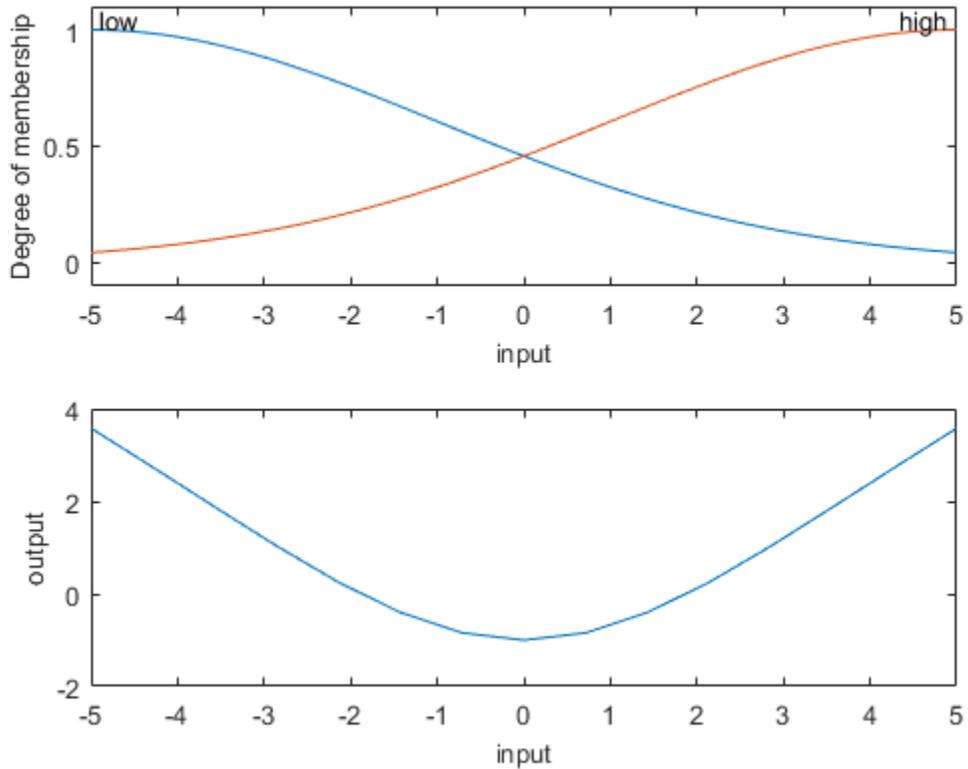
```
showrule(fismat)
```

```
ans =  
  
2×48 char array  
  
'1. If (input is low) then (output is line1) (1) '  
'2. If (input is high) then (output is line2) (1)'
```

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

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





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 2-100
- “What Is Mamdani-Type Fuzzy Inference?” on page 2-30

Comparison of Sugeno and Mamdani Systems

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

Note You can use the MATLAB command-line function `mam2sug` to convert a Mamdani system into a Sugeno system (not necessarily with a single output) with constant output membership functions. It uses the centroid associated with all of the output membership functions of the Mamdani system.

The following are some final considerations about the two different methods.

Advantages of the Sugeno Method

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

Advantages of the Mamdani Method

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

See Also

`mam2sug`

More About

- “What Is Mamdani-Type Fuzzy Inference?” on page 2-30
- “What Is Sugeno-Type Fuzzy Inference?” on page 2-92

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
- “Modeling Inverse Kinematics in a Robotic Arm” on page 3-51
- “Adaptive Noise Cancellation Using ANFIS” on page 3-62

Neuro-Adaptive Learning and ANFIS

In this section...
“When to Use Neuro-Adaptive Learning” on page 3-2
“Model Learning and Inference Through ANFIS” on page 3-3
“References” on page 3-5

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 Structure

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

- The **Fuzzy Logic Designer**
- The Membership Function Editor
- The Rule Editor from the **Neuro-Fuzzy Designer** (which allows an FIS structure 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 structure 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 structure.

The `genfis` function, when used with grid partitioning, produces an FIS structure based on a fixed number of membership functions. This structure 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 structure 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 Structure for Training Data

`fismat1` is the output FIS structure corresponding to the minimum training error. This FIS structure 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 structure 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 structure 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 Structure for Checking Data

The output of the command-line `anfis`, `fismat2`, is the output FIS structure with the minimum checking error. This FIS structure 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 structure 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 structure 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**.

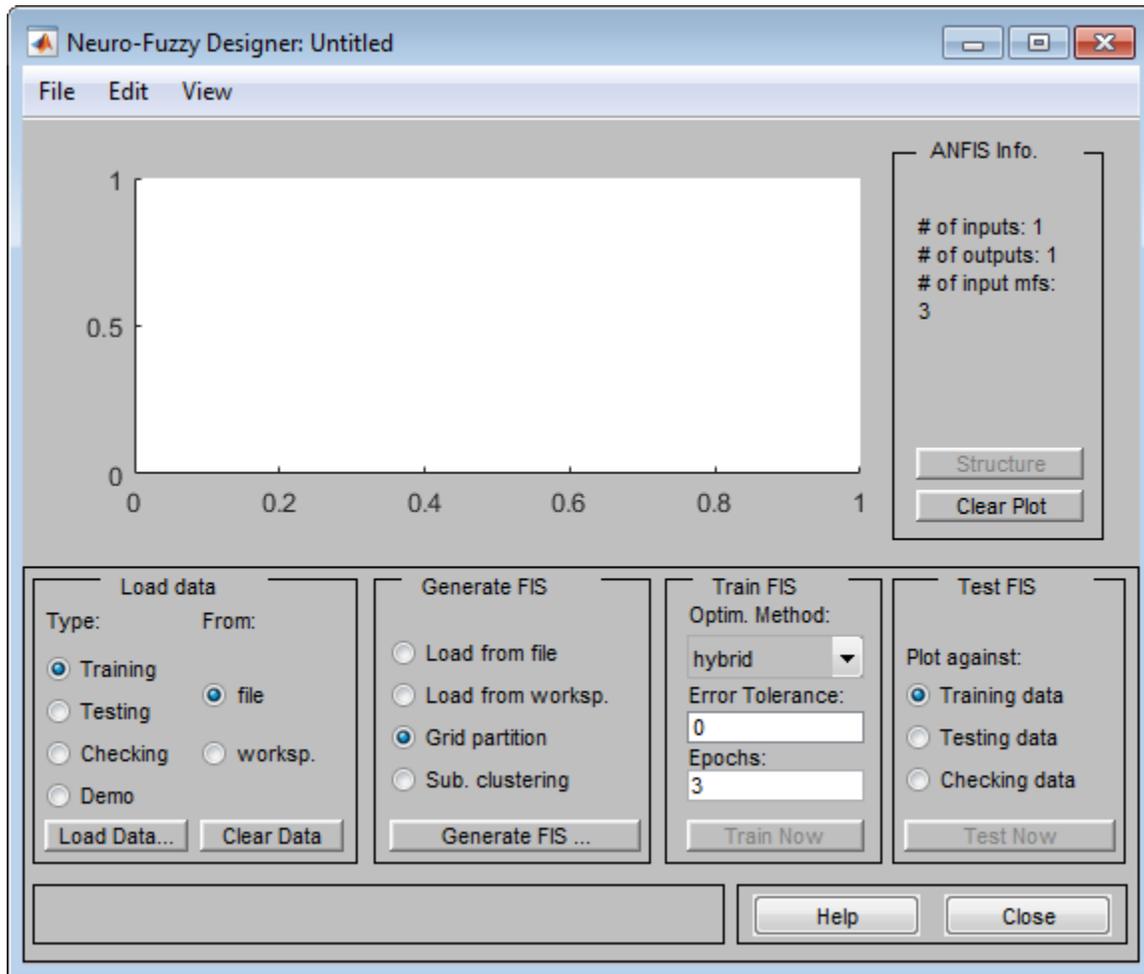
To start the app, type the following command at the MATLAB prompt:

```
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 3-15
- 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

In this section...

“Checking Data Helps Model Validation” on page 3-18

“Checking Data Does Not Validate Model” on page 3-29

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.

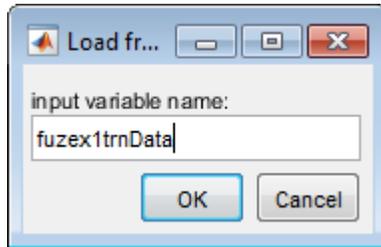
To load the data sets from the workspace into the **Neuro-Fuzzy Designer**:

- 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
```

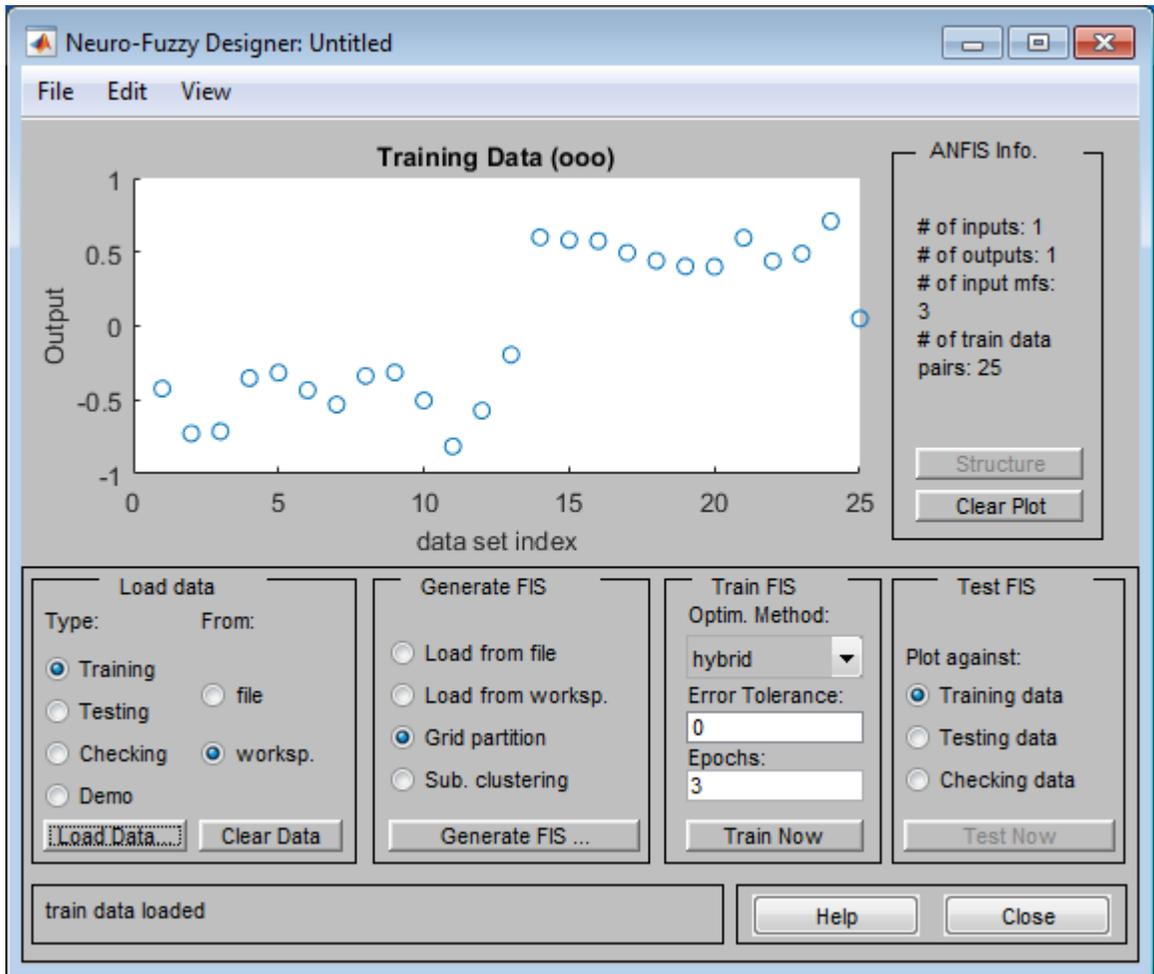
- 2 Open the **Neuro-Fuzzy Designer** by typing `neuroFuzzyDesigner` in the MATLAB command line.
- 3 To load the training data set from the workspace:
 - a In the **Load data** portion of the designer, select the following options:
 - **Type: Training**

- **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*.

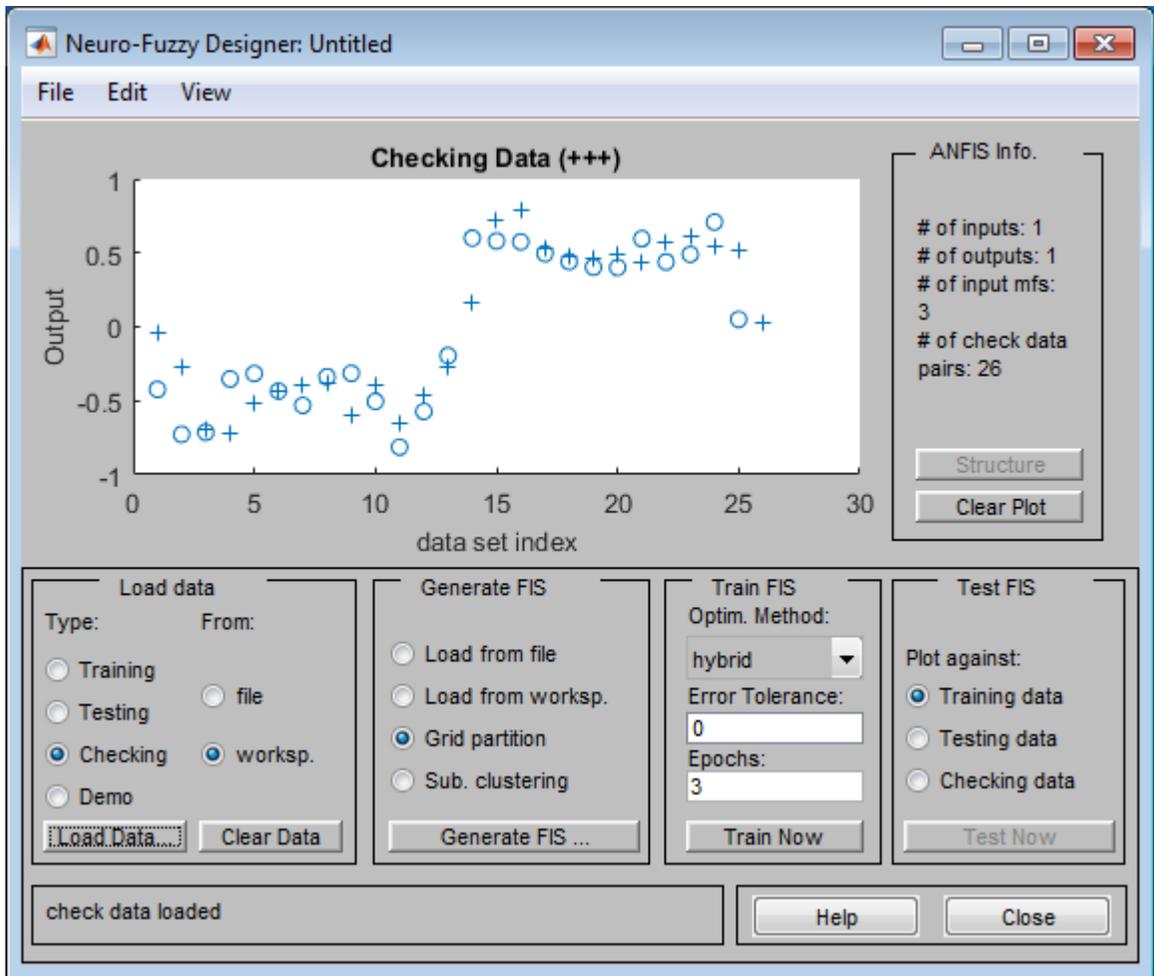


The horizontal axis is marked **data set index**. This index indicates the row from which that input data value was obtained (whether or not the input is a vector or a scalar).

- 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

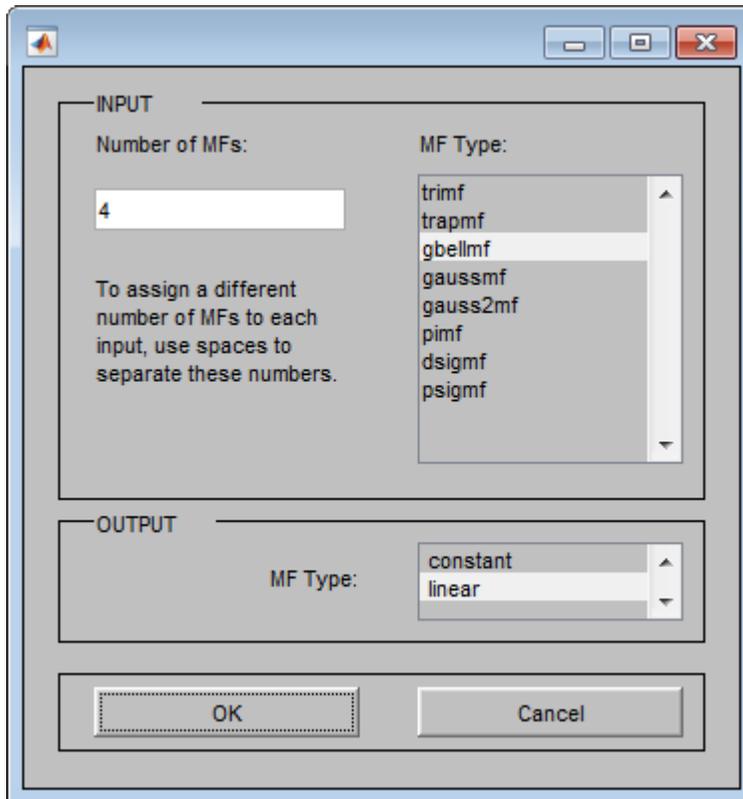
You can either initialize the FIS parameters to your own preference, or if you do not have any preference for how you want the initial membership functions to be parameterized, you can let `anfis` initialize the parameters for you, as described in the following sections:

- “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.

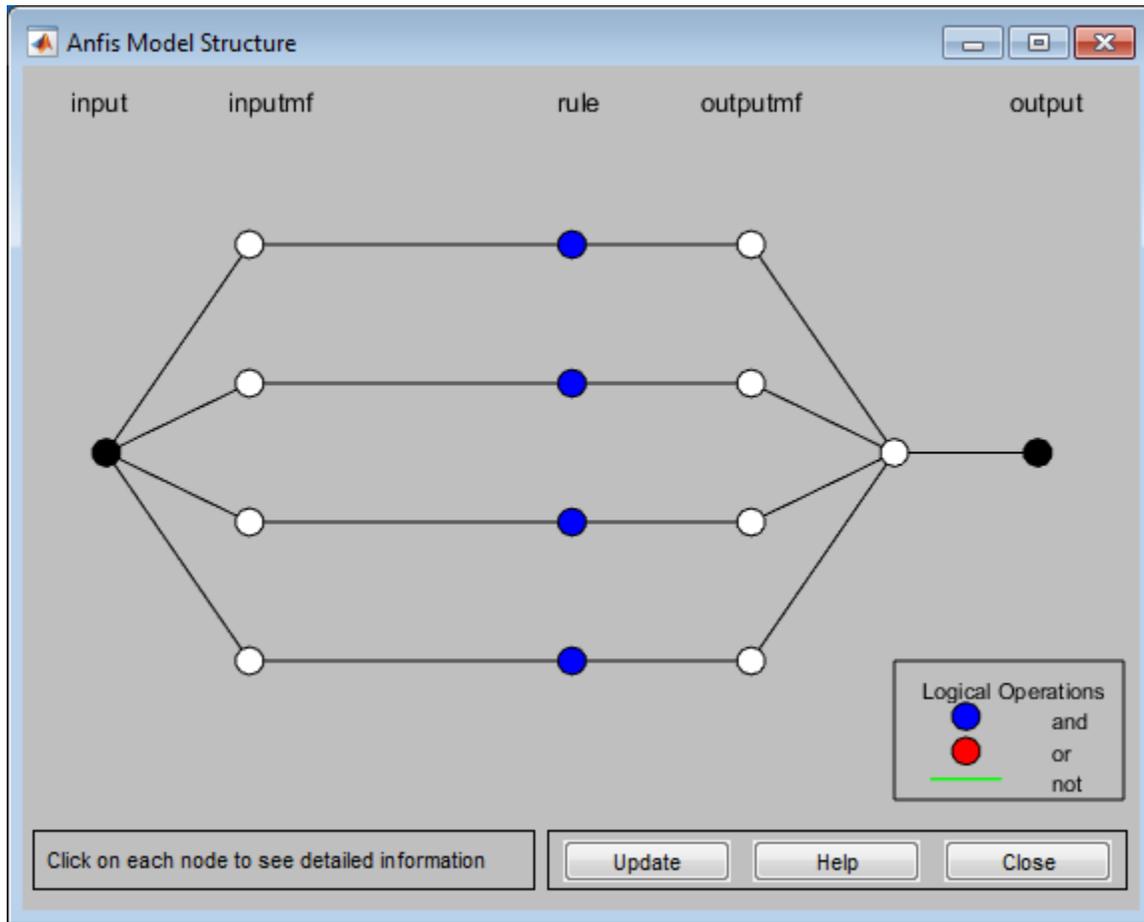
To define your own FIS structure and parameters:

- 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 2-34 and “The Membership Function Editor” on page 2-39.

- 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 2-47).
- 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 characterizes the rules and indicates 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

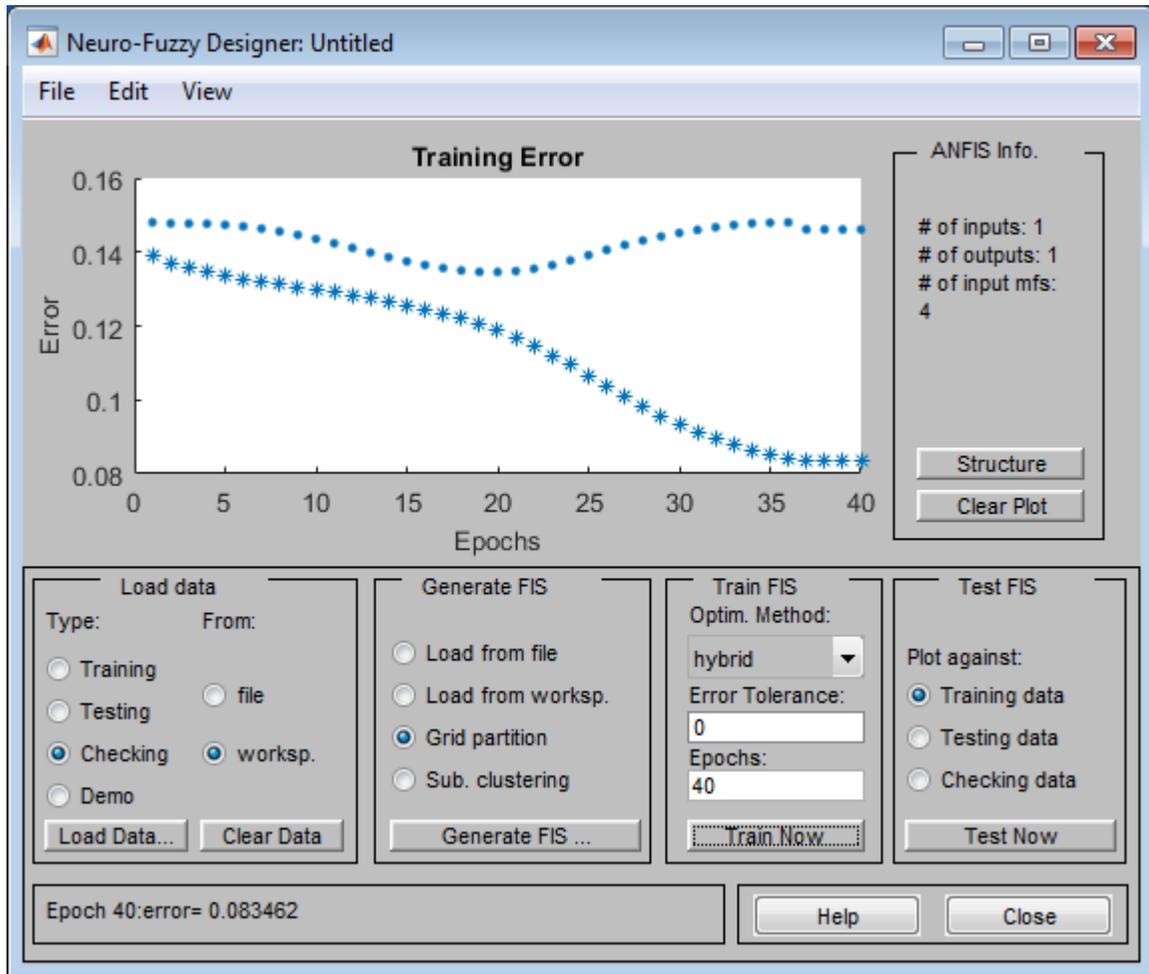
The two `anfis` parameter optimization method options available for FIS training are `hybrid` (the default, mixed least squares and backpropagation) and `backpropa` (backpropagation). **Error Tolerance** is used to create a training stopping criterion, which is related to the error size. The training 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**.

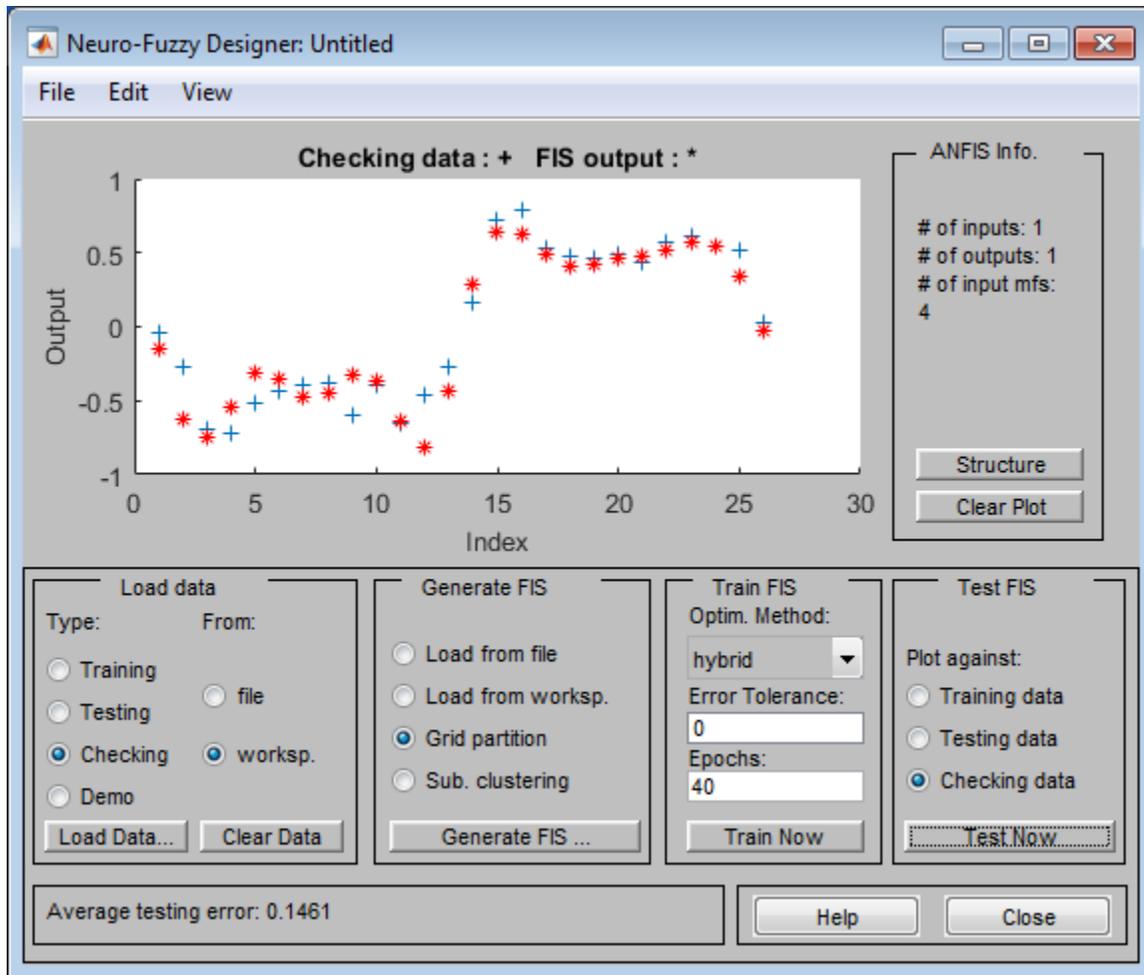
The following window appears on your screen.



The plot shows the checking error as $\blacklozenge \blacklozenge$ 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.

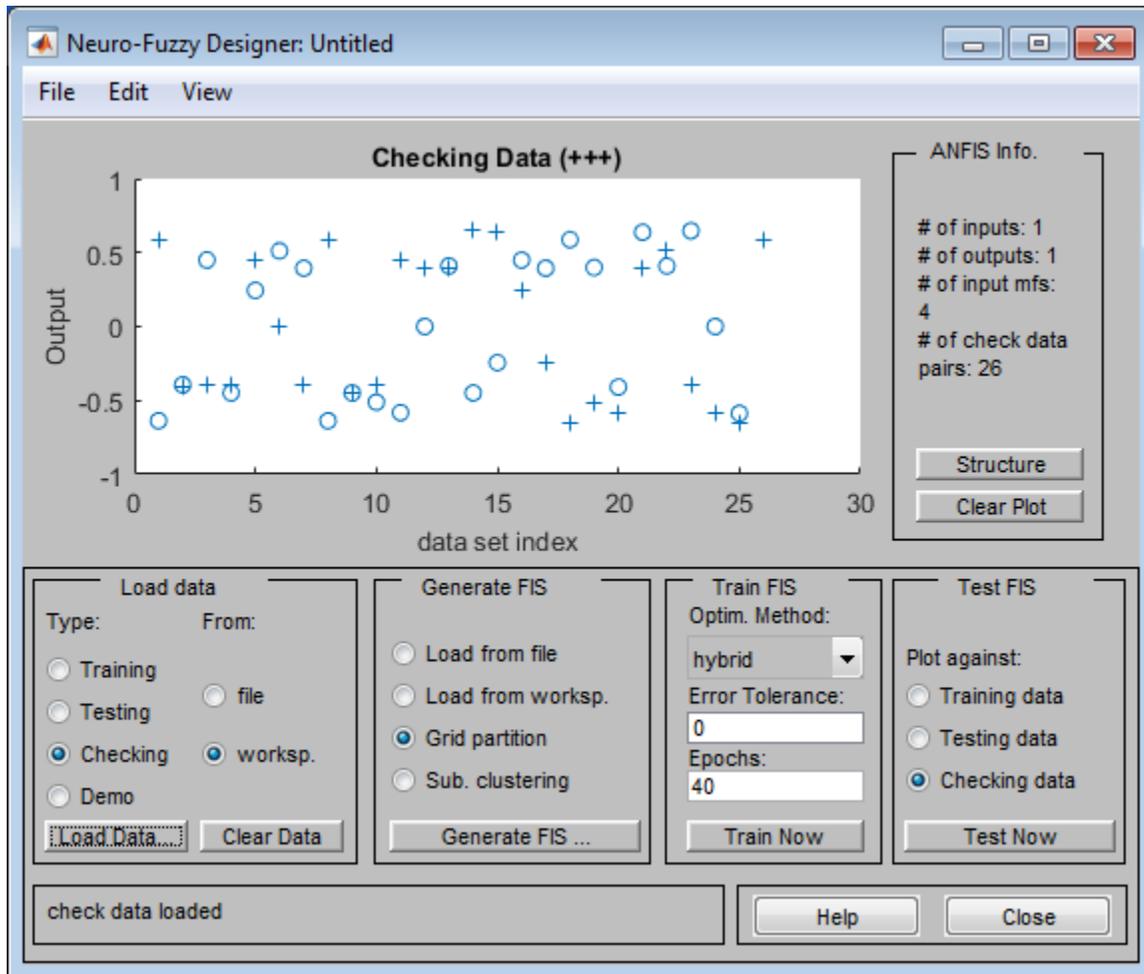
After clearing the data, you must regenerate your FIS. After the FIS is generated, you can use your first training experience to decide on the number of training epochs you want for the second round of training.

Checking Data Does Not Validate Model

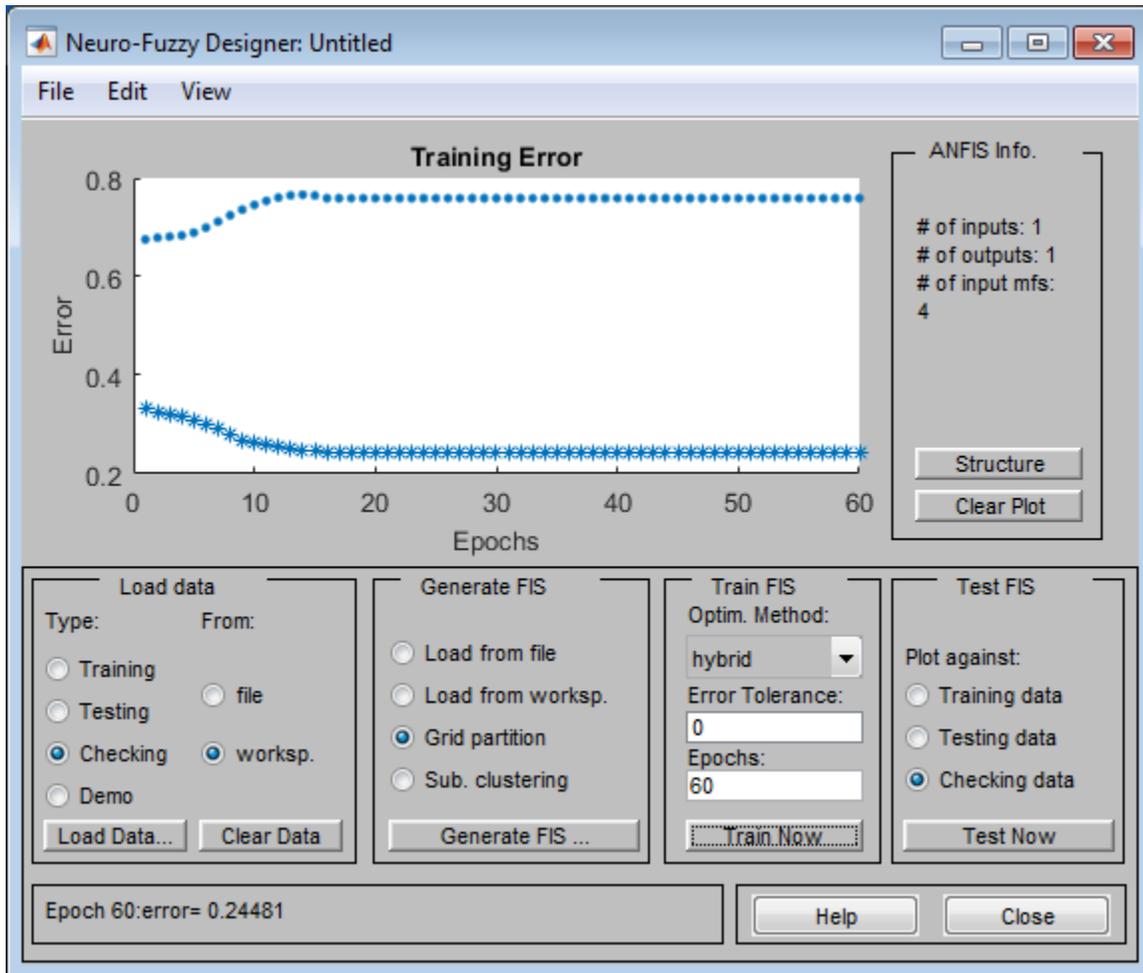
This example examines what happens when the training and checking data sets are sufficiently different. To see how the **Neuro-Fuzzy Designer** can be used to learn something about data sets and how they differ:

- 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.



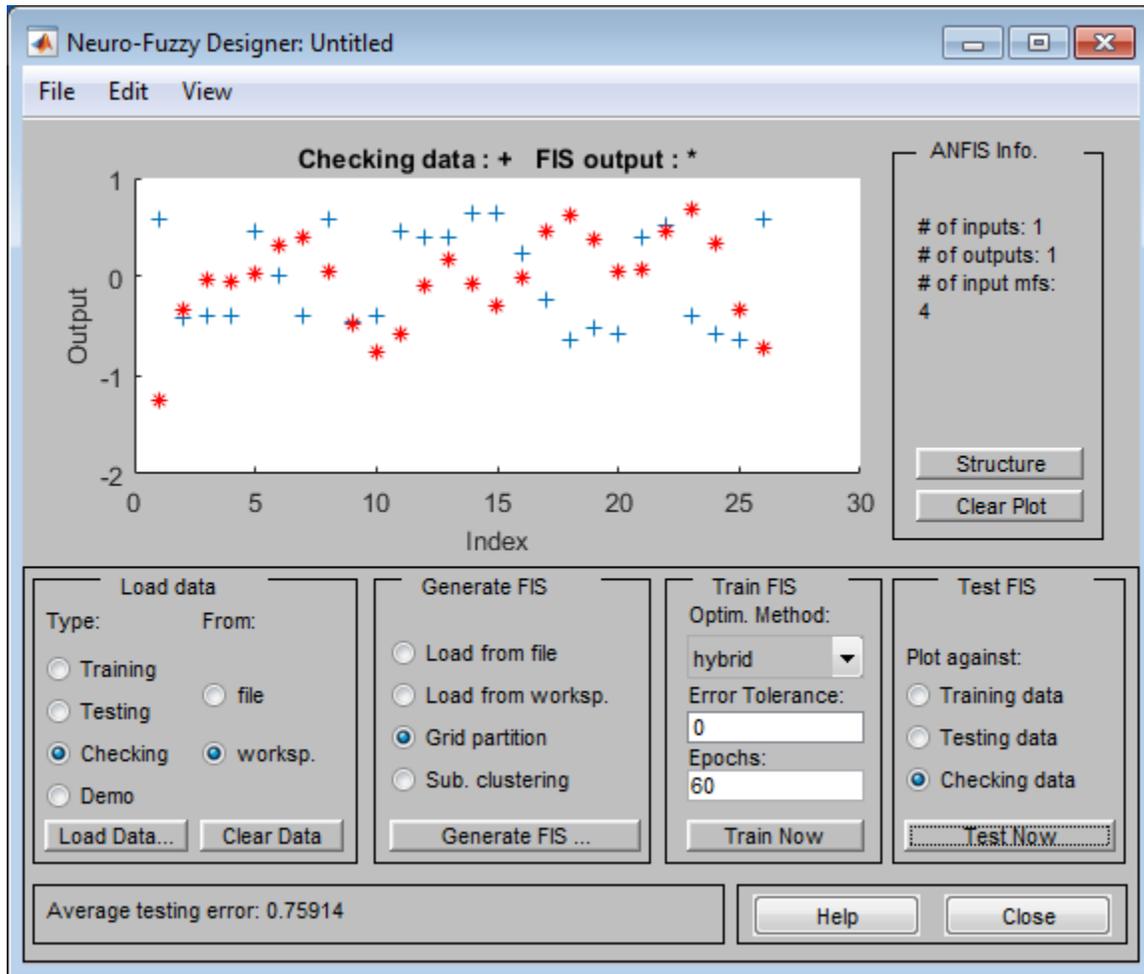
In this case, the checking error is quite large. It appears that the minimum checking error occurs within the first epoch. Using the checking data option with `anfis` automatically sets the FIS parameters to be those associated with the minimum checking error. Clearly this set of membership functions 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.

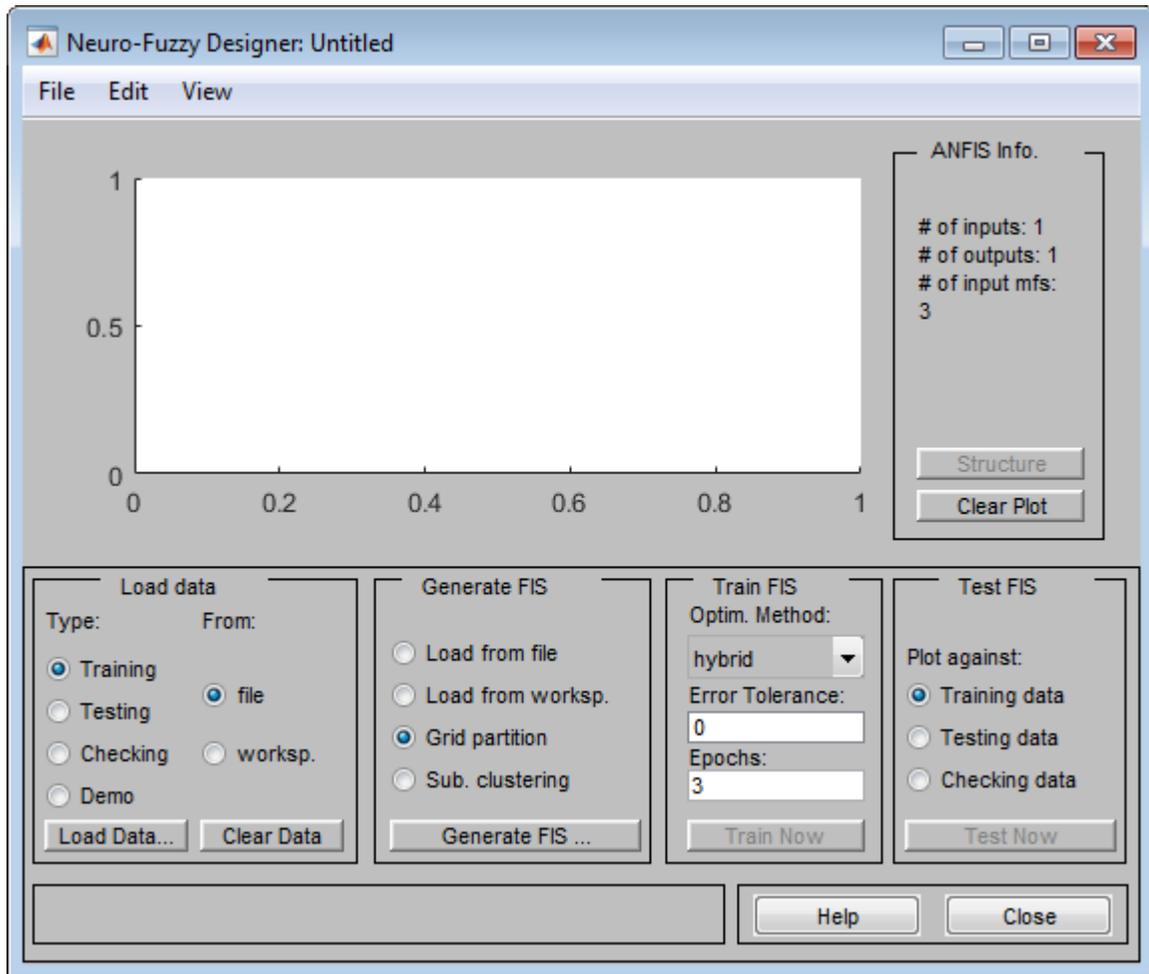
The following example shows how to save the training error generated during ANFIS training to the MATLAB workspace:

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

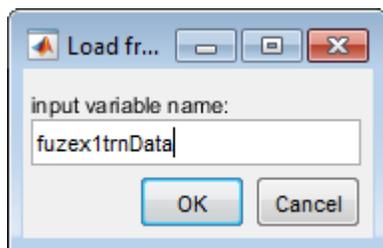
```
load fuzex1trnData.dat  
load fuzex1chkData.dat
```

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

```
neuroFuzzyDesigner
```

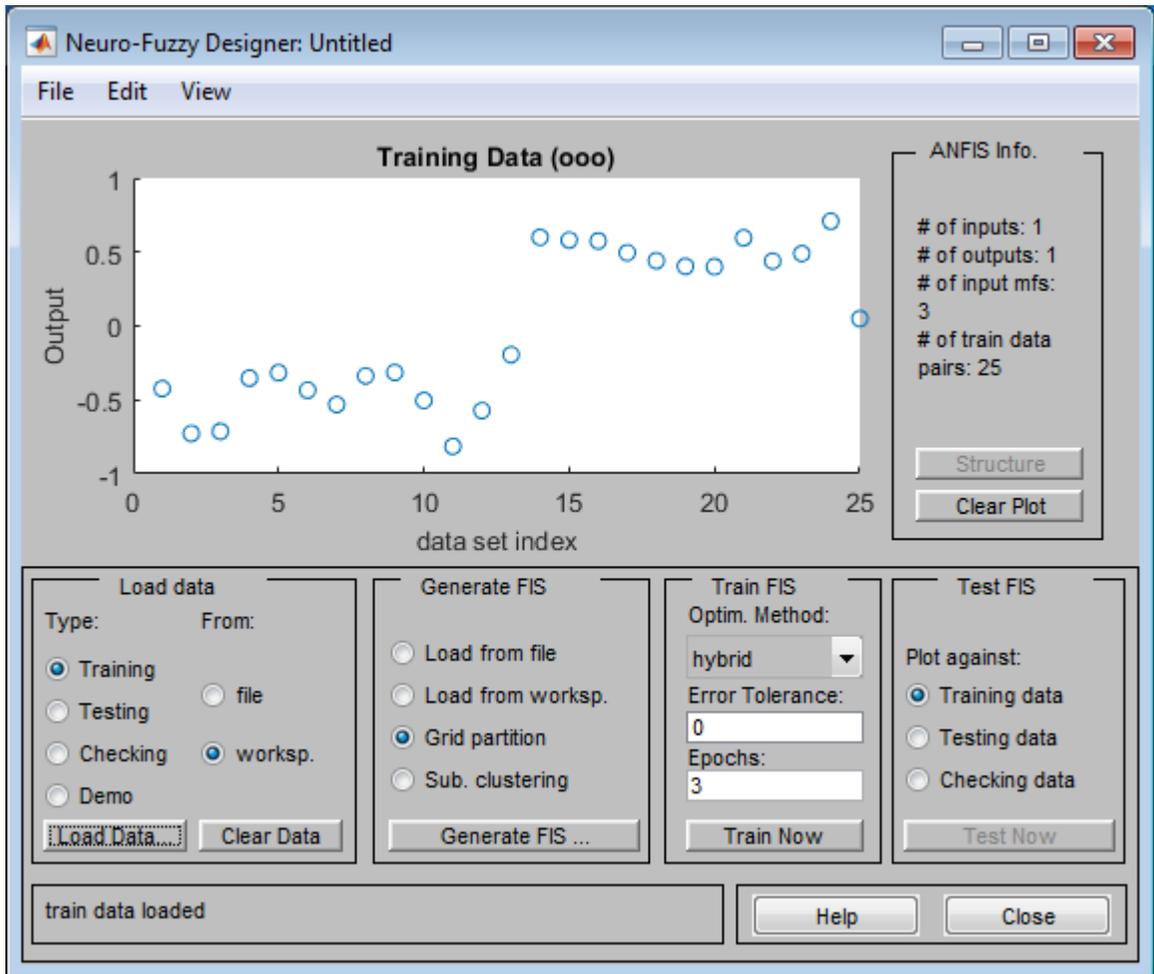


- 3 Load the training data from the MATLAB workspace into the **Neuro-Fuzzy Designer**:
 - a In the **Load data** panel of the **Neuro-Fuzzy Designer**, verify that **Training** is selected in the **Type** column.
 - b Select **worksp** in the **From** column.
 - c Click **Load Data** to open the Load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



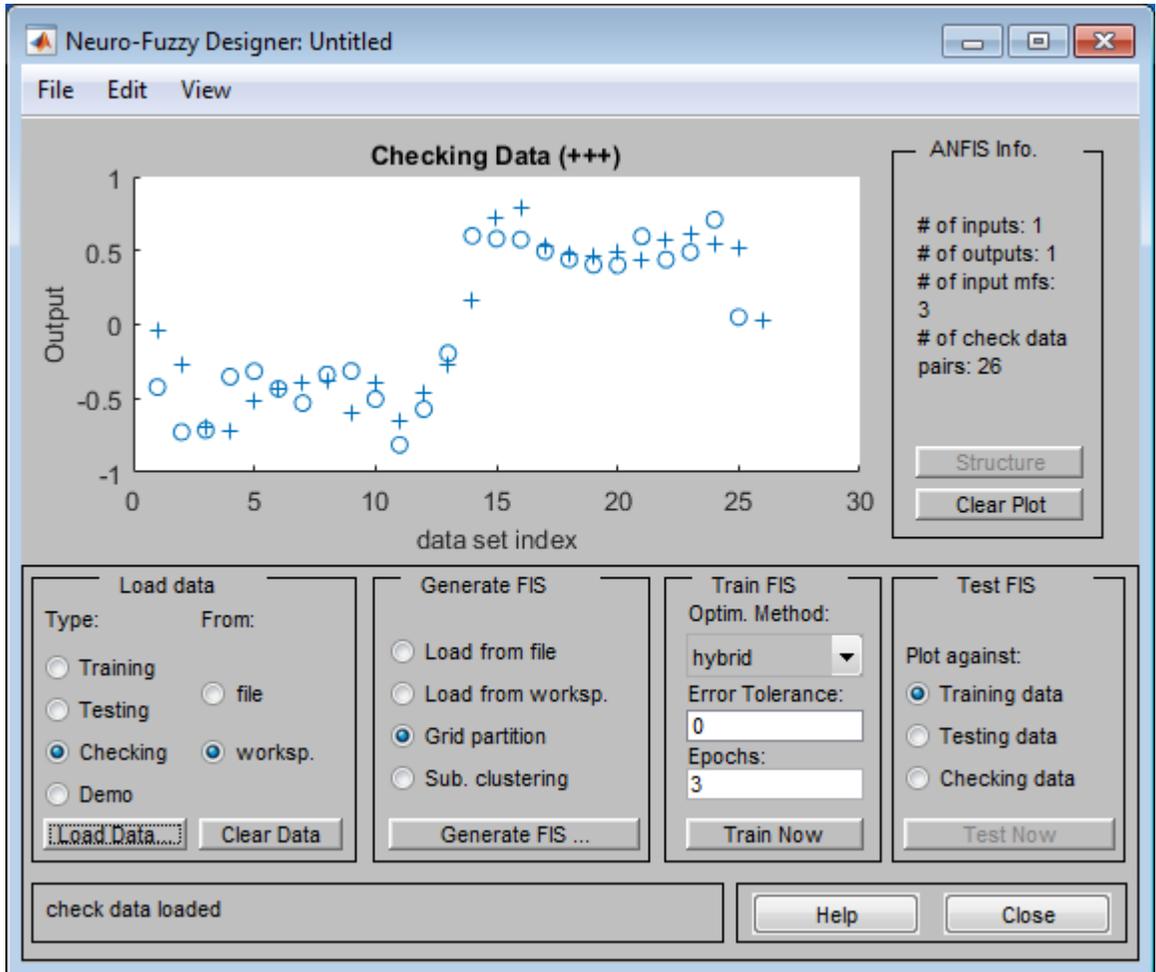
- d Type `fuzex1trnData`, and click **OK**.

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**.

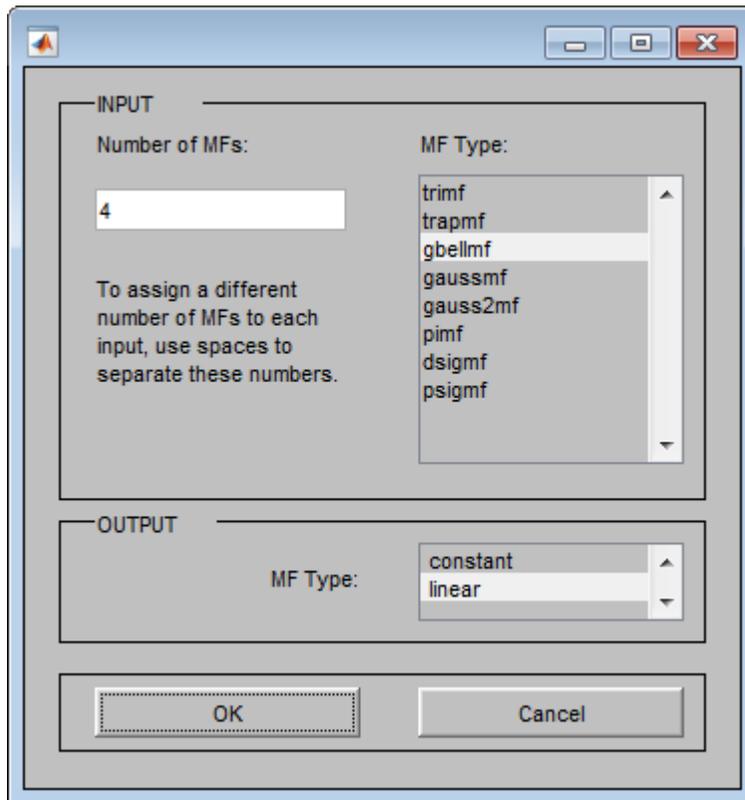
The **Neuro-Fuzzy Designer** displays the checking data as plus signs (+) superimposed on the training data.



- 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.

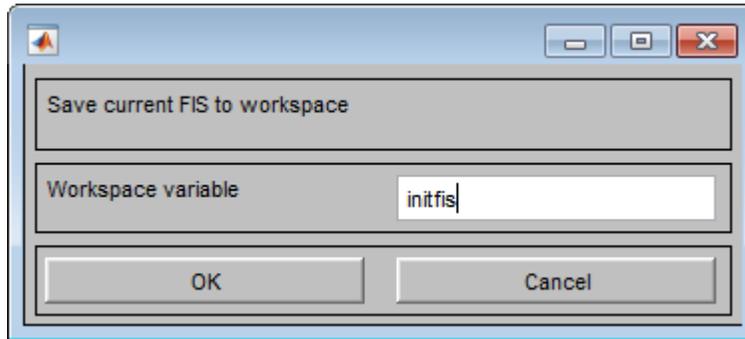
- c In the dialog box, specify the following:
 - Enter 4 in the **Number of MFs** field.
 - Select **gbellmf** as the **Membership Type** for the input.
 - Select **linear** as the **Membership Type** for the output.



- 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.

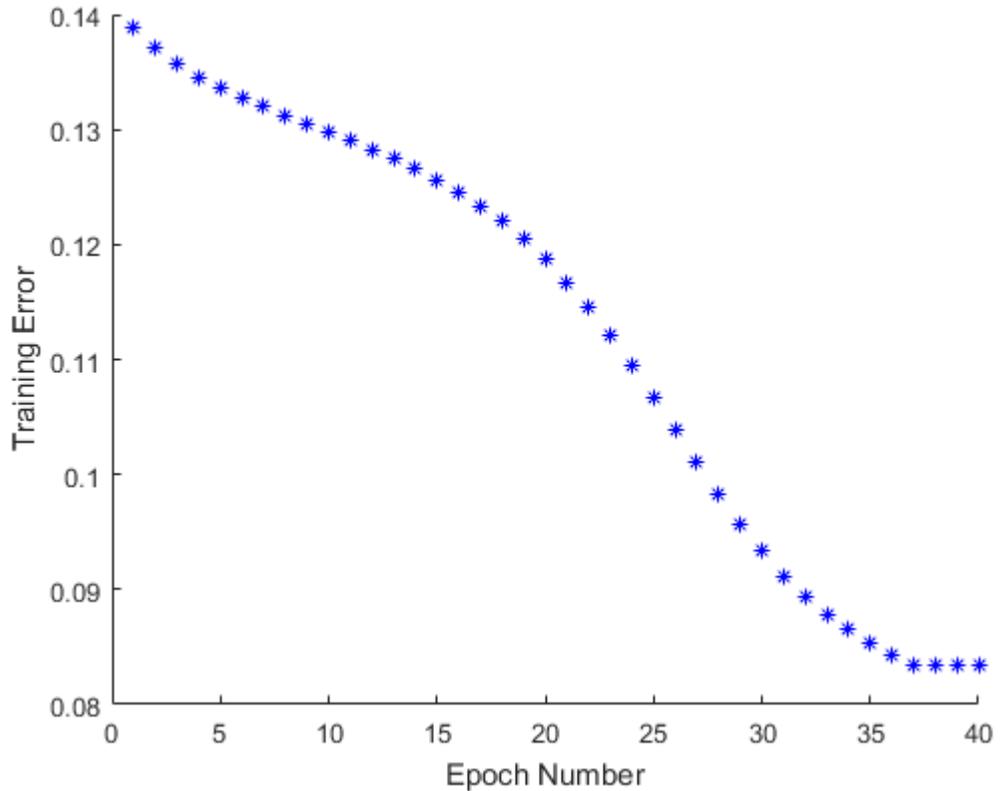
A variable named `initfis` now appears in the MATLAB workspace.

- 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 an FIS using the ANFIS Editor GUI is quite simple.

However, you need to be cautious about implementing the checking data validation feature of `anfis`. You must check that the checking data error does what it is supposed to. Otherwise, you need to retrain the FIS.

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

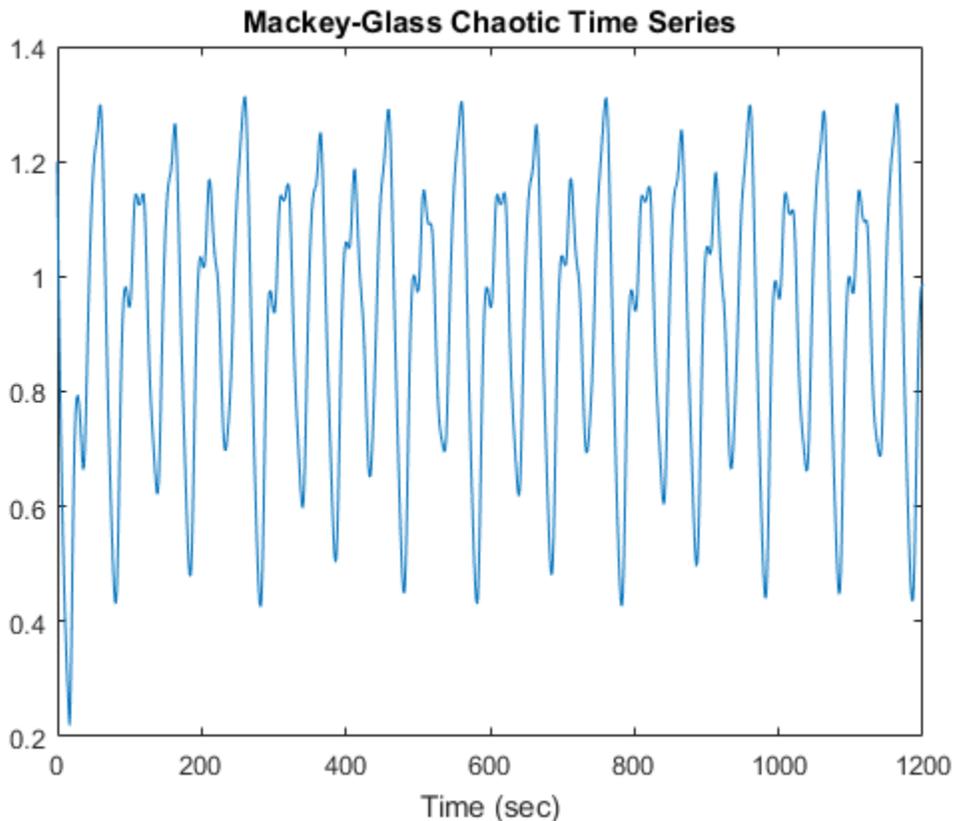
$$\dot{x}(t) = \frac{0.2x(t - \tau)}{1 + x^{10}(t - \tau)} = 0.1x(t)$$

This time series is chaotic 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 need to 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 (these become the training data set), 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 structures, `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,:);
```

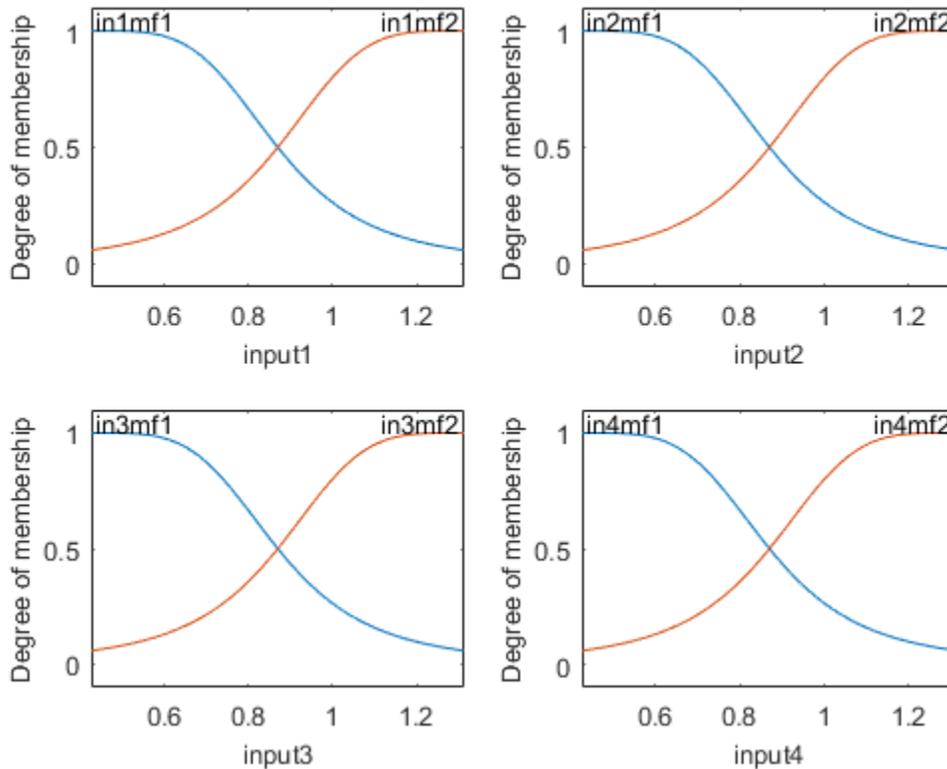
To start the training, specify an initial FIS structure using the `genfis` command. By default, `genfis` creates the FIS structure using grid partitioning.

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

Because 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 structure 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.

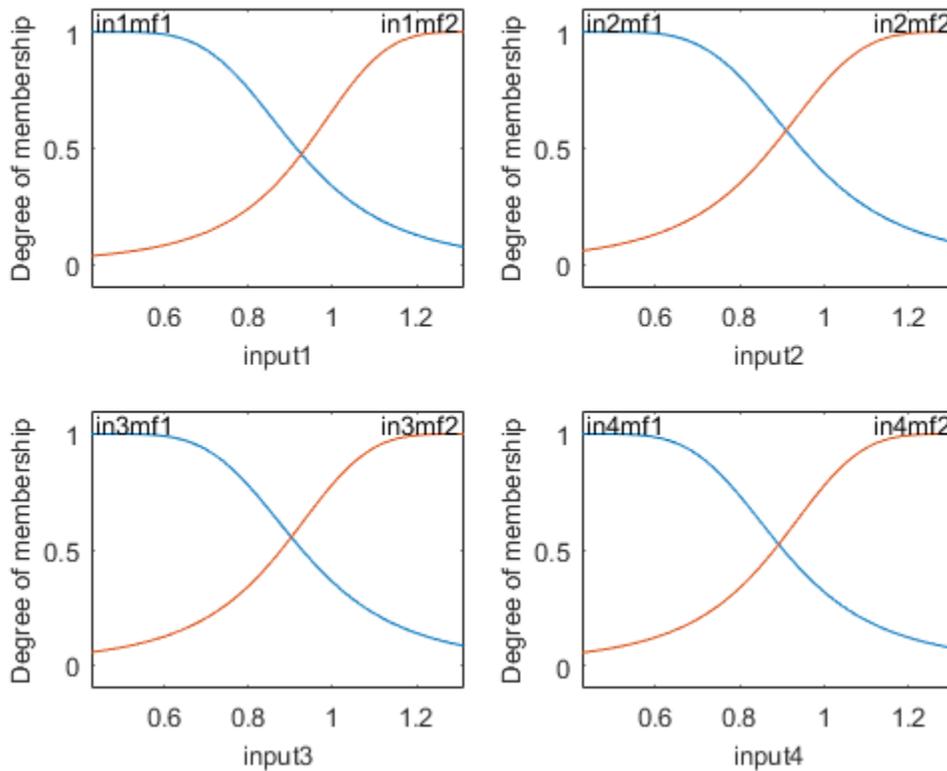
```
opt.DisplayANFISInformation = 0;
opt.DisplayErrorValues = 0;
opt.DisplayStepSize = 0;
opt.DisplayFinalResults = 0;
```

Train the FIS using the specified training data and options.

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

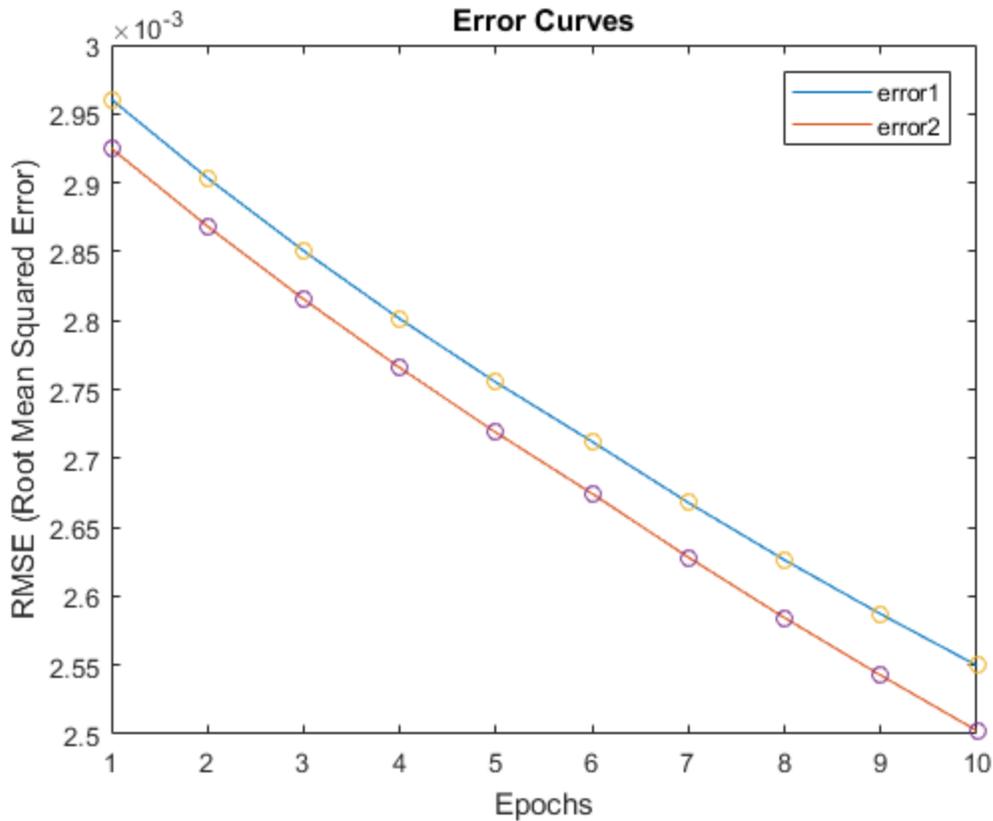
Because the `ValidationData` options is set, the final FIS you choose is the one associated with the minimum checking error. This result is stored in `fismat2`. Plots these new membership functions.

```
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')
```

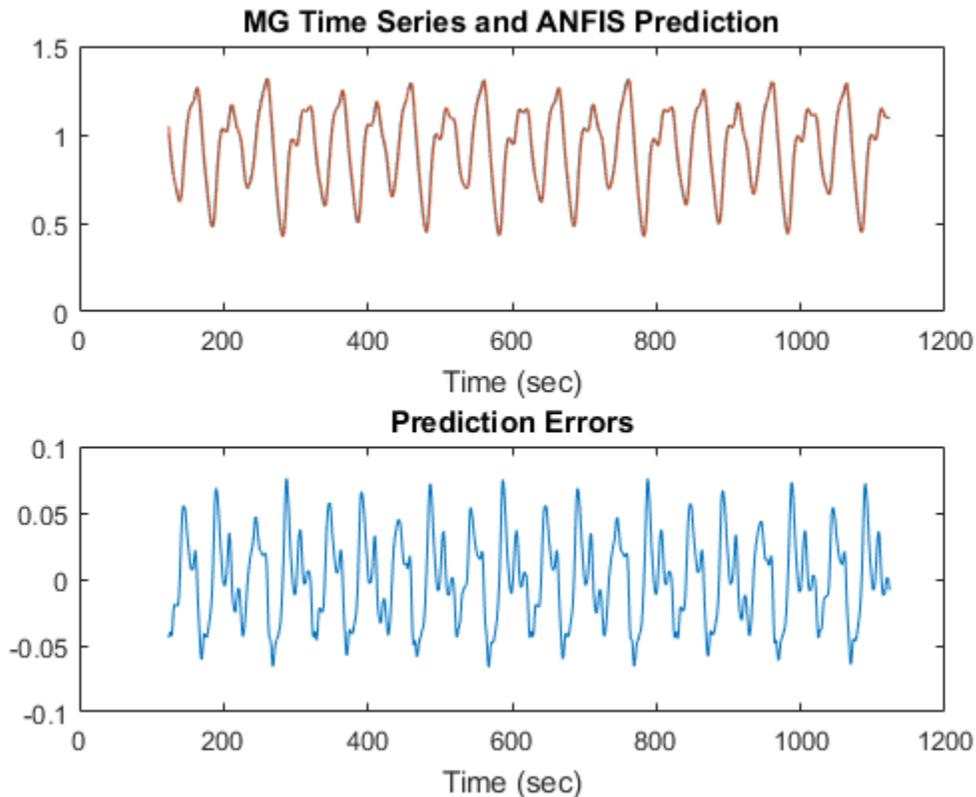


In addition to these error plots, you may want to plot the FIS output versus the training or checking data. To compare the original MG time series and the fuzzy prediction side by side, try:

```

figure(5)
anfis_output = evalfis([trnData(:,1:4); chkData(:,1:4)],fismat2);
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 very 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

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.

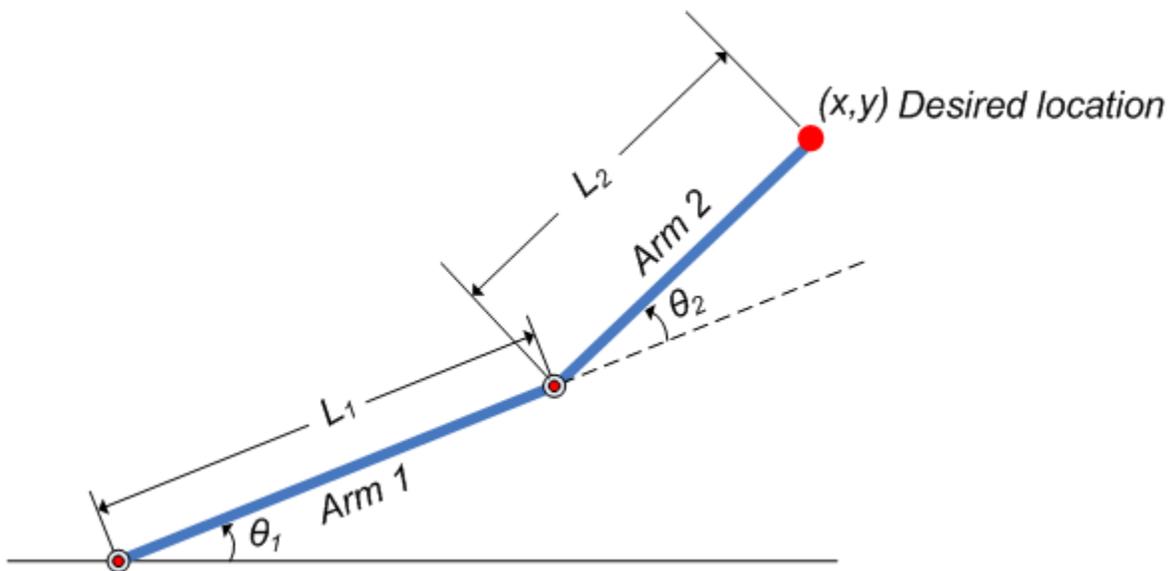


Figure 1: Illustration showing the two-joint robotic arm with the two angles, θ_1 and θ_2

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 $0 \leq \theta_2 \leq \pi$.

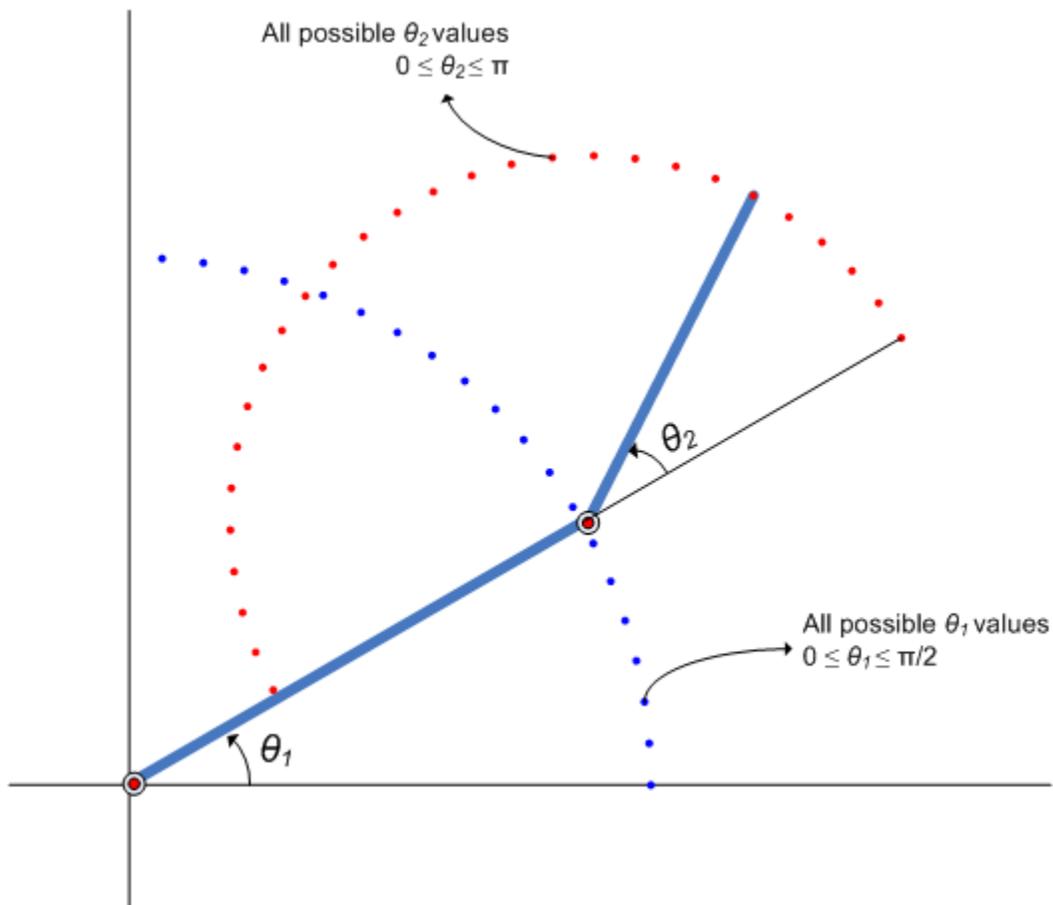


Figure 2: Illustration showing all possible `theta1` and `theta2` values.

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
```

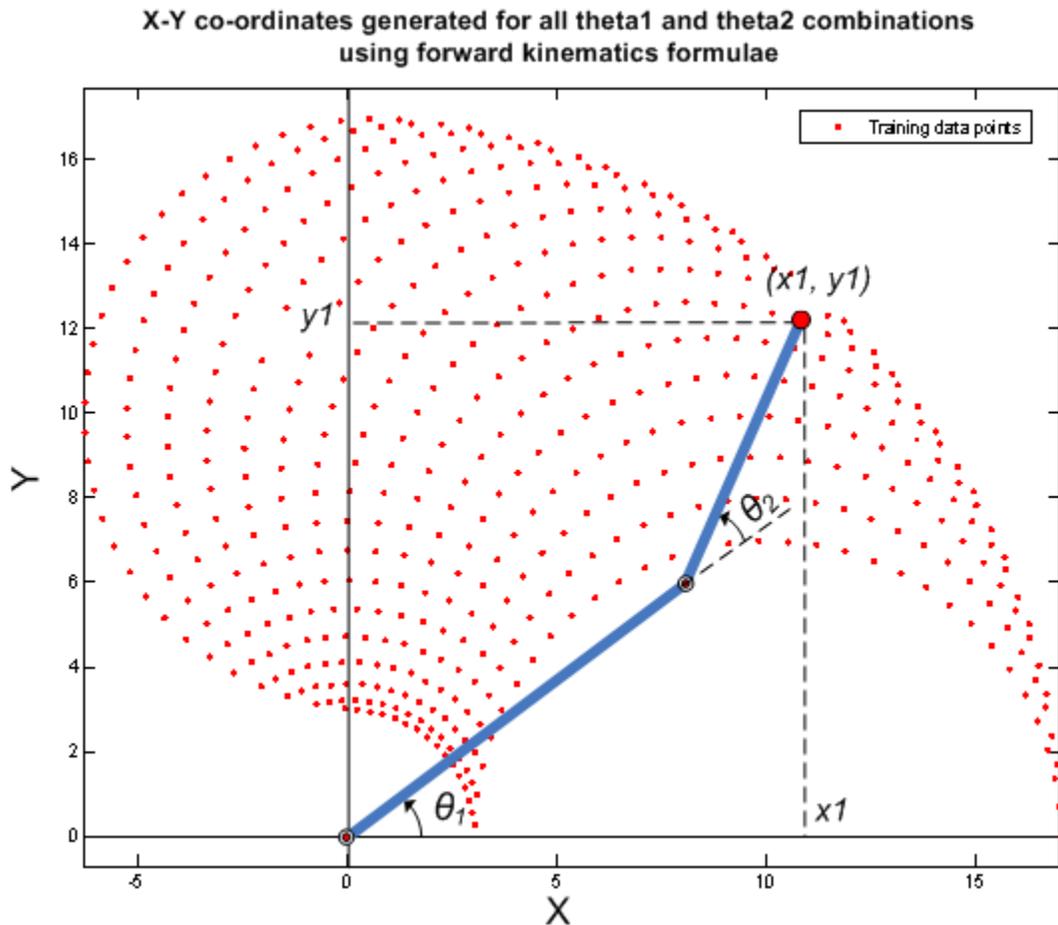


Figure 3: X-Y coordinates generated for all theta1 and theta2 combinations using forward kinematics formulae

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 structure 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.')
```

```
anfis1 = anfis(data1,opt);
```

```
--> Training first ANFIS network.
```

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.')
```

```
opt.InitialFIS = 6;  
anfis2 = anfis(data2,opt);
```

```
--> Training second ANFIS network.
```

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 - 11^2 - 12^2)/(2*11*12);
s2 = sqrt(1 - c2.^2);
THETA2D = atan2(s2,c2); % theta2 is deduced

k1 = 11 + 12.*c2;
k2 = 12*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.

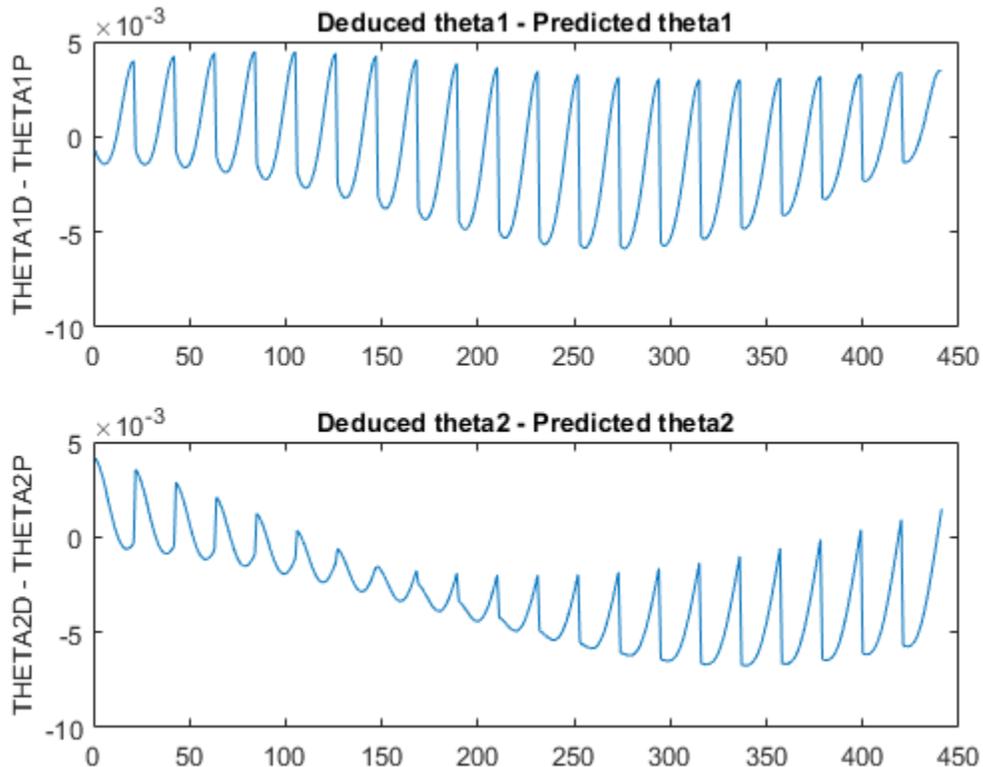
`theta1` and `theta2` values predicted by the trained `anfis` networks are obtained by using the command `evalfis` which evaluates a FIS for the given inputs.

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(XY,anfis1); % theta1 predicted by anfis1  
THETA2P = evalfis(XY,anfis2); % 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 `invkine` command launches a GUI that shows how the two trained ANFIS networks perform when asked to trace an ellipse.

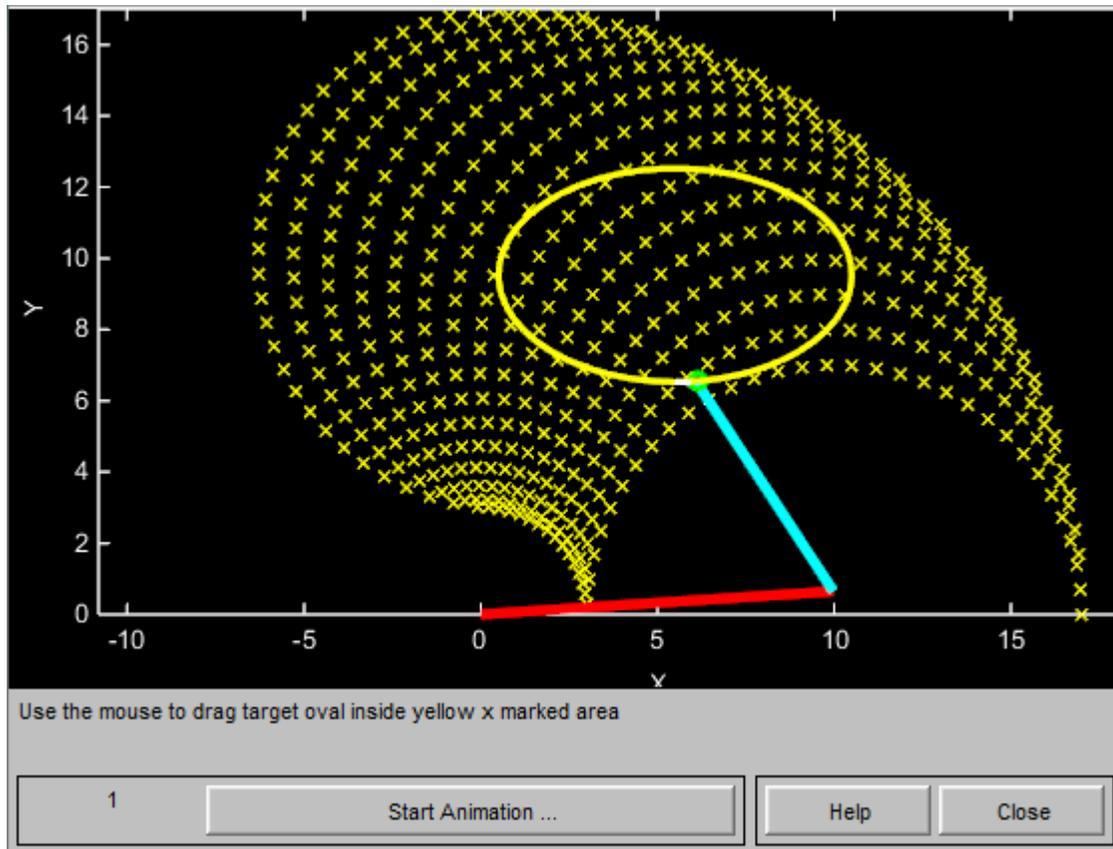


Figure 4: GUI for Inverse Kinematics Modeling.

The two ANFIS networks used in the example have been pre-trained 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

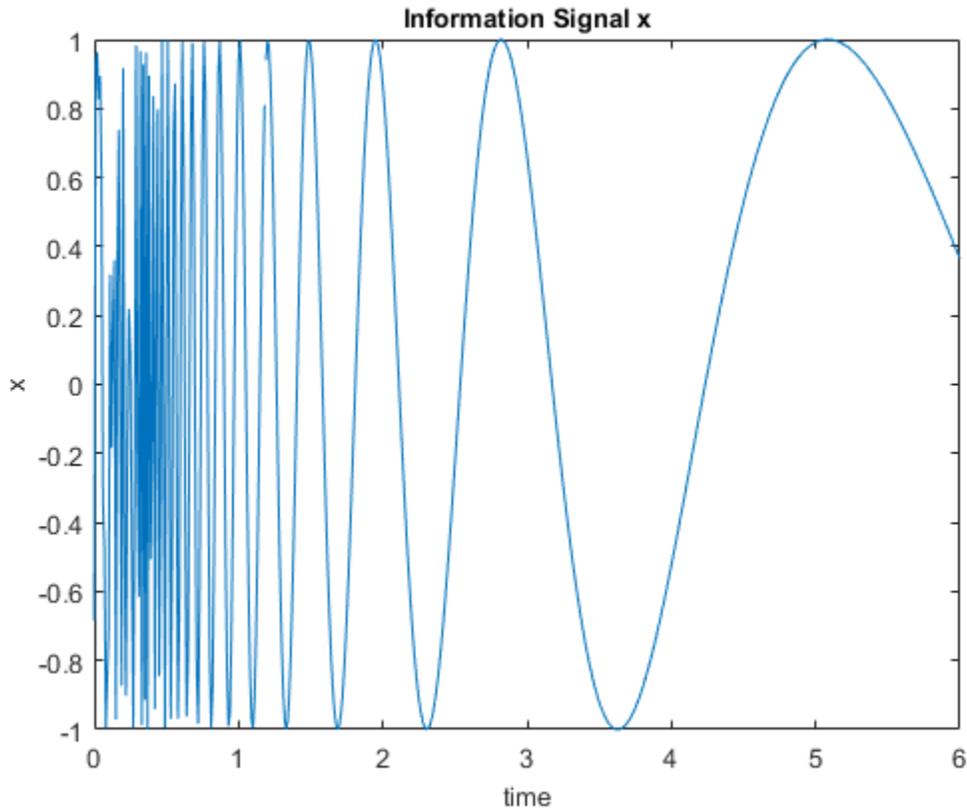
Adaptive Noise Cancellation Using ANFIS

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.

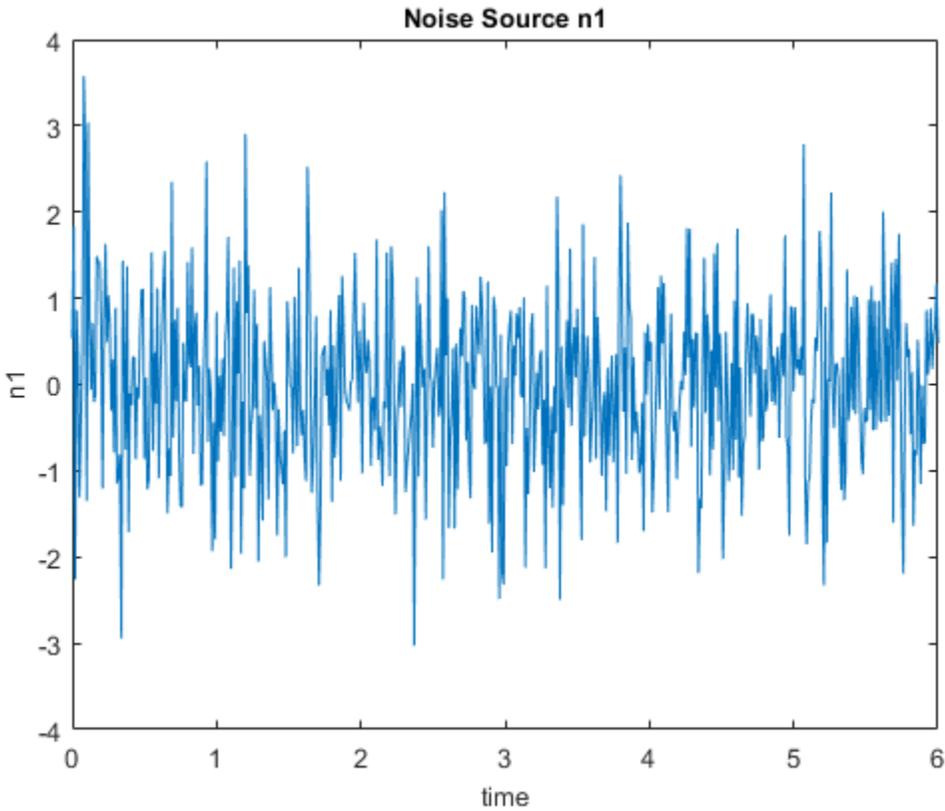
```
time = (0:0.01:6)';  
x = sin(40./(time+0.01));  
plot(time,x)  
title('Information Signal x','fontsize',10)  
xlabel('time','fontsize',10)  
ylabel('x','fontsize',10)
```



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 n1','fontsize',10)  
xlabel('time','fontsize',10)  
ylabel('n1','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.

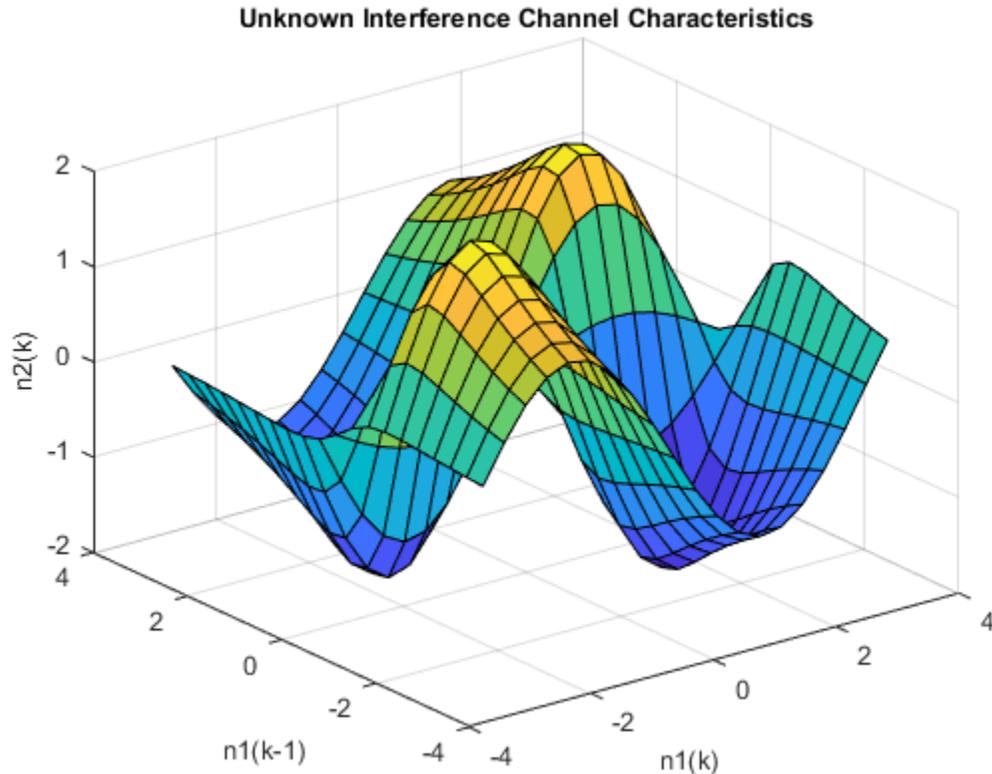
```
domain = linspace(min(n1),max(n1),20);
[xx,yy] = meshgrid(domain,domain);
zz = 4*sin(xx).*yy./(1+yy.^2);

surf(xx,yy,zz);
```

```

xlabel('n1(k)', 'fontsize', 10);
ylabel('n1(k-1)', 'fontsize', 10);
zlabel('n2(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.

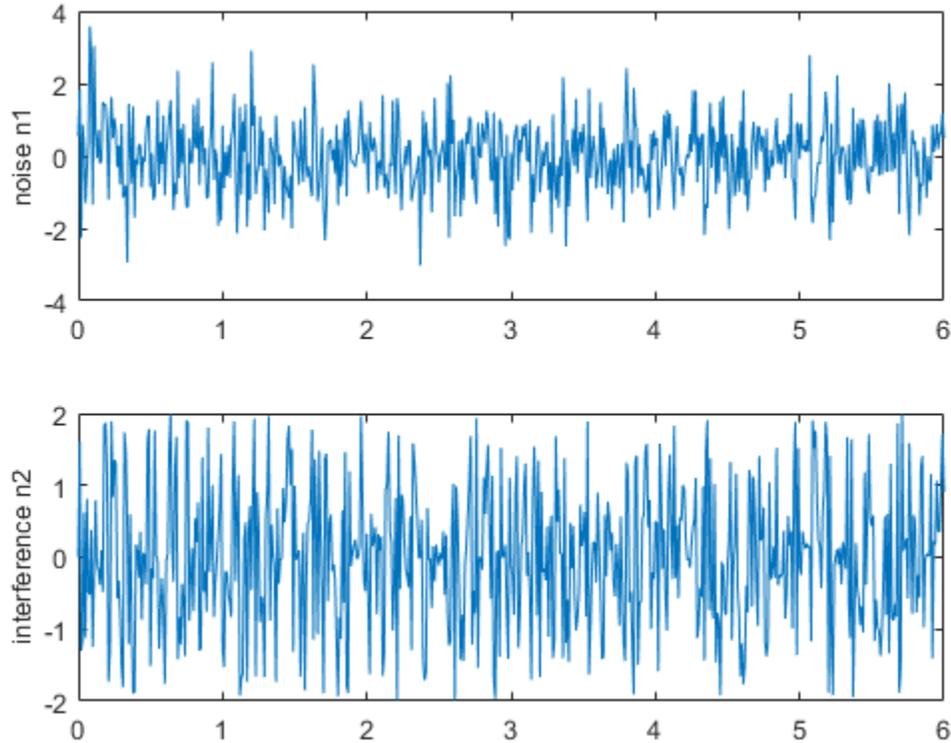
```

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 n1', 'fontsize', 10);

```

```
subplot(2,1,2)  
plot(time,n2);  
ylabel('interference n2','fontsize',10);
```

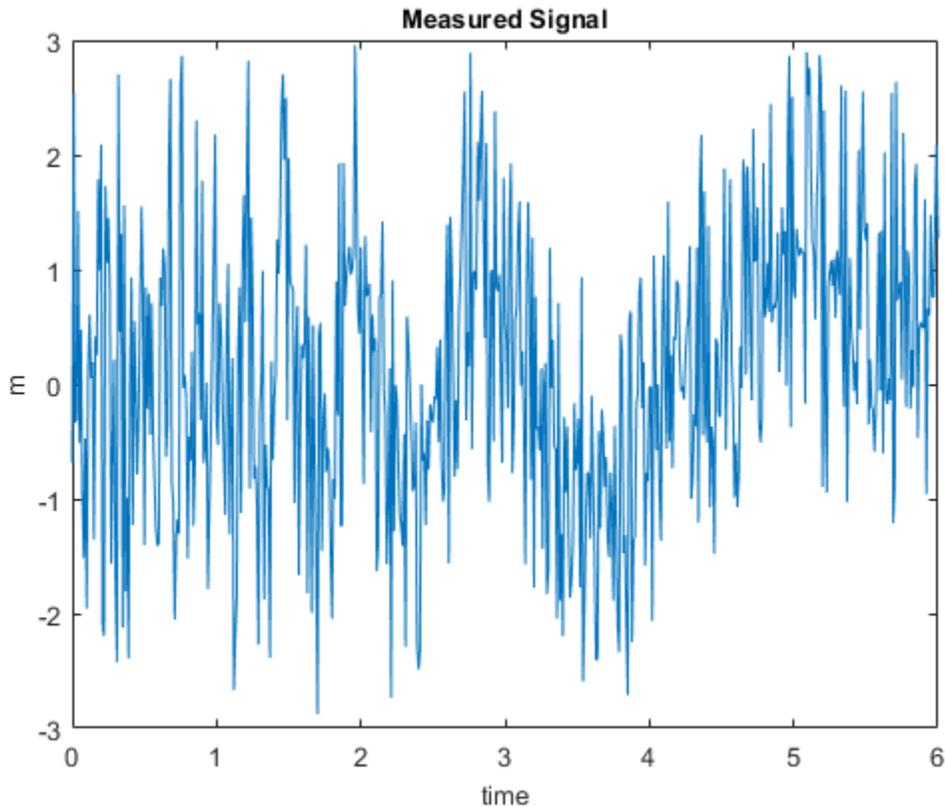


$n2$ is related to $n1$ via the highly nonlinear process shown previously; from the plots, it is hard to see if these two signals are correlated in any way.

The measured signal, m , is the sum of the original information signal, x , and the interference, $n2$. However, we do not know $n2$. The only signals available to us are the noise signal, $n1$, 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 $n1$ and $n2$. While $n2$ is not directly available, you can assume that m is a "contaminated" version of $n2$ 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 structure for training.

Define the training data. The first two columns of `data` are the inputs to the ANFIS structure, `n1` and a delayed version of `n1`. 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 structure. 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 `n1` and `n2`.

Evaluate Model

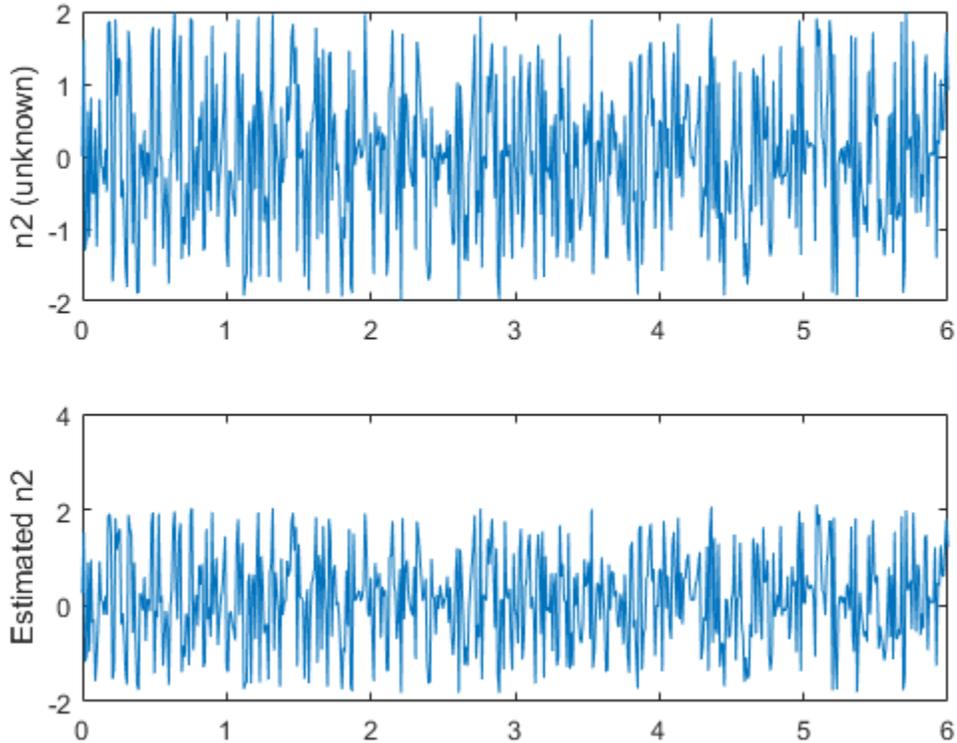
Calculate the estimated interference signal, `estimated_n2`, by evaluating the tuned FIS using the original training data.

```
estimated_n2 = evalfis(data(:,1:2),outFIS);
```

Plot the and actual `n2` signal and the estimated version from the ANFIS output.

```
subplot(2,1,1)
plot(time, n2)
ylabel('n2 (unknown)');

subplot(2,1,2)
plot(time, estimated_n2)
ylabel('Estimated n2');
```

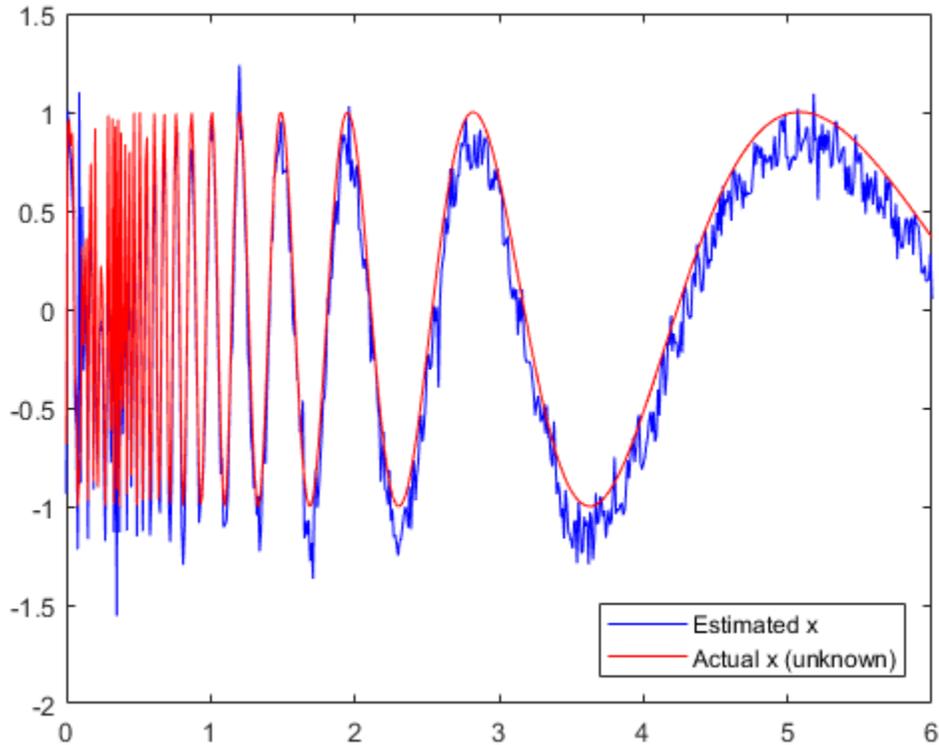


The estimated information signal is equal to the difference between the measured signal, m , and the estimated interference (ANFIS output).

```
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

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
- “Model Suburban Commuting Using Subtractive Clustering” on page 4-12
- “Data Clustering Using the Clustering Tool” on page 4-24

Fuzzy Clustering

In this section...
“What Is Data Clustering?” on page 4-2
“Fuzzy C-Means Clustering” on page 4-2
“Subtractive Clustering” on page 4-3
“References” on page 4-3

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] Bezdec, 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

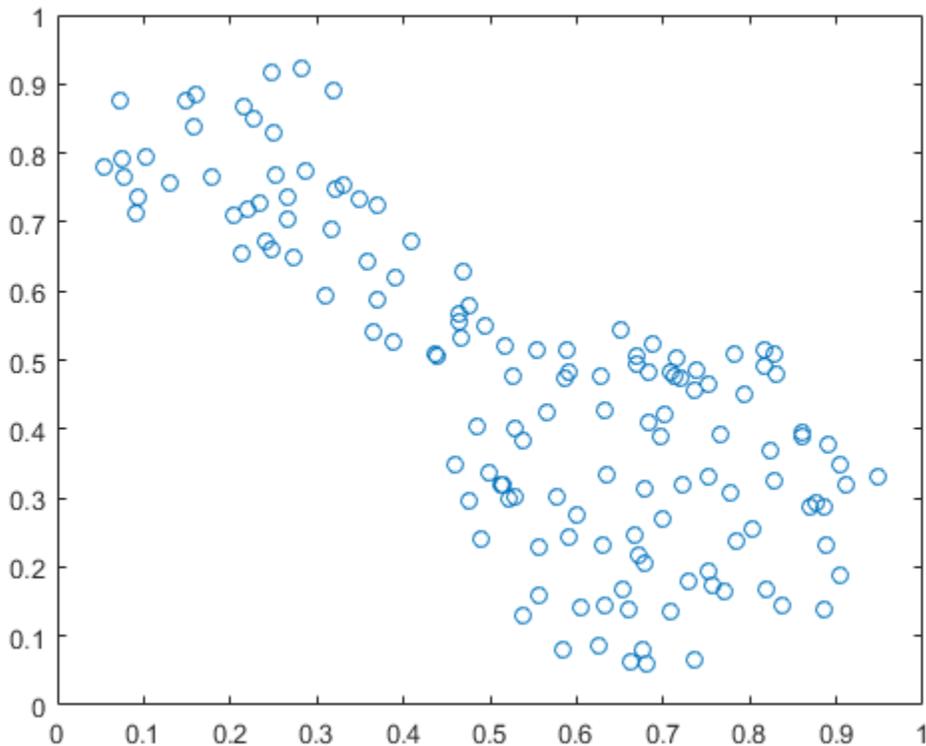
- “Cluster Quasi-Random Data Using Fuzzy C-Means Clustering” on page 4-4
- “Model Suburban Commuting Using Subtractive Clustering” on page 4-12
- “Data Clustering Using the Clustering Tool” on page 4-24

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
```

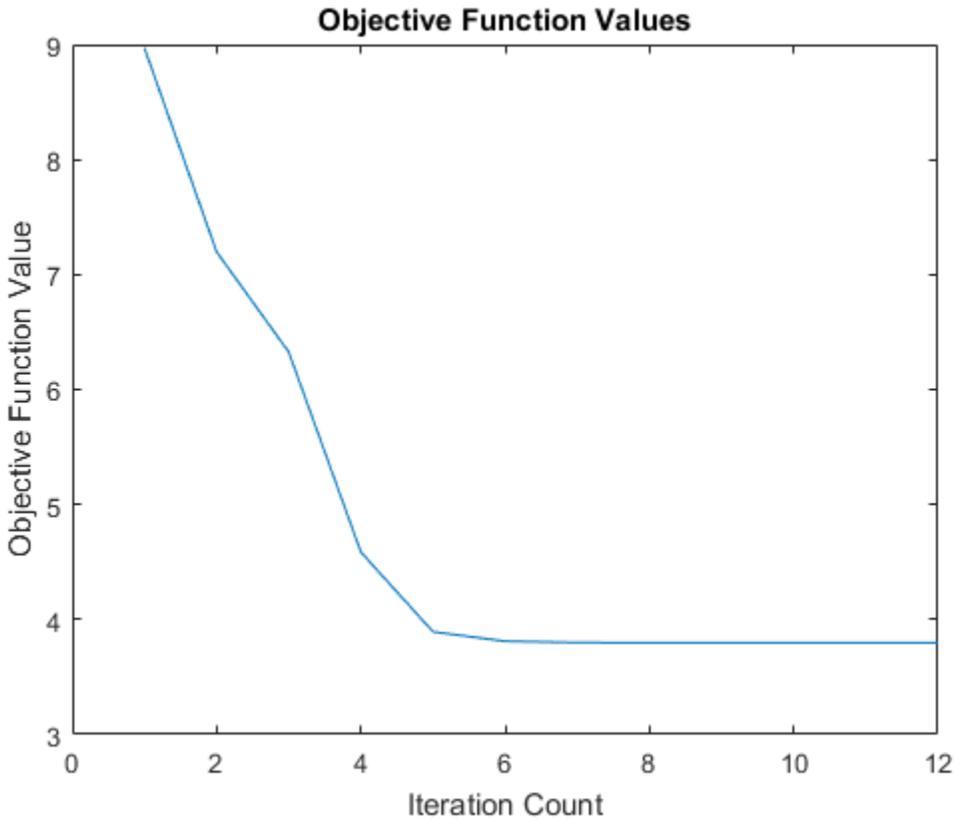
`center` contains the coordinates of the two cluster centers, `U` contains the membership grades for each of the data points, and `objFcn` contains a history of the objective function across the iterations.

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

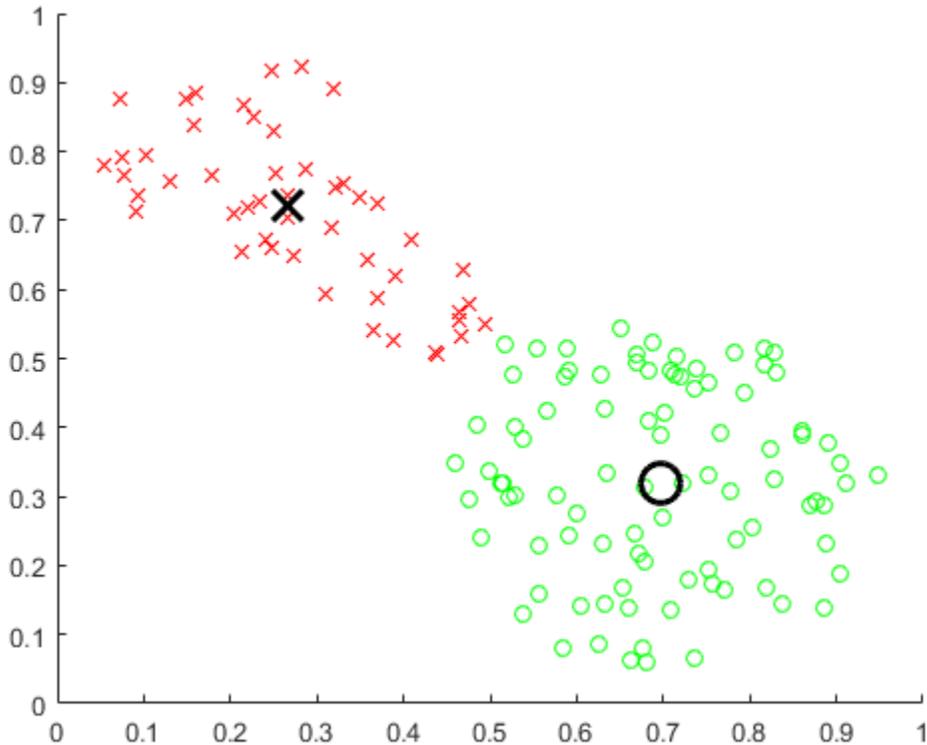
To view the progress of the clustering, plot the objective function.

```
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.

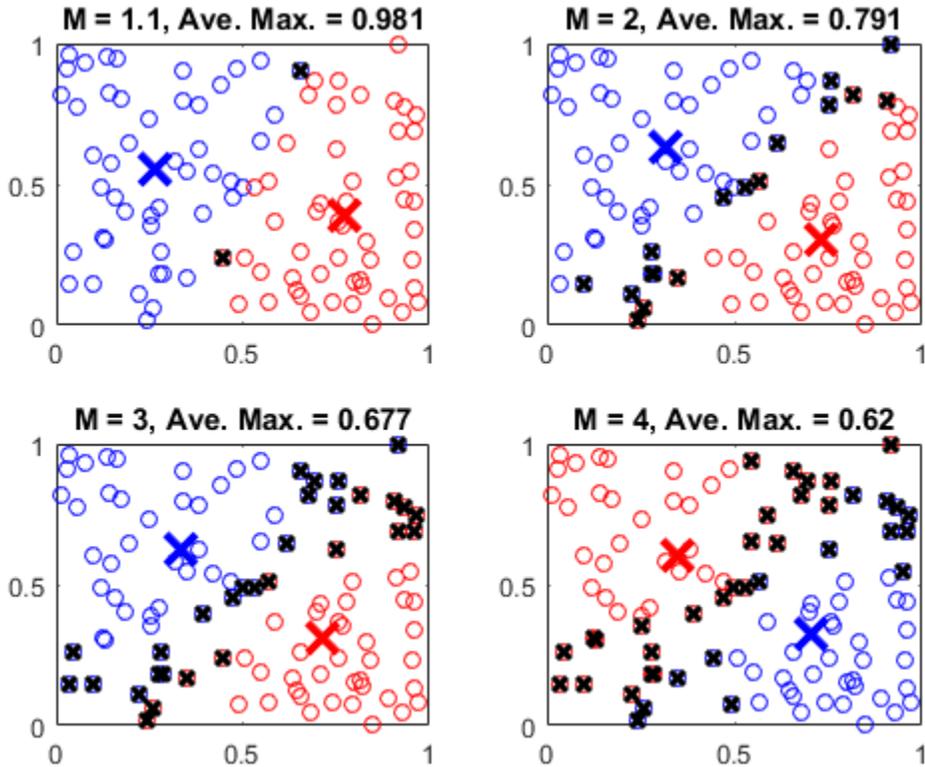
For each overlap exponent:

- 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.
- Calculate the average maximum membership value across all data points to quantify the degree of fuzzy overlap. 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 \times 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 indicates 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 completely crisp clusters, with smaller values indicating more overlap.

See Also

`fcm`

More About

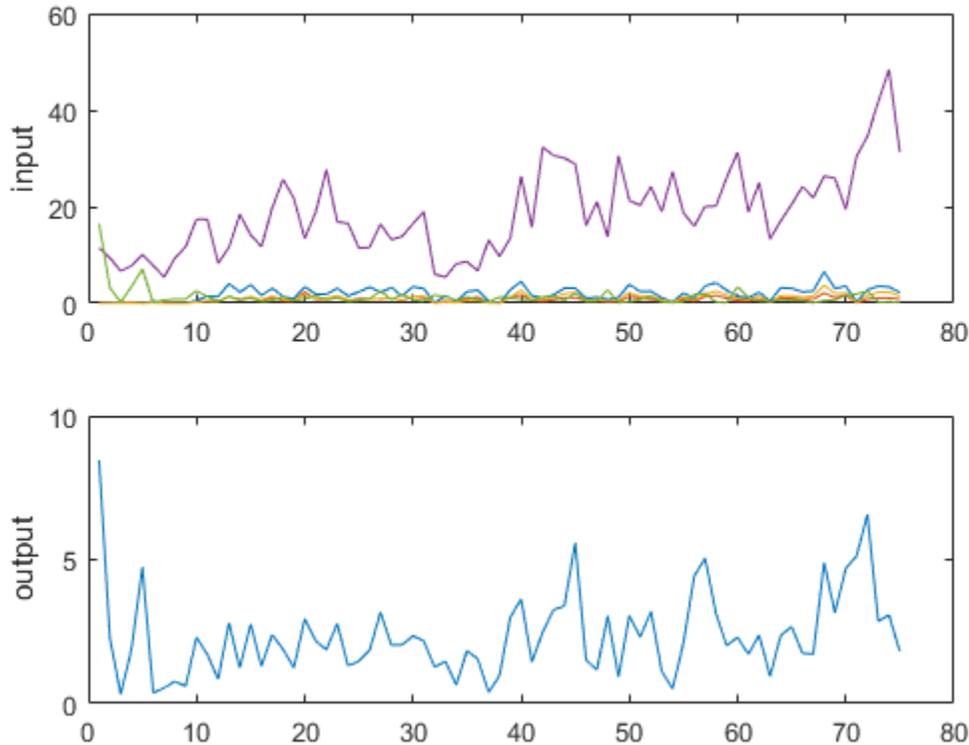
- “Fuzzy Clustering” on page 4-2
- “Cluster Quasi-Random Data Using Fuzzy C-Means Clustering” on page 4-4

Model Suburban Commuting Using Subtractive Clustering

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

Load 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 pairs are denoted by `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 structure is a first order Sugeno model with three rules.

Verify the model. Here, `trnRMSE` is the root mean square error of the system generated by the training data.

```
fuzout = evalfis(datin,fismat);  
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 square error of the system generated by the validation data.

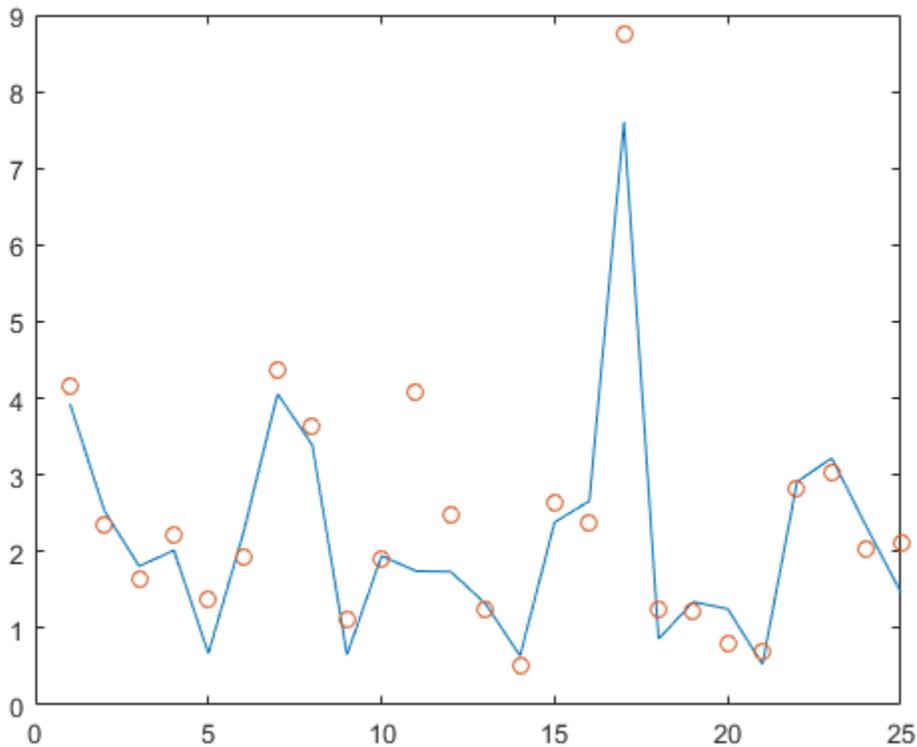
```
chkfuzout = evalfis(chkdatin,fismat);  
chkRMSE = norm(chkfuzout-chkdatout)/sqrt(length(chkfuzout))
```

```
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 `anfis` training (20 epochs) without using validation data, and then test the resulting FIS model against the testing data.

```
anfisOpt = anfisOptions('InitialFIS',fismat,'EpochNumber',20,...
                        'InitialStepSize',0.1);
fismat2 = anfis([datin datout],anfisOpt);
```

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(datin,fismat2);
```

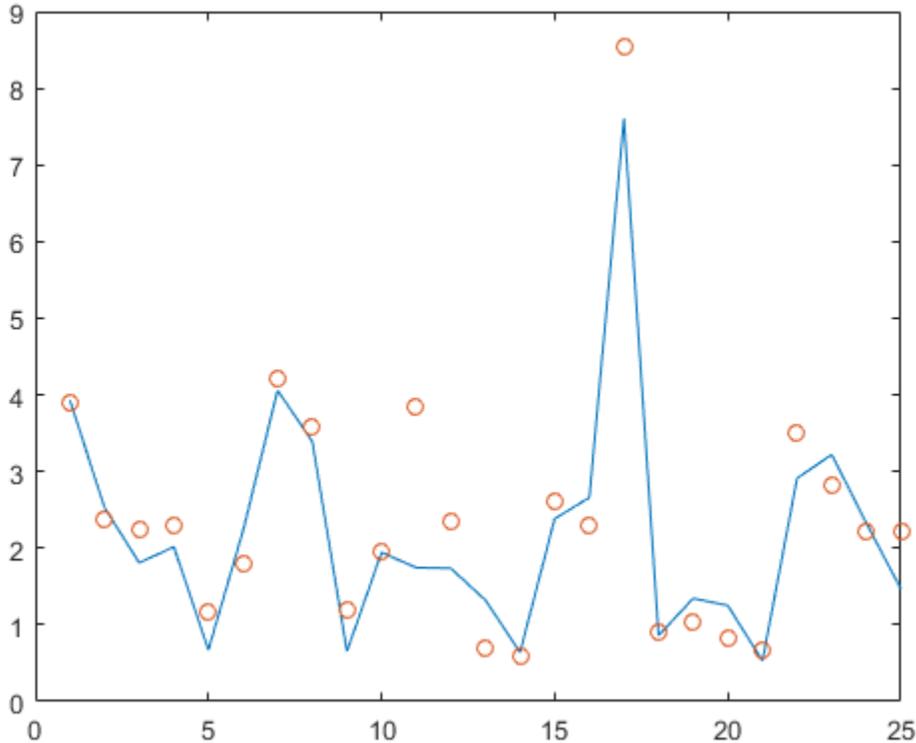
```
trnRMSE2 = norm(fuzout2-datout)/sqrt(length(fuzout2))
chkfuzout2 = evalfis(chkdatin,fismat2);
chkRMSE2 = norm(chkfuzout2-chkdatout)/sqrt(length(chkfuzout2))
```

```
trnRMSE2 =
    0.4203
```

```
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 an 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 structure when the training error reaches a minimum.
- `fismat4` is the snapshot FIS structure 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(datin, fismat4);  
trnRMSE4 = norm(fuzout4 - datout) / sqrt(length(fuzout4))  
chkfuzout4 = evalfis(chkdatin, fismat4);  
chkRMSE4 = norm(chkfuzout4 - chkdatout) / sqrt(length(chkfuzout4))
```

```
trnRMSE4 =
```

```
0.3393
```

```
chkRMSE4 =
```

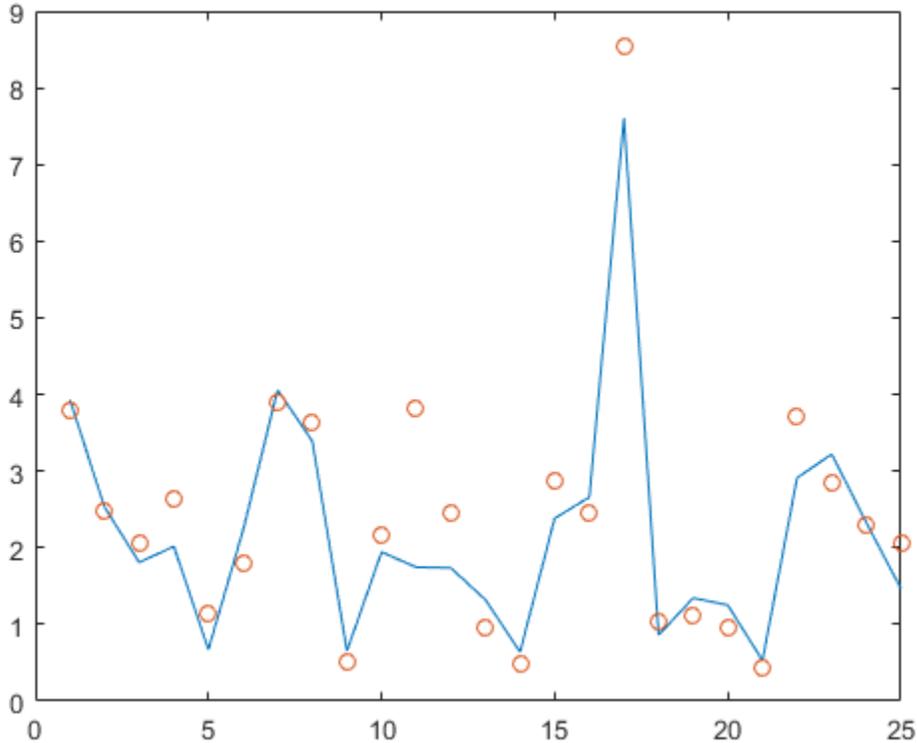
```
0.5834
```

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 perhaps there is an overfit of

the system to the training data. Overfitting 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.

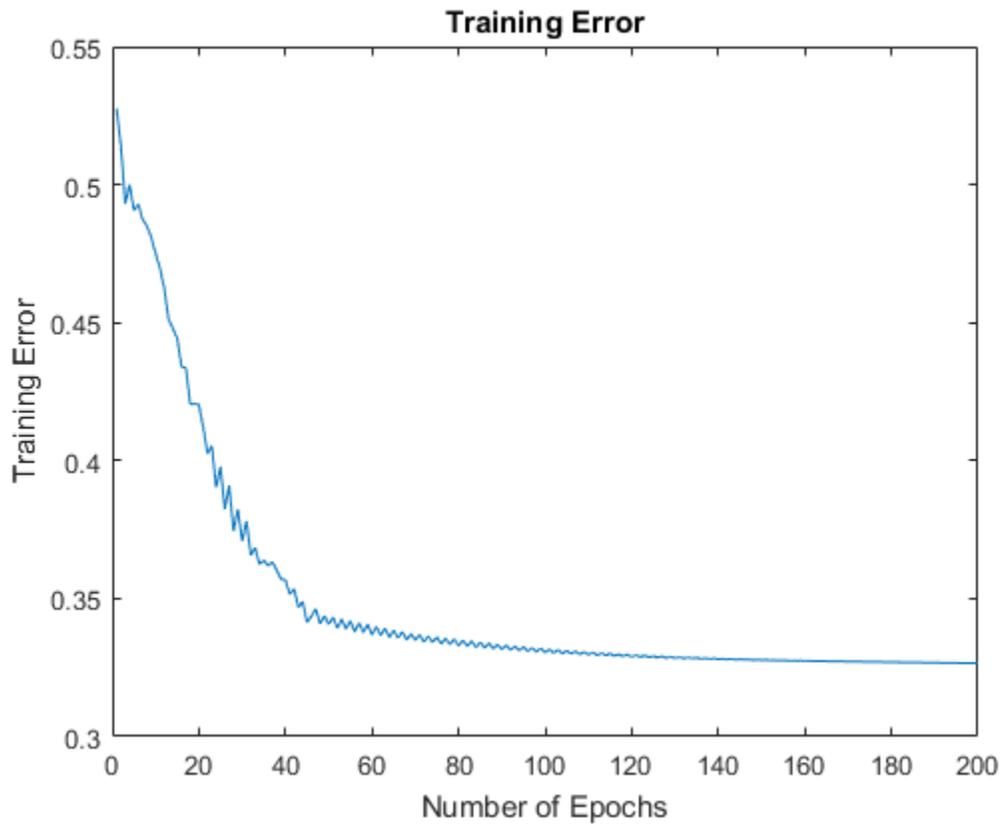
```
figure
plot(chkdatout)
hold on
plot(chkfuzout4, 'o')
hold off
```



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')
```

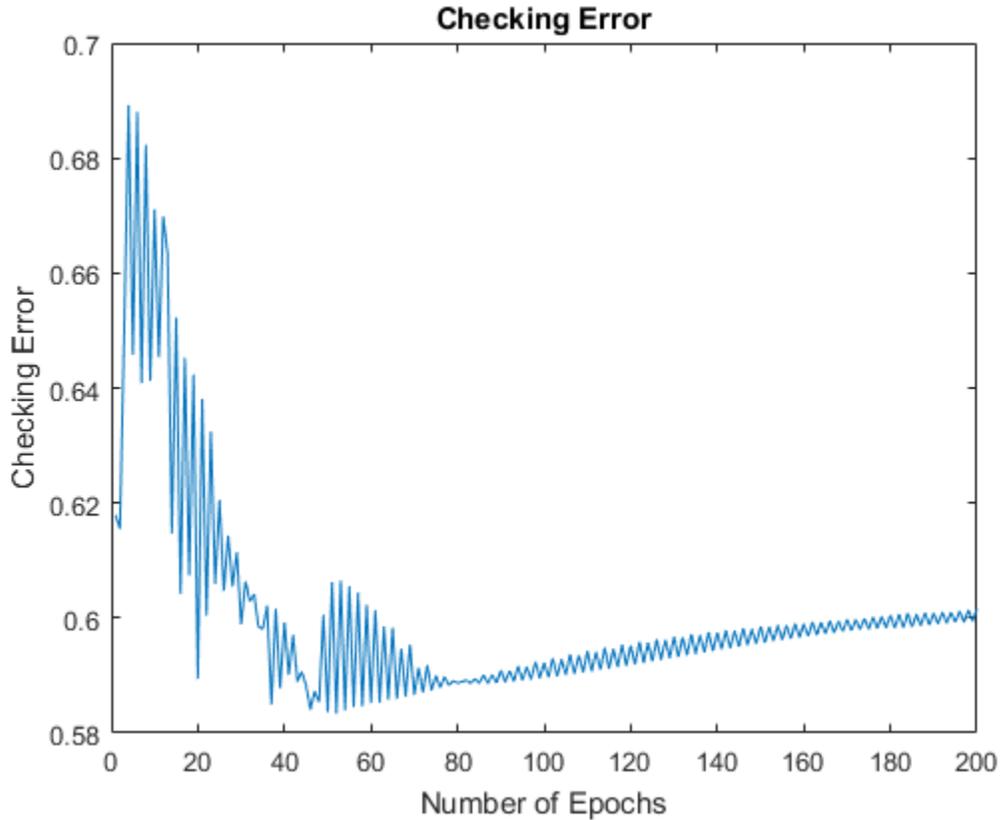


This plot shows that the training error settles at about the 60th epoch point.

Plot the checking error, `chkErr`.

```
figure
plot(chkErr)
```

```
title('Checking Error')
xlabel('Number of Epochs')
ylabel('Checking Error')
```

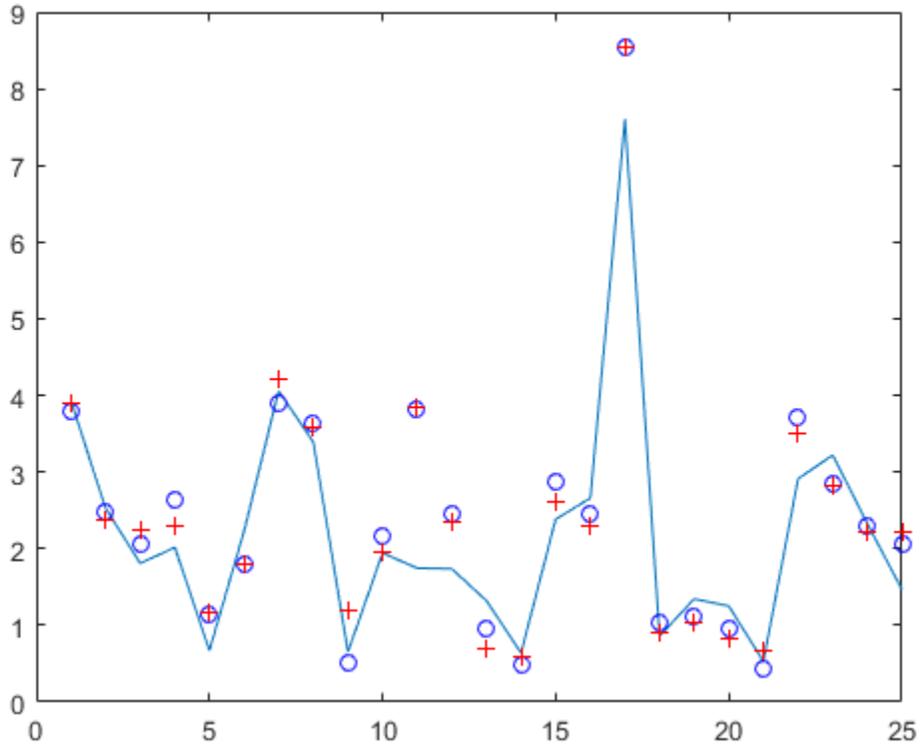


The plot shows that the smallest value of the validation data error occurs at the 52nd epoch, after which it increases slightly even as `anfis` continues to minimize the error against the training data all the way to the 200th epoch. Depending on the specified error tolerance, the plot also indicates 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

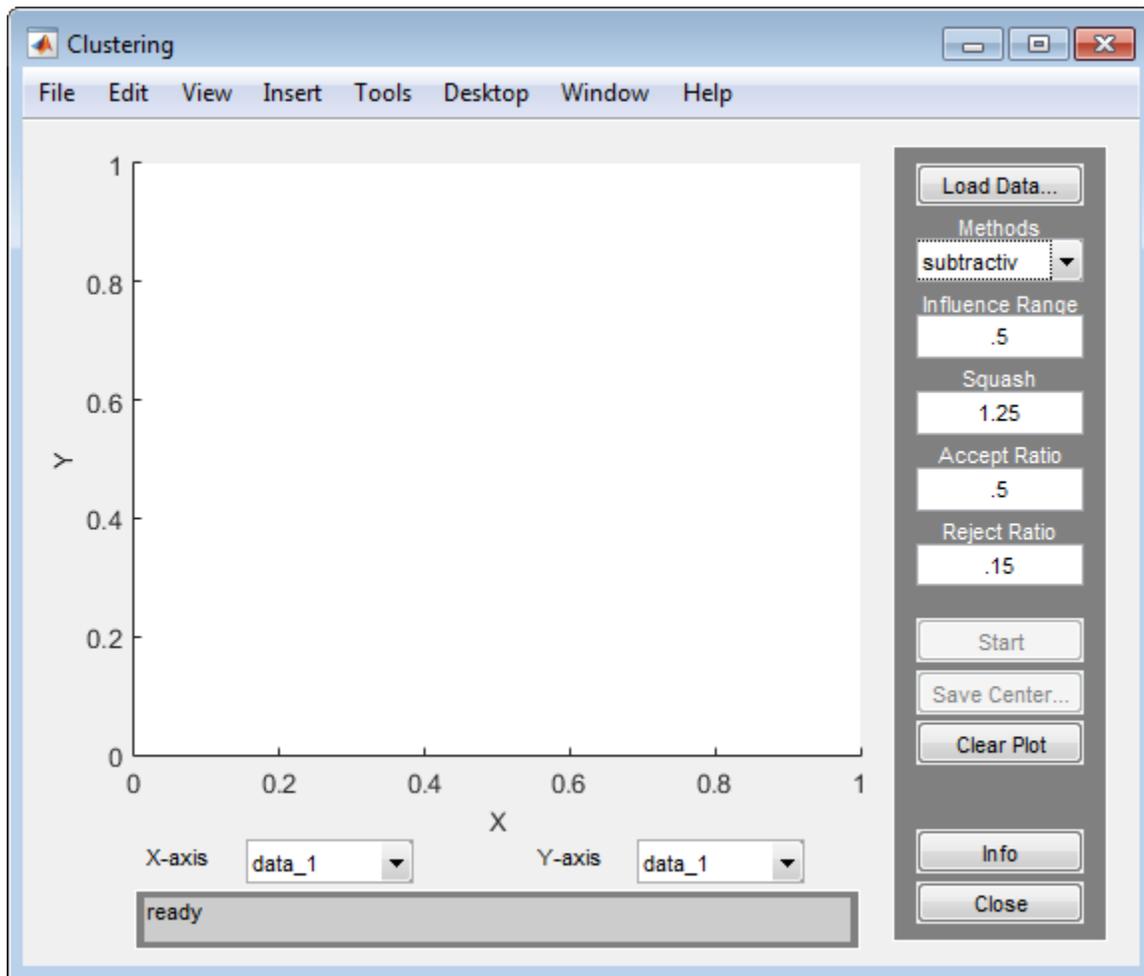
- “Fuzzy Clustering” on page 4-2

Data Clustering Using the 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 functions, see “Fuzzy Clustering” on page 4-2.

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

```
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 the 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, in the MATLAB Command Window.

The data set must have the extension `.dat`. For example, enter:

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

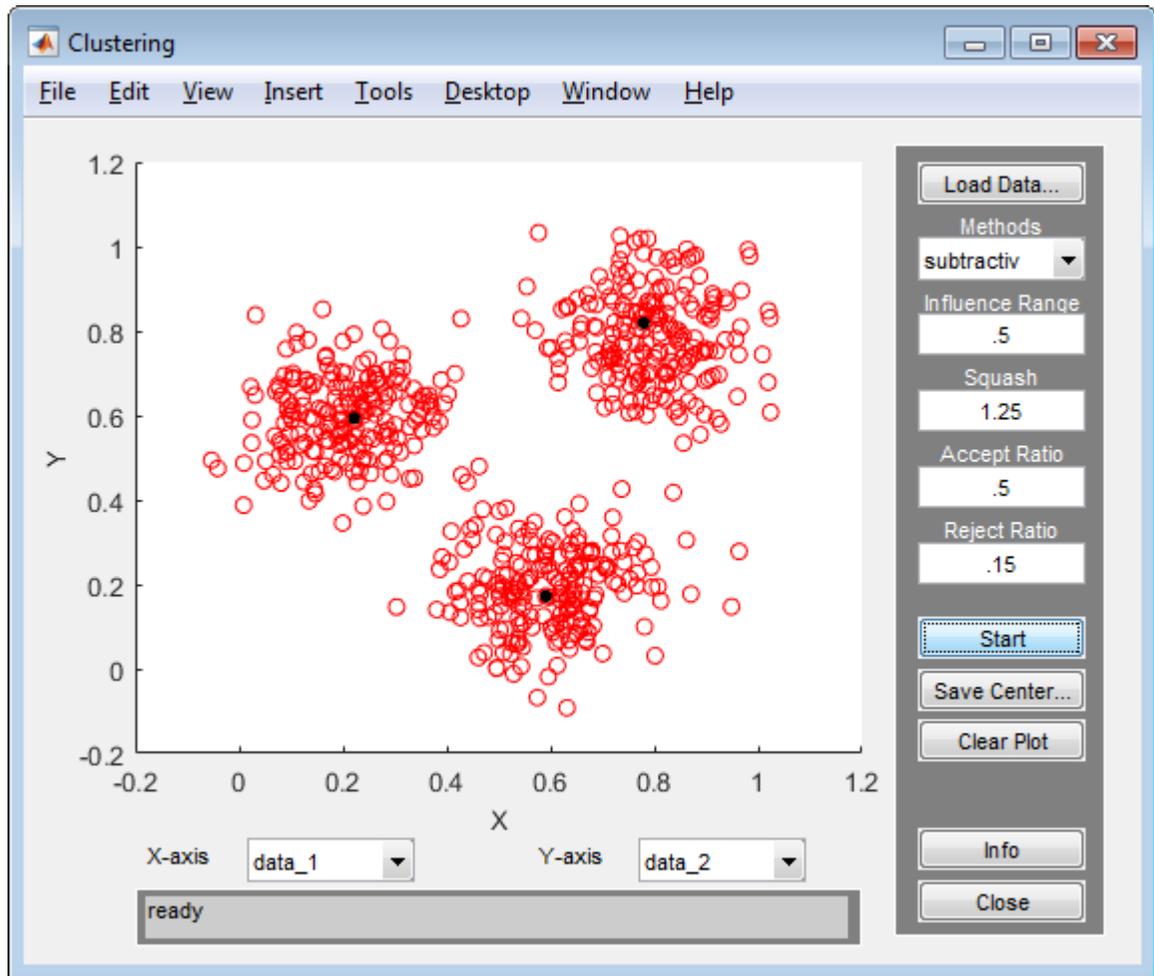
The Clustering Tool works on multidimensional data sets, but displays only two of those dimensions on the plot. To select other dimensions in the data set for plotting, you can use the drop-down lists under **X-axis** and **Y-axis**.

Perform the Clustering

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 the selected method using the **Influence Range**, **Squash**, **Aspect Ratio**, and **Reject Ratio** fields.
- 3 Begin clustering by clicking **Start**.

Once the clustering is complete, the cluster centers appear in black as shown in the next figure.



For more information on the clustering methods and their options, refer to `fcm` and `subclust`.

Save the Cluster Centers

To save the cluster centers, click **Save Center**.

See Also

fcM | findcluster | subclust

More About

- “Fuzzy Clustering” on page 4-2

Deployment

- “Fuzzy Inference Engine” on page 5-2
- “Compile and Evaluate Fuzzy Systems on Windows Platforms” on page 5-3
- “Compile and Evaluate Fuzzy Systems on UNIX Platforms” on page 5-6

Fuzzy Inference Engine

Fuzzy Logic Toolbox software provides a standalone C code fuzzy inference engine. You can use the engine as an alternative tool to evaluate the outputs of your fuzzy inference system (FIS), without using the MATLAB environment.

You can perform the following tasks using the fuzzy inference engine:

- Perform fuzzy inference using an FIS structure file and an input data file. To learn more about how to create an FIS structure file, see “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31.
- Customize the fuzzy inference engine to include your own membership functions.
- Embed the executable code in other external applications.

The standalone fuzzy inference engine consists of two C code source files, `fismain.c` and `fis.c`, in the `matlabroot/toolbox/fuzzy/fuzzy` folder.

The `fismain.c` file contains only the `main()` function and you can easily modify it for your own applications. It is ANSI C compatible, and you can compile it with any ANSI C compiler.

The `fis.c` file contains all the necessary functions to perform the fuzzy inference process:

- This file provides the 11 Fuzzy Logic Toolbox membership functions with their default settings.
- You can add different membership functions or reasoning mechanisms by updating the `fis.c` file.

The `fismain` executable program, generated after compiling the source code, reads an input data file and an FIS structure file to simulate the output. The syntax for calling `fismain` is similar to its counterpart `evalfis`, except that all matrices are replaced with files. To learn more about `evalfis`, see the `evalfis` function reference page.

More About

- “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31
- “Compile and Evaluate Fuzzy Systems on Windows Platforms” on page 5-3
- “Compile and Evaluate Fuzzy Systems on UNIX Platforms” on page 5-6

Compile and Evaluate Fuzzy Systems on Windows Platforms

You can compile the fuzzy inference engine using the LCC64 compiler or any other ANSI C compiler. After compilation, evaluate your fuzzy system for specified input data using the resulting standalone executable program.

Create Fuzzy Inference System

For this example, import the `mam21` system, which has two inputs and one output. At the MATLAB command line, enter:

```
sys = readfis('mam21');
```

For more information on creating fuzzy systems, see “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31.

Save the FIS structure `sys` to the file `fuzzySys.fis` in your current working folder.

```
writefis(sys, 'fuzzySys');
```

Create Input Data

Create input data for your simulation that covers the expected input range of your input variables.

```
[x,y] = meshgrid(-5:5,-5:5);  
input_data = [x(:) y(:)];
```

These commands specify the input data as a 121-by-2 matrix, where each row represents an input vector. For more information on input data requirements, see `evalfis`.

Save the input data to an ASCII file, `fis_in`, in your current working folder.

```
save fis_in input_data -ascii
```

Copy Source Files

Copy the `fismain.c` and `fis.c` source files for the fuzzy inference engine from the `matlabroot/toolbox/fuzzy/fuzzy` folder to your current working folder.

```
copyfile([matlabroot '/toolbox/fuzzy/fuzzy/fismain.c'], ['./fismain.c']);  
copyfile([matlabroot '/toolbox/fuzzy/fuzzy/fis.c'], ['./fis.c']);
```

Tip: To find the root folder of your MATLAB installation, type `matlabroot` at the MATLAB command line.

Compile Fuzzy Inference Engine

Open a Windows[®] command prompt, and change the folder to your current working folder.

Compile the fuzzy inference engine source code. A version of the LCC64 compiler ships with MATLAB.

```
matlabroot\sys\lcc64\lcc64\bin\lcc64 -c -Imatlabroot\sys\lcc64\lcc64\include64 fismain
```

This command creates the `fismain.obj` object file in your current working folder.

Create an executable file from the compiled code.

```
matlabroot\sys\lcc64\lcc64\bin\lcc64 -Lmatlabroot\sys\lcc64\lcc64\lib64 fismain.obj
```

This command creates the `fismain.exe` executable file in your current working folder.

After you have created the standalone executable file, you do not need to recreate it for each FIS system you want to simulate. To simulate a different FIS system, export its FIS structure and corresponding input data to the same folder as the executable file.

Note: If *matlabroot* contains spaces, enclose the file paths within double quotes. For example:

```
"matlabroot\sys\lcc64\lcc64\bin\lcc64" -c -I"matlabroot\sys\lcc64\lcc64\include64" fismain
```

Evaluate Fuzzy Inference System

Run the standalone executable using the input data, `fis_in`, and the FIS structure, `fuzzySys.fis`.

```
fismain fis_in fuzzySys.fis > fis_out
```

This command evaluates the fuzzy inference system for each input vector in `fis_in` and saves the results in `fis_out`.

Verify Fuzzy Inference Engine Output

Evaluate the fuzzy inference system using the previously created input data. At the MATLAB command line, enter:

```
matlab_out = evalfis(input_data,sys);
```

Load the results from the standalone fuzzy inference engine simulation.

```
load fis_out
```

Determine the maximum difference between the simulation results.

```
max(abs(fis_out-matlab_out))
```

```
ans =
```

```
4.9760e-13
```

The near-zero maximum value indicates that the compiled fuzzy inference engine simulation matches the MATLAB simulation using `evalfis`.

See Also

`evalfis` | `writefis`

More About

- “Fuzzy Inference Engine” on page 5-2
- “Compile and Evaluate Fuzzy Systems on UNIX Platforms” on page 5-6

Compile and Evaluate Fuzzy Systems on UNIX Platforms

You can compile the fuzzy inference engine using the CC compiler or any other ANSI C compiler. After compilation, evaluate your fuzzy system for specified input data using the resulting standalone executable program.

Create Fuzzy Inference System

For this example, import the `mam21` system, which has two inputs and one output. At the MATLAB command line, enter:

```
sys = readfis('mam21');
```

For more information on creating fuzzy systems, see “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31.

Save the FIS structure `sys` to the file `fuzzySys.fis` in your current working folder.

```
writefis(sys, 'fuzzySys');
```

Create Input Data

Create input data for your simulation that covers the expected input range of your input variables.

```
[x,y] = meshgrid(-5:5,-5:5);  
input_data = [x(:) y(:)];
```

These commands specify the input data as a 121-by-2 matrix, where each row represents an input vector. For more information on input data requirements, see `evalfis`.

Save the input data to an ASCII file, `fis_in`, in your current working folder.

```
save fis_in input_data -ascii
```

Copy Source Files

Copy the `fismain.c` and `fis.c` source files for the fuzzy inference engine from the `matlabroot/toolbox/fuzzy/fuzzy` folder to your current working folder.

```
copyfile([matlabroot '/toolbox/fuzzy/fuzzy/fismain.c'], ['./fismain.c']);  
copyfile([matlabroot '/toolbox/fuzzy/fuzzy/fis.c'], ['./fis.c']);
```

Tip: To find the root folder of your MATLAB installation, type `matlabroot` at the MATLAB command line.

Compile Fuzzy Inference Engine

Open a UNIX[®] terminal window, and change the folder to your current working folder.

Compile the fuzzy inference engine source code.

```
cc -o fismain fismain.c -lm
```

This command creates the `fismain` executable file in your current working folder.

After you have created the standalone executable file, you do not need to recreate it for each FIS system you want to simulate. To simulate a different FIS system, export its FIS structure and corresponding input data to the same folder as the executable file.

Evaluate Fuzzy Inference System

Run the standalone executable using the input data, `fis_in`, and the FIS structure, `fuzzySys.fis`.

```
./fismain fis_in fuzzySys.fis > fis_out
```

This command evaluates the fuzzy inference system for each input vector in `fis_in` and saves the results in `fis_out`.

Verify Fuzzy Inference Engine Output

Evaluate the fuzzy inference system using the previously created input data. At the MATLAB command line, enter:

```
matlab_out = evalfis(input_data,sys);
```

Load the results from the standalone fuzzy inference engine simulation.

```
load fis_out
```

Determine the maximum difference between the simulation results.

```
max(abs(fis_out-matlab_out))
```

```
ans =
```

4.9760e-13

The near-zero maximum value indicates that the compiled fuzzy inference engine simulation matches the MATLAB simulation using `evalfis`.

See Also

`evalfis` | `writefis`

More About

- “Fuzzy Inference Engine” on page 5-2
- “Compile and Evaluate Fuzzy Systems on Windows Platforms” on page 5-3

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 Mamdani Systems Using Fuzzy Logic Designer” on page 2-31

Programmatic Use

`fuzzyLogicDesigner` opens the **Fuzzy Logic Designer** app.

`fuzzyLogicDesigner(fuzzySys)` opens the app and loads the fuzzy inference system `fuzzySys`. `fuzzySys` can be any fuzzy inference system structure 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`.

See Also

See Also

Apps

Neuro-Fuzzy Designer

Functions

`evalfis` | `mfedit` | `newfis` | `plotfis` | `ruleedit` | `ruleview` | `surfview`

Topics

“Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31

“What Is Fuzzy Logic?” on page 1-3

“Foundations of Fuzzy Logic” on page 2-2

“Fuzzy Inference Process” on page 2-22

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(fuzzySys)` opens the app and loads the fuzzy inference system `fuzzySys`. `fuzzySys` can be any valid Sugeno-type fuzzy system structure 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`.

See Also

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

addmf

Add membership function to fuzzy inference system

Syntax

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

Description

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

The function requires six input arguments:

- **inFIS** — Fuzzy inference system in the MATLAB workspace, specified as an FIS structure.
- **varType** — Type of variable to add membership function to, specified as 'input' or 'output'.
- **varIndex** — Index of variable to add membership function to, specified as a positive integer less than or equal to the number of variables of the type specified by **varType**.
- **mfName** — The name of the new membership function, specified as a character vector or string.
- **mfType** — The type of the new membership function, specified as one of the following:

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

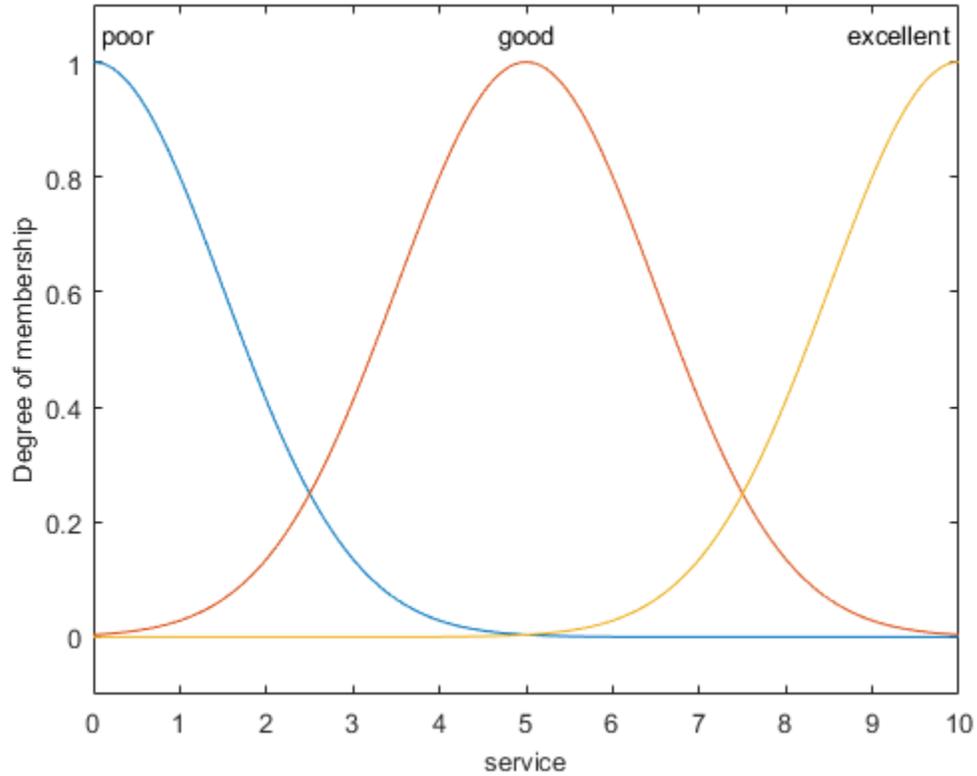
Membership function type	Description	For more information
'trimf'	Triangular membership function	trimf
'trapmf'	Trapezoidal membership function	trapmf
'sigmf'	Sigmoidal membership function	sigmf
'dsigmf'	Difference between two sigmoidal membership functions	dsigmf
'psigmf'	Product of two sigmoidal membership functions	psigmf
'zmf'	Z-shaped membership function	zmf
'pimf'	Pi-shaped membership function	pimf
'smf'	S-shaped membership function	smf
Character vector or string	Custom membership function in the current working folder or on the MATLAB path	“Build Mamdani Systems Using Custom Functions” on page 2-55

- mfParams — The parameters of the new membership function, specified as an array.

Examples

Add Membership Functions to FIS Inputs

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



See Also

See Also

`addrule` | `addvar` | `plotmf` | `rmmf` | `rmvar`

Introduced before R2006a

addrule

Add rule to fuzzy inference system

Syntax

```
fis = addrule(fis,ruleList)
```

Description

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

The first m columns refer to the inputs of the system, and the next n columns refer to the outputs of the system. Each column contains a number that refers to the index of the membership function for that variable. To indicate a NOT condition for an input or output variable, use a negative number. To indicate a don't care condition, use a 0.

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

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

For information on how to delete rules from an FIS, see “Tips” on page 7-6.

Examples

Add Rules to Fuzzy Inference System

Create the fuzzy inference system. For this example, load the `tipper` FIS and clear the existing rules for the FIS.

```
sys = readfis('tipper');
```

```
sys.rule = [];
```

Create a rule list.

```
rule1 = [1 1 1 1 1];  
rule2 = [1 2 2 1 1];  
ruleList = [rule1;rule2];
```

Add the rule list to the FIS.

```
sys = addrule(sys,ruleList);
```

Verify that the rules were added.

```
showrule(sys)
```

```
ans =
```

```
2×73 char array
```

```
'1. If (service is poor) and (food is rancid) then (tip is cheap) (1)      '  
'2. If (service is poor) and (food is delicious) then (tip is average) (1)'
```

Tips

- To delete a rule from an FIS, set the corresponding element of the rule list to an empty matrix, []. For example, load the `tipper` FIS and view the existing rule list.

```
sys = readfis('tipper');  
showrule(sys)
```

```
ans =
```

```
3×78 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)'
```

Delete the second rule by setting it to [].

```
sys.rule(2) = [];
```

Verify that the rule was deleted.

```
showrule(sys)
```

```
ans =
```

```
2×78 char array
```

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

```
'2. If (service is excellent) or (food is delicious) then (tip is generous) (1)'
```

- To delete all rules from an FIS, set the rule list to an empty matrix. For example:

```
sys.rule = [];
```

See Also

See Also

addmf | addvar | parsrule | rmmf | rmvar | showrule

Topics

“Build Mamdani Systems Using Custom Functions” on page 2-55

Introduced before R2006a

addvar

Add variable to fuzzy inference system

Syntax

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

Description

addvar has four input arguments:

- **fis** — Fuzzy inference system in the MATLAB workspace, specified as an 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, so 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]
```

See Also

See Also

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 structure 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 structure 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 structure 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.
```

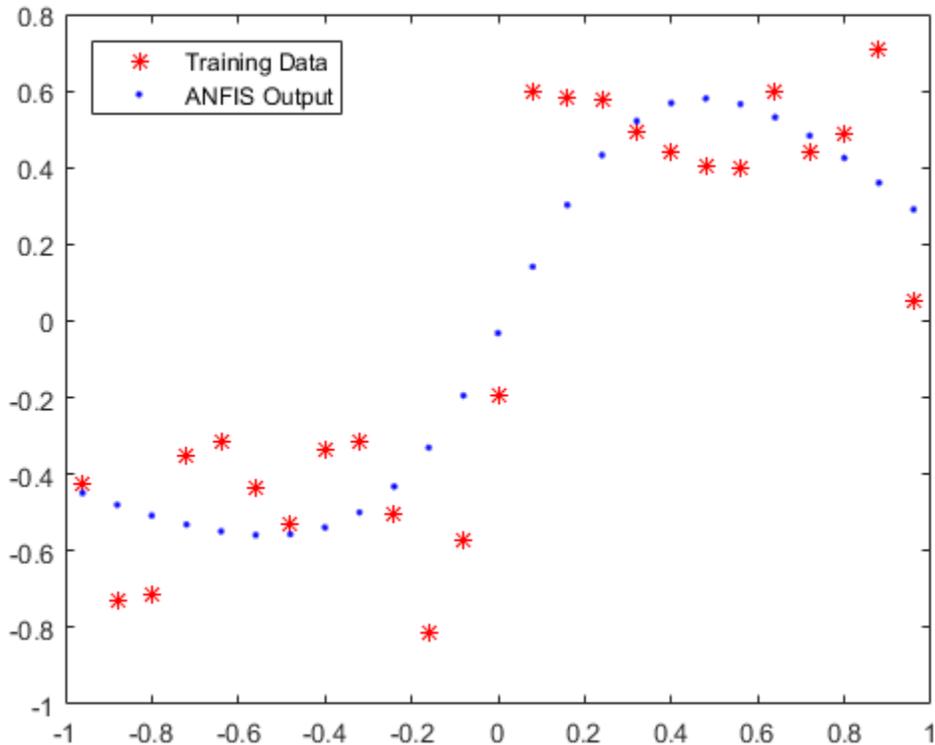
```
10 0.22465
```

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(x,fis);
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);
```

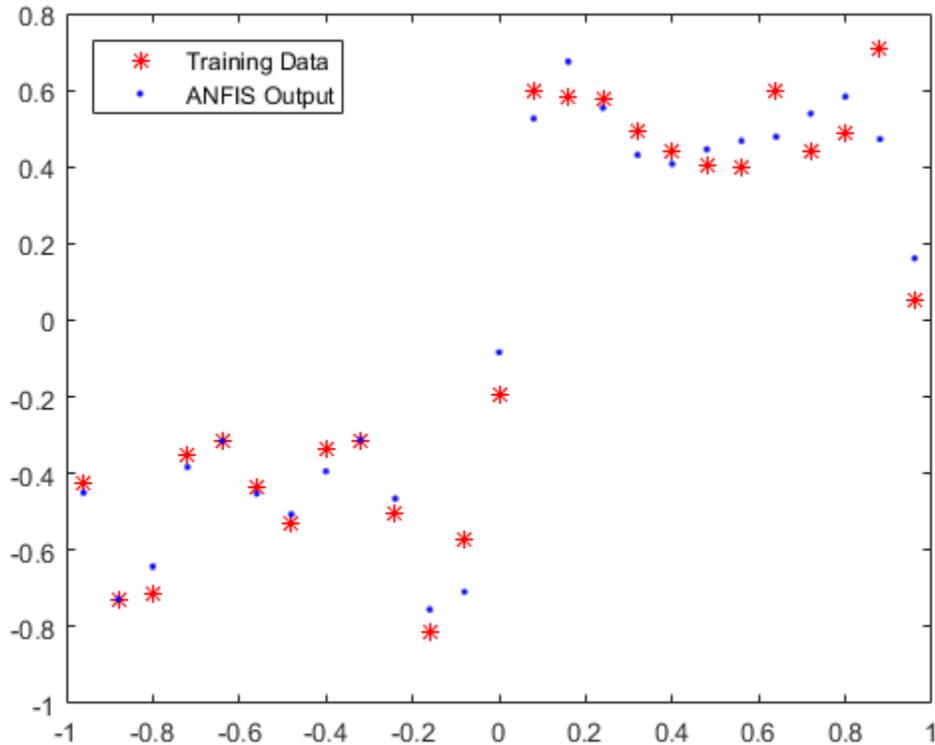
ANFIS info:

```
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(x,fis);  
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.

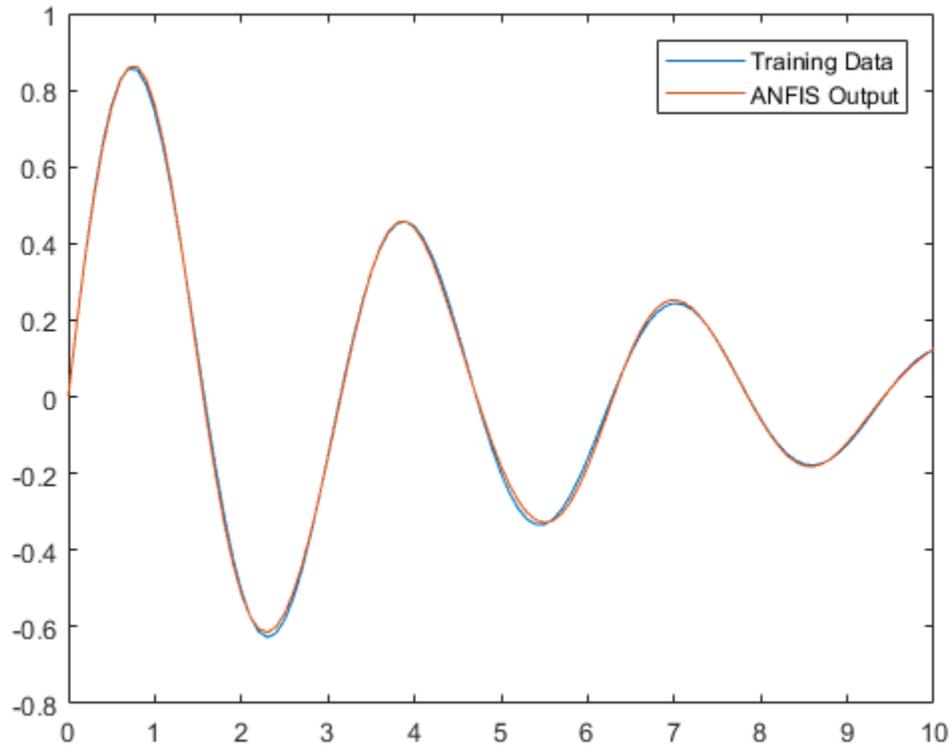
```
opt = anfisOptions('InitialFIS',inFIS);  
opt.DisplayANFISInformation = 0;  
opt.DisplayErrorValues = 0;  
opt.DisplayStepSize = 0;  
opt.DisplayFinalResults = 0;
```

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(x,outFIS))  
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 square 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;
```

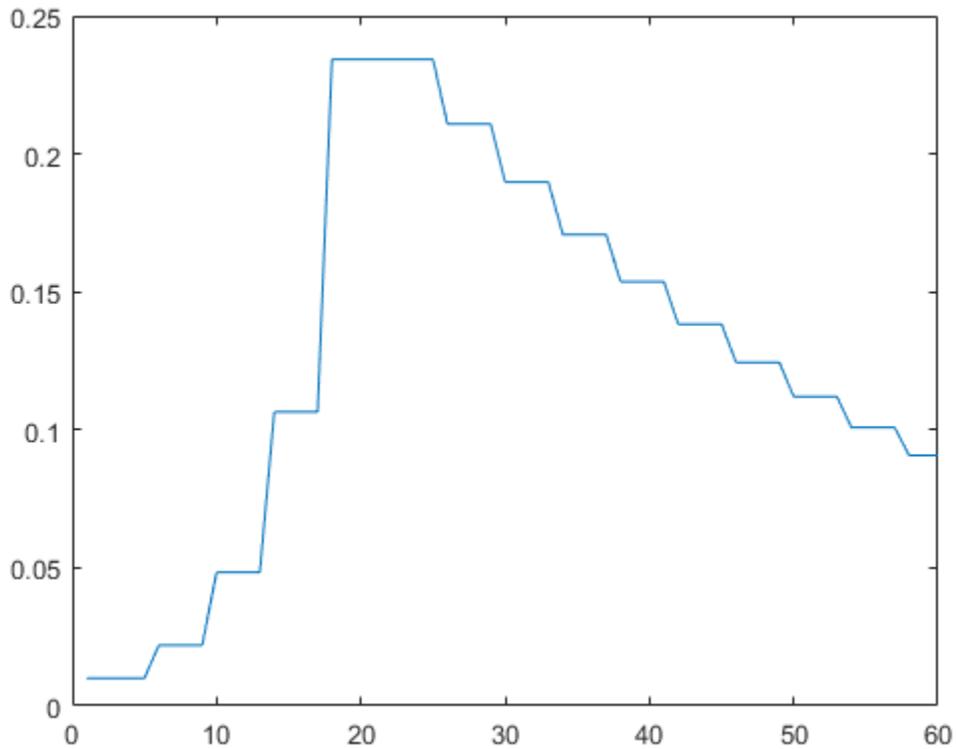
Train the FIS, and return the step size array.

```
[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.

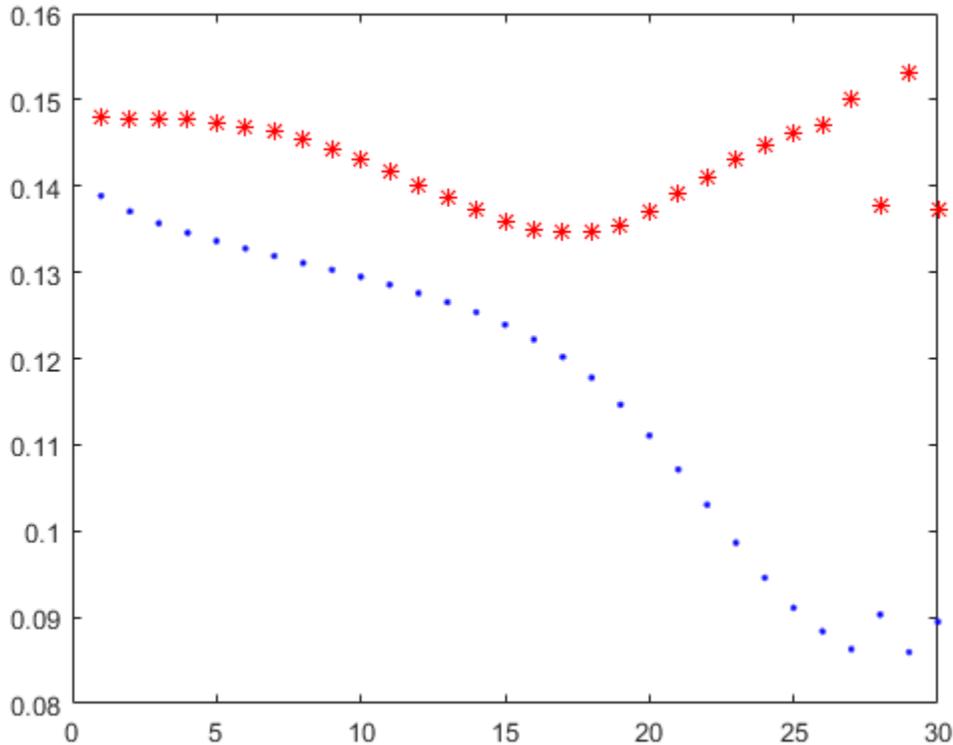
```
opt.ValidationData = fuzex1chkData;
```

Train the FIS, and return the validation results.

```
[fis,trainError,stepSize,chkFIS,chkError] = anfis(fuzex1trnData,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

FIS structure

Trained fuzzy inference system with membership function parameters tuned using the training data, returned as an FIS structure. 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

FIS structure

Tuned FIS for which the validation error is minimum, returned as an FIS structure. 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`.

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

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 3-51

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 single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, . . . , **NameN**, **ValueN**.

Example:

'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

See Also

anfis | genfis

Introduced in R2017a

convertfis

Convert Fuzzy Logic Toolbox Version 1.0 Fuzzy Inference System matrix to current-version Fuzzy Inference System structure

Syntax

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

Description

`convertfis` takes a Version 1.0 FIS matrix and converts it to an FIS structure compatible with the current version.

Introduced before R2006a

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 Mamdani Systems Using Custom Functions” on page 2-55.

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

See Also

Fuzzy Logic Designer

Topics

“Fuzzy Inference Process” on page 2-22

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, `a1`, `c1`, `a2`, and `c2`, and is the difference between two of these sigmoidal functions.

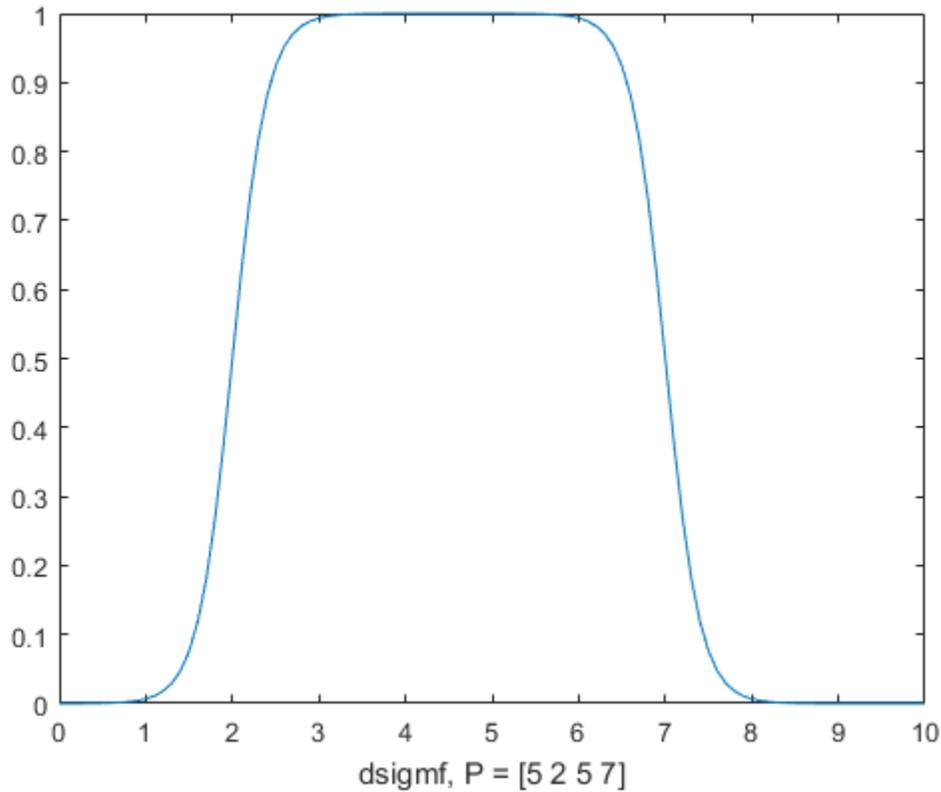
$$f_1(x; a_1, c_1) - f_2(x; a_2, c_2)$$

The parameters are listed in the order: `[a1 c1 a2 c2]`.

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]')
```



See Also

See Also

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

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

Introduced before R2006a

evalfis

Perform fuzzy inference calculations

Syntax

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

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

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

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

Description

`evalfis` has the following arguments:

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

The range labels for `evalfis` are as follows:

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

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

- **IRR**: the result of evaluating the input values through the membership functions. This matrix is of the size *numRules*-by-*N*, where *numRules* is the number of rules, and *N* is the number of input variables.

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

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

Examples

Evaluate FIS

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

```
out =
```

```
    7.0169  
   19.6810
```

See Also

See Also

[gensurf](#) | [ruleview](#)

Topics

“Fuzzy Inference Process” on page 2-22

Introduced before R2006a

evalmf

Generic membership function evaluation

Syntax

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

Description

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

You can specify `mfType` as any of the following membership function 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

Membership function type	Description	For more information
'smf'	S-shaped membership function	smf
Character vector or string	Custom membership function in the current working folder or on the MATLAB path	“Build Mamdani Systems Using Custom Functions” on page 2-55

Examples

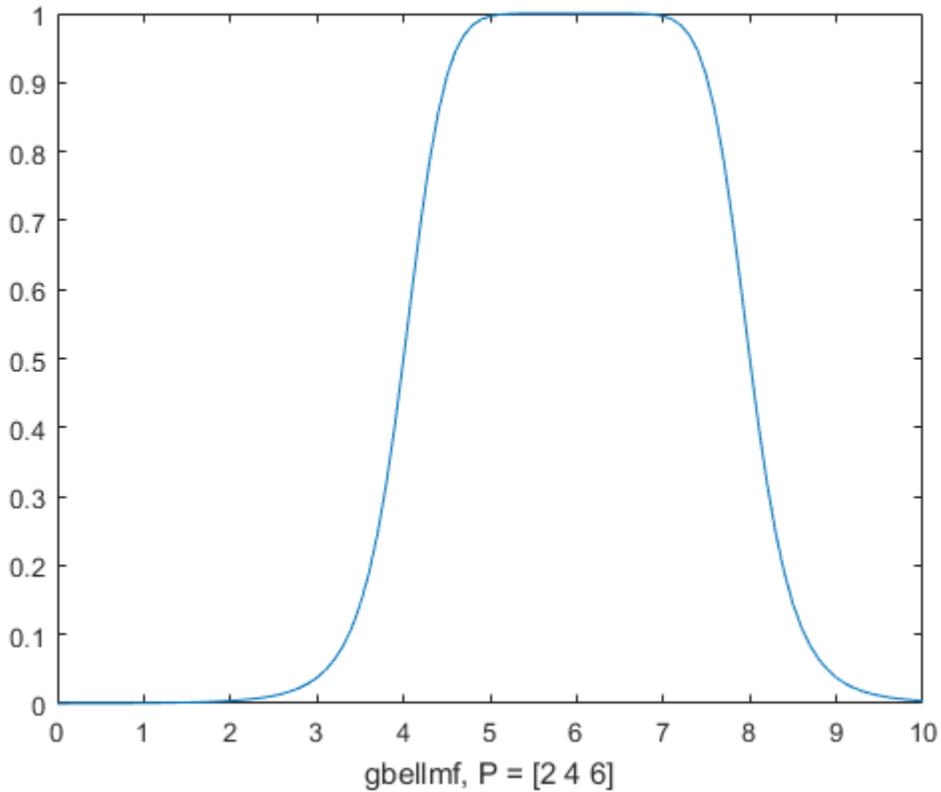
Evaluate Membership Function

Evaluate a generalized bell-shaped membership function.

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

Plot the evaluation.

```
plot(x,y)  
xlabel('gbellmf, P = [2 4 6]')
```



See Also

See Also

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

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

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
```

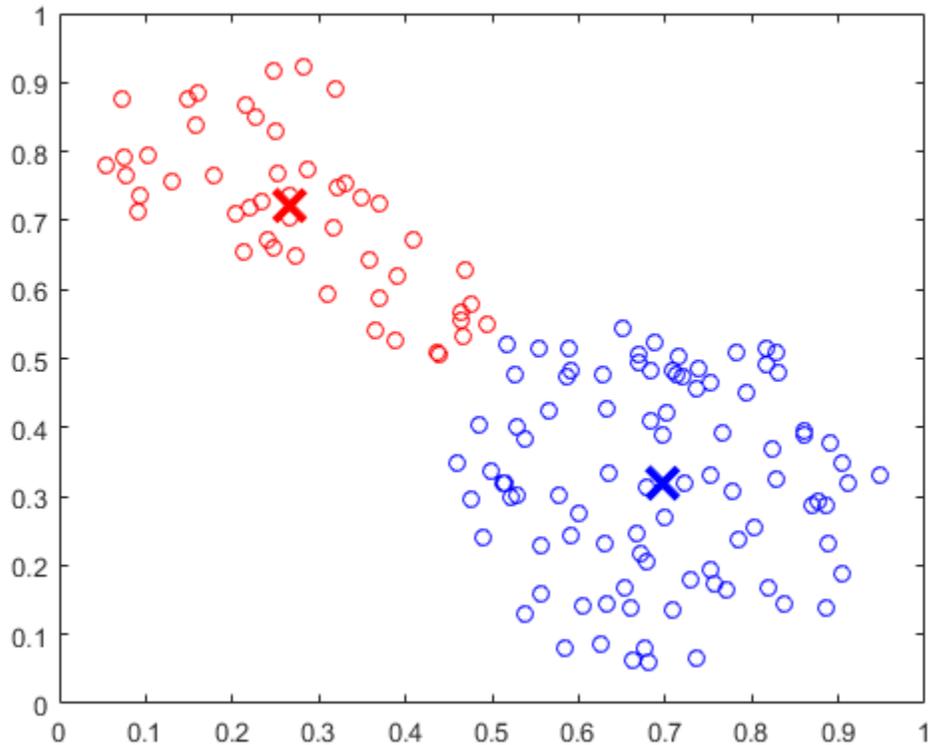
```
Iteration count = 3, obj. fcn = 6.325579
Iteration count = 4, obj. fcn = 4.586142
Iteration count = 5, obj. fcn = 3.893114
Iteration count = 6, obj. fcn = 3.810804
Iteration count = 7, obj. fcn = 3.799801
Iteration count = 8, obj. fcn = 3.797862
Iteration count = 9, obj. fcn = 3.797508
Iteration 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(fcmdata(index1,1),fcmdata(index1,2),'ob')
hold on
plot(fcmdata(index2,1),fcmdata(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);
```

Specify a large fuzzy partition matrix exponent to increase the amount of fuzzy overlap between the clusters.

```
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);
```

View the objective function vector to determine which termination condition stopped the clustering.

```
objFun
```

```
objFun =
```

```
54.7257  
42.9867  
42.8554  
42.1857  
39.0857  
31.6814  
28.5736  
27.1806  
20.7359  
15.7147  
15.4353  
15.4306  
15.4305
```

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(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. 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. For an example of fuzzy overlap adjustment, see “Adjust Fuzzy Overlap in Fuzzy C-Means Clustering” on page 4-8.	2.0
options(Maximum number of iterations, specified as a positive integer.	100
options(Minimum improvement in objective function between two consecutive iterations, specified as a positive scalar.	1e-5

Option	Description	Default
<code>options</code>	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"> <code>true</code> — Display objective function. <code>false</code> — Do not display objective function. 	<code>true</code>

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 `NC` 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 `NC` 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

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
```

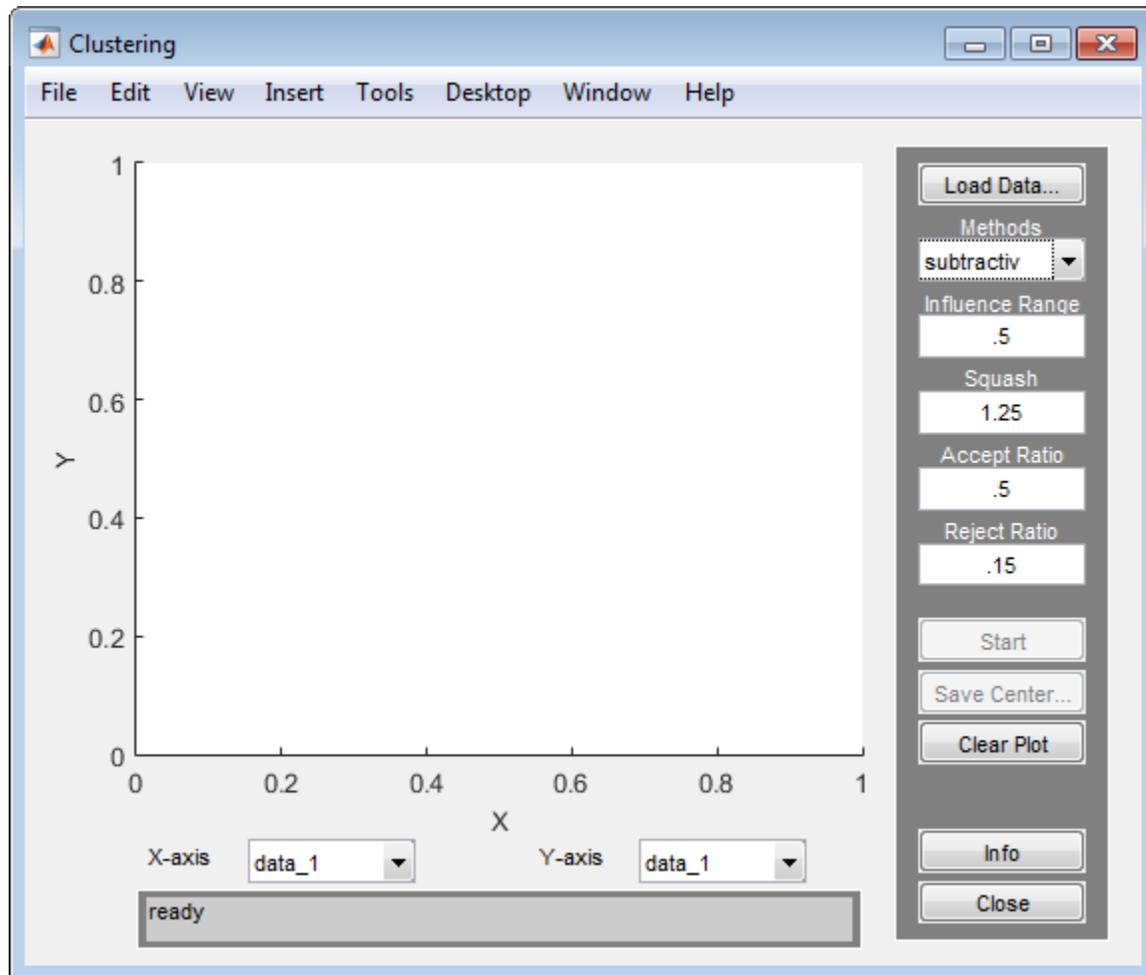
```
findcluster(fileName)
```

Description

`findcluster` opens a GUI to implement either fuzzy c-means (**fc**m), fuzzy subtractive clustering (**sub**tractiv) using the pull-down tab under **Method** on the GUI,. Data is entered using the **Load Data** button. The options for each of these methods are set to default values. These default values can be changed. See the `fc`m reference page for a description of the options for fuzzy c-means. The `sub`clust reference page provides a description of the options for subtractive clustering.

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

`findcluster(fileName)` opens the GUI, loads the data set in the file `fileName`, specified as a character vector or string, and plots the first two dimensions of the data. You can choose which two dimensions of the data you want to view after the GUI appears.



Examples

Open Clustering Tool and Load Data Set

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

See Also

See Also

fcM | subclust

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]);
```

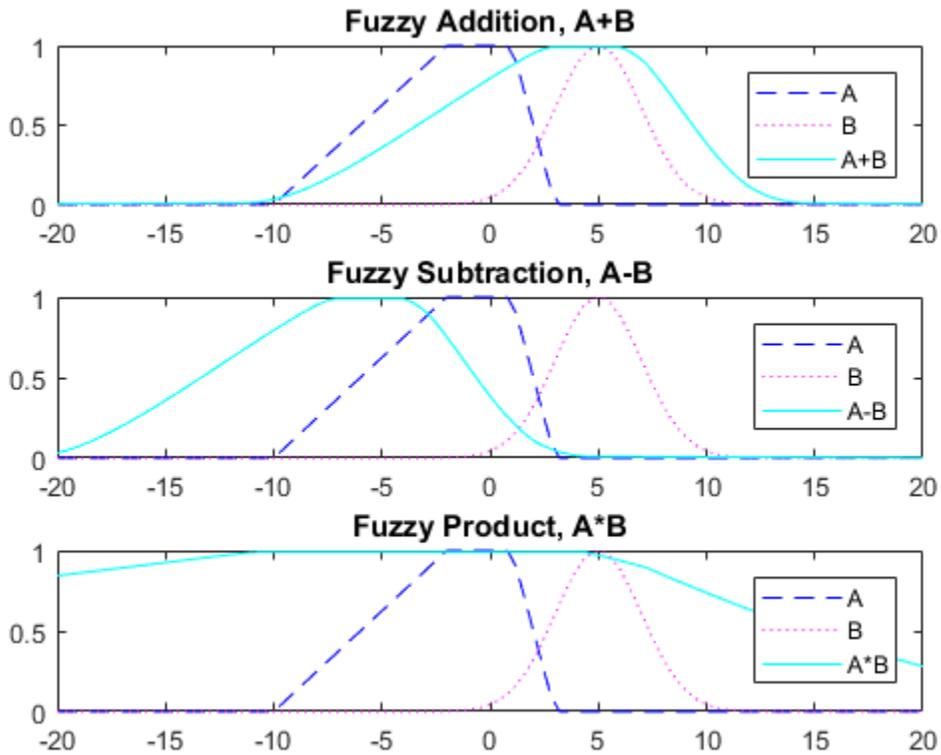
```
B = gaussmf(x,[2 5]);
```

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

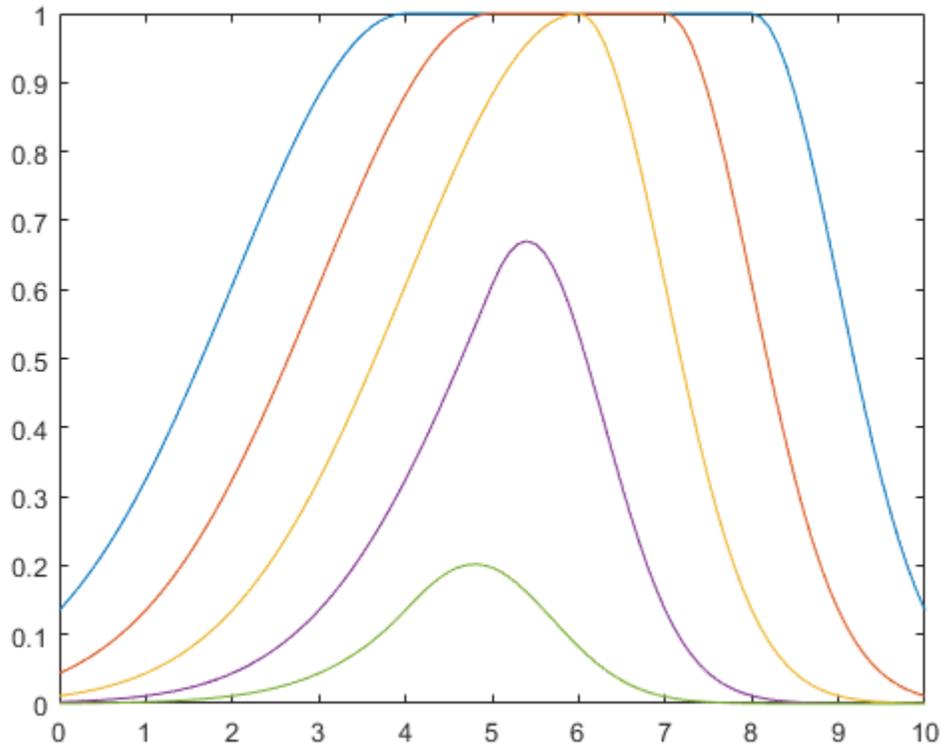
$$f(x; \sigma, c) = e^{-\frac{(x-c)^2}{2\sigma^2}}$$

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])
```



See Also

See Also

`dsigmf` | `evalmf` | `gaussmf` | `gbellmf` | `mf2mf` | `pimf` | `psigmf` | `sigmf` | `smf` | `trapmf` | `trapmf` | `trimf` | `trimf` | `zmf`

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

Introduced before R2006a

gaussmf

Gaussian curve membership function

Syntax

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

Description

The symmetric Gaussian function depends on two parameters σ and c as given by

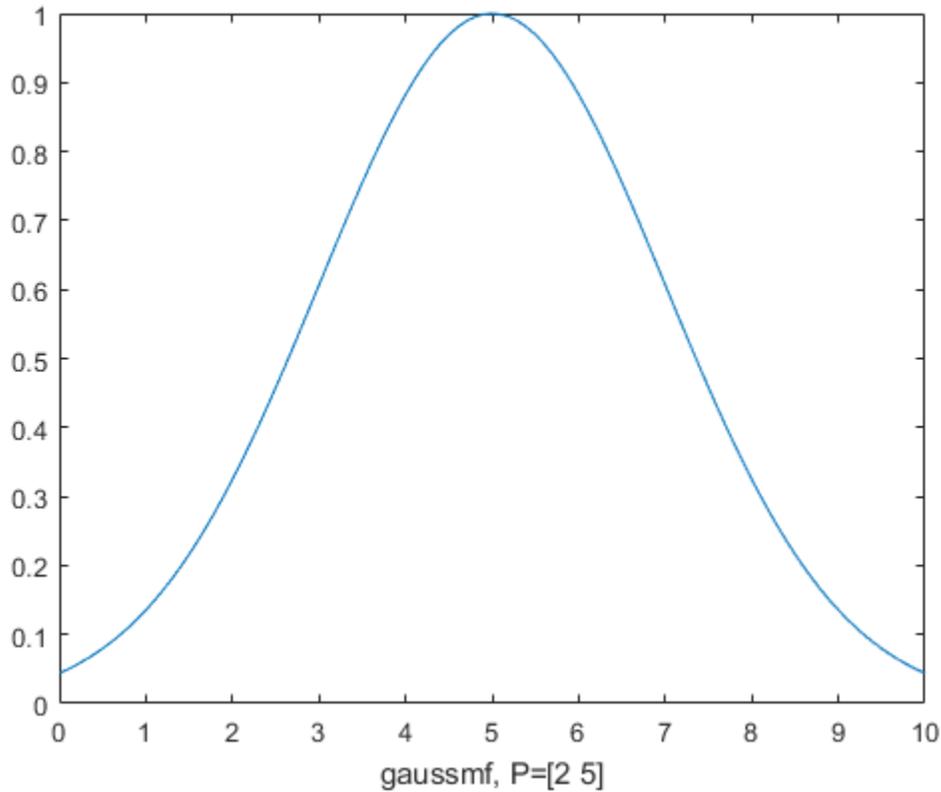
$$f(x;\sigma,c) = e^{-\frac{(x-c)^2}{2\sigma^2}}$$

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]')
```



See Also

See Also

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

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

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

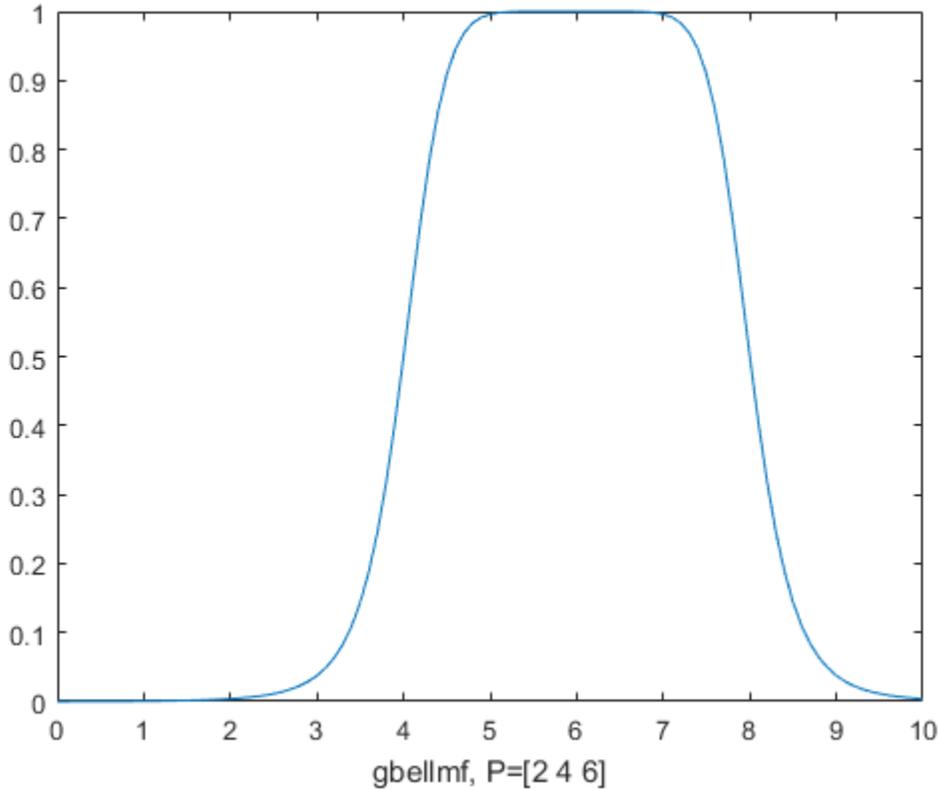
$$f(x;a,b,c) = \frac{1}{1 + \left| \frac{x-c}{a} \right|^{2b}}$$

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]')
```



See Also

See Also

`dsigmf` | `evalmf` | `gauss2mf` | `gaussmf` | `mf2mf` | `pimf` | `psigmf` | `sigmf` | `smf` | `trapmf` | `trapmf` | `trimf` | `trimf` | `zmf`

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

Introduced before R2006a

genfis

Generate fuzzy inference system structure 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];
inputData = rand(10,1);
```

Generate a fuzzy inference system.

```
fis = genfis(inputData,inputData);
```

The generated system, `fis`, is generated 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

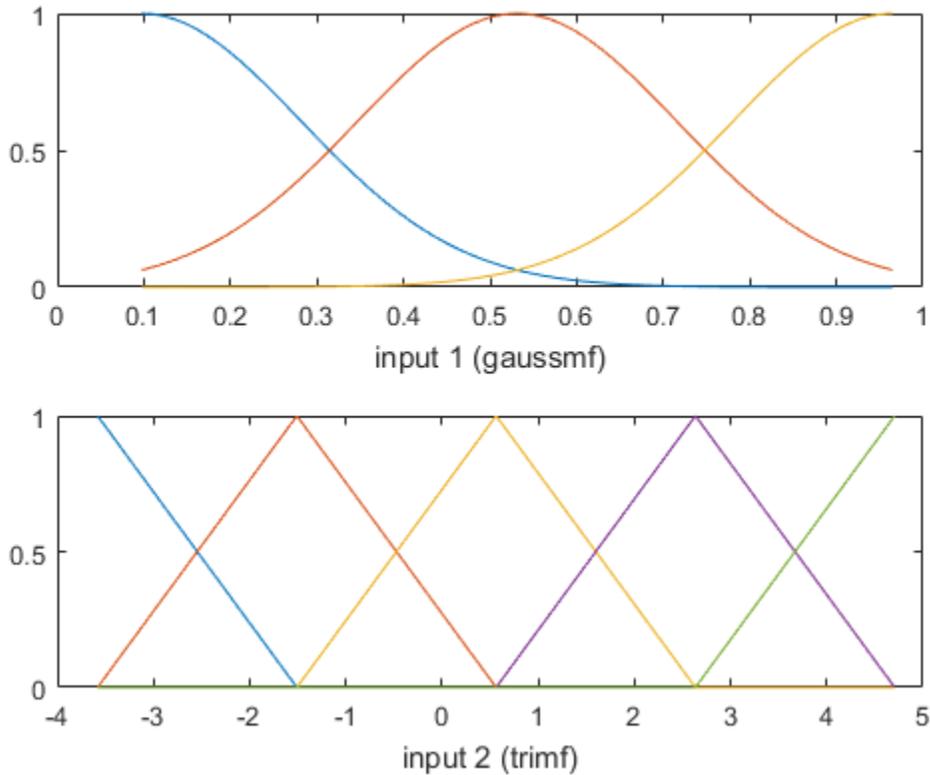
```
opt.NumMembershipFunctions = [3 5];  
opt.InputMembershipFunctionType = ["gaussmf" "trimf"];
```

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 =
```

```
3×83 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)
```

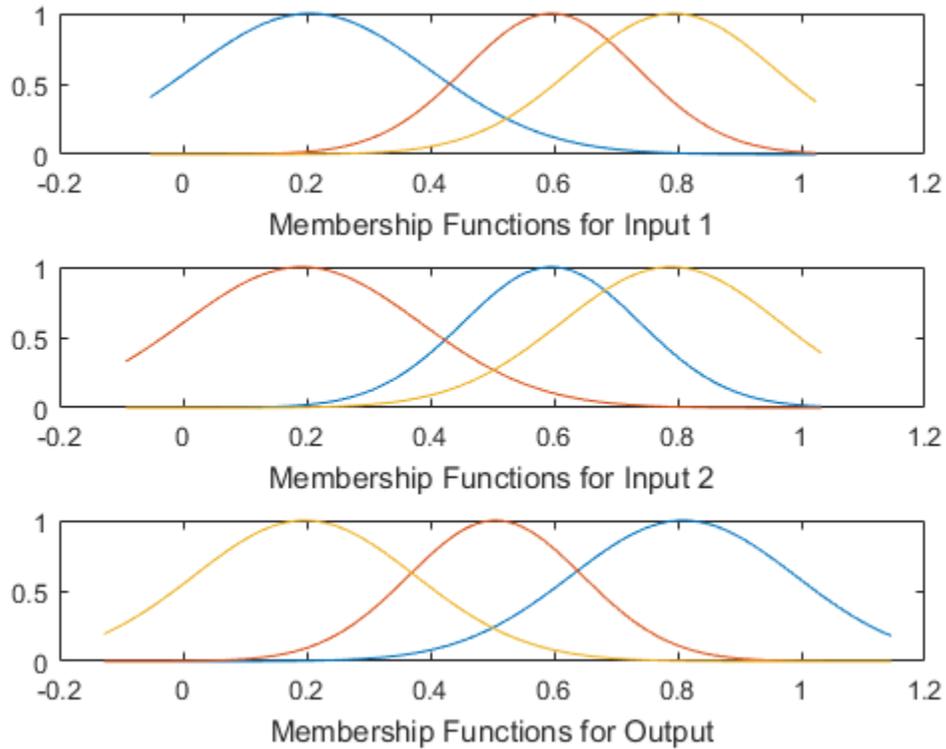
```
ans =
```

```
3×83 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

FIS structure

Fuzzy inference system, returned as an FIS structure. The properties of `fis` depend on the type of clustering used and the corresponding options.

Cluster Type	Fuzzy System Type	Input Membership Functions	Fuzzy Rules	Output Membership Functions
Grid Partition	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.InputMembershipFunction</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.OutputMembershipFunction</code> .
Subcluster	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 Cluster	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.

See Also

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.

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)')
```

See Also

See Also

anfis | 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.

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., `Xin` has two columns and `Xout` has one column), `radii = [0.5 0.4 0.3]` specifies that the ranges of influence in the first, second, and third data dimensions (i.e., the first column of `Xin`, the second column of `Xin`, and the column of `Xout`) are 0.5, 0.4, and 0.3 times the width of the data space, respectively. If `radii` is a scalar 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 `Xin` and `Xout` into a unit hyperbox, where N is the data (row) dimension. The first row of `xBounds` contains the minimum axis range values and the second row contains the maximum axis range values for scaling the data in each dimension.

For example, `xBounds = [-10 0 -1; 10 50 1]` specifies that data values in the first data dimension are to be scaled from the range `[-10 +10]` into values in the range `[0 1]`; data values in the second data dimension are to be scaled from the range `[0 50]`; and data values in the third data dimension are to be scaled from the range `[-1 +1]`. If `xBounds` is an empty matrix or not provided, then `xBounds` defaults to the minimum and maximum data values found in each data dimension.

- `options` is an optional vector for specifying algorithm parameters to override the default values. These parameters are explained in the help text for `subclust`. 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.

See Also

See Also

[anfis](#) | [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.

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')
```

See Also

See Also

[anfis](#) | [fcm](#) | [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 single 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) | positive integer | vector of positive integers

Number of input membership functions for each input variable, specified as the comma-separated pair consisting of 'NumMembershipFunctions' and one of the following:

- Positive integer — Specify the same number of membership functions for all inputs.

- Vector of positive integers 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
Character vector or string	Custom membership function in the current working folder or on the MATLAB path	“Build Mamdani Systems Using Custom Functions” on page 2-55

- 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 2-20.

'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

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

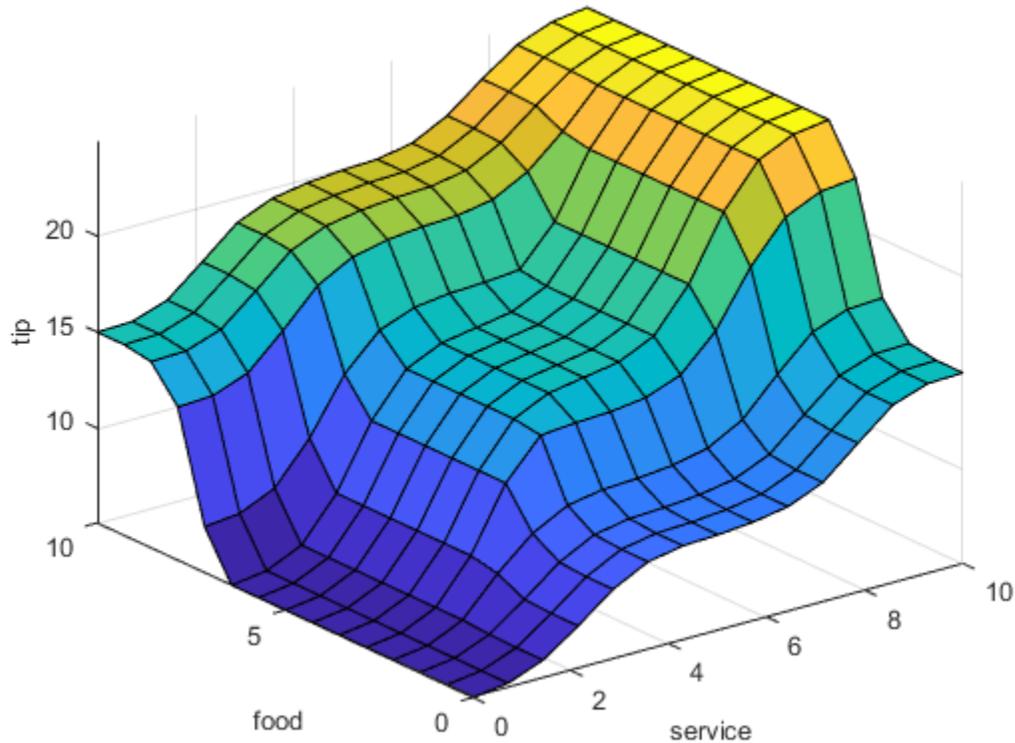
Load a fuzzy inference system.

```
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.

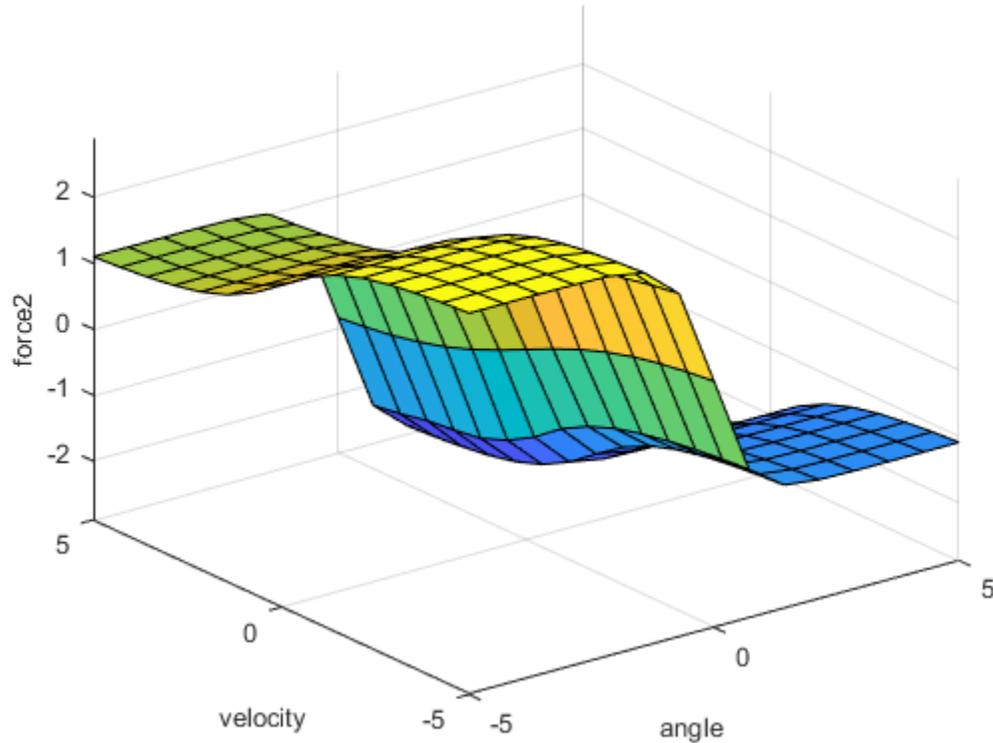
```
fis = readfis('mam22.fis');
```

Create an 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:

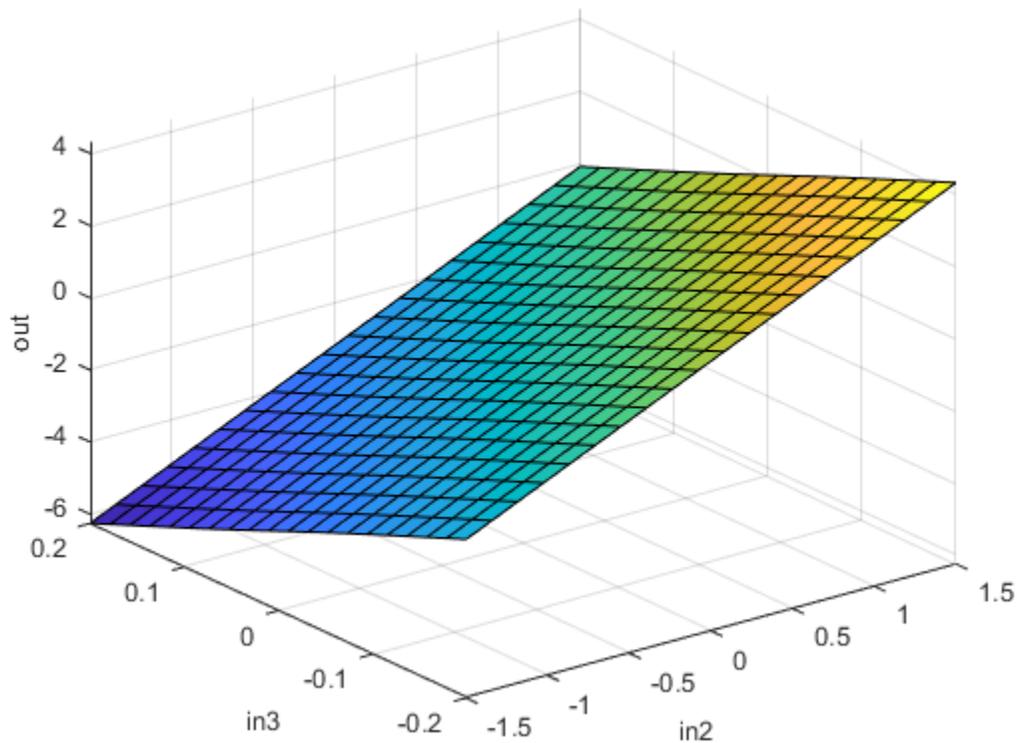
- Plot the output against the second and third input variable.
- Use 20 grid points for both inputs.

- 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

FIS structure

Fuzzy inference system, specified as an FIS structure.

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) = evalfis([X(i,j) Y(i,j)],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`.

See Also

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 single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, . . . , **NameN**, **ValueN**.

Example:

'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 100

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 100. For more information on membership function evaluation, see `evalfis`.

Output Arguments

opt — Option set for gensurf command

`gensurfOptions` option set

Option set for `gensurf` command, returned as a `gensurfOptions` option set.

See Also

See Also

evalfis | gensurf

Introduced in R2017a

getfis

Get fuzzy system properties

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 2-6.

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
<code>name</code>	FIS name, returned as a character vector.
<code>type</code>	FIS type, returned as a character vector.
<code>andMethod</code>	AND fuzzy operator method, returned as a character vector.
<code>orMethod</code>	OR fuzzy operator method, returned as a character vector.
<code>defuzzMethod</code>	Defuzzification method, returned as a character vector.
<code>impMethod</code>	Implication method, returned as a character vector.
<code>aggMethod</code>	Aggregation method, returned as a character vector.
<code>input</code>	Input variable information, returned as a structure or structure array. Each input variable structure contains the following fields: <ul style="list-style-type: none"> <code>name</code> — Variable name <code>range</code> — Variable range <code>mf</code> — Membership function names
<code>output</code>	Output variable information, returned as a structure or structure array. Each output variable structure contains the following fields: <ul style="list-style-type: none"> <code>name</code> — Variable name <code>range</code> — Variable range <code>mf</code> — Membership function names

Field	Description
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.
'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.
'numiutputs'	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
'defuzzmethod'	Defuzzification method for computing crisp output values, returned as one of the following for Mamdani systems:

fisProperty	fisInfo
	<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 2-6.

See Also

See Also

setfis | showfis

Introduced before R2006a

mam2sug

Transform Mamdani Fuzzy Inference System into Sugeno Fuzzy Inference System

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.

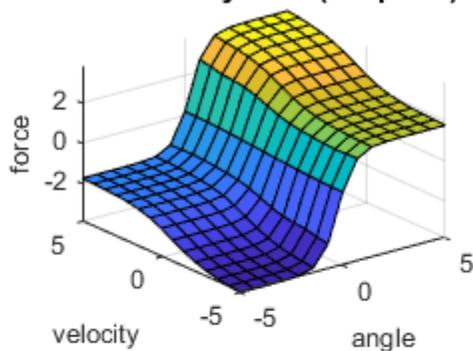
```
sug_fismat = mam2sug(mam_fismat);
```

Plot the output surfaces for both fuzzy systems.

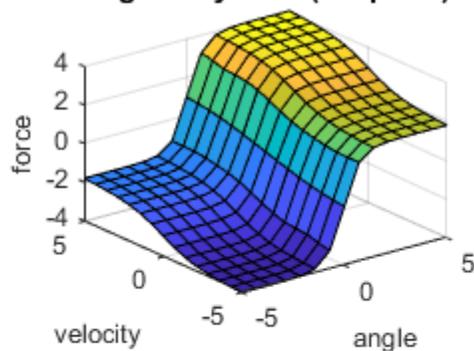
```
subplot(2,2,1)
gensurf(mam_fismat)
title('Mamdani system (Output 1)')
subplot(2,2,2)
gensurf(sug_fismat)
title('Sugeno system (Output 1)')
subplot(2,2,3)
gensurf(mam_fismat,gensurfOptions('OutputIndex',2))
title('Mamdani system (Output 2)')
subplot(2,2,4)
```

```
gensurf(sug_fismat,gensurfOptions('OutputIndex',2))
title('Sugeno system (Output 2)')
```

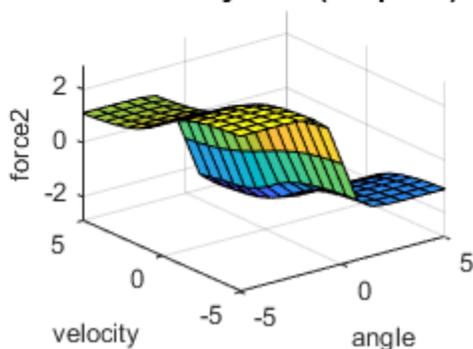
Mamdani system (Output 1)



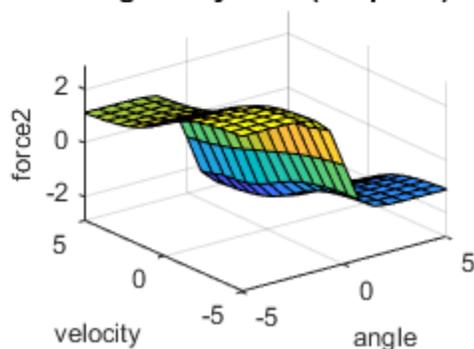
Sugeno system (Output 1)



Mamdani system (Output 2)



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. See “Build Mamdani Systems at the

Command Line” on page 2-69 and “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31 for more information.

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.

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**.

See Also

See Also

Fuzzy Logic Designer

Topics

“What Is Mamdani-Type Fuzzy Inference?” on page 2-30

“What Is Sugeno-Type Fuzzy Inference?” on page 2-92

“Build Mamdani Systems at the Command Line” on page 2-69

“Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31

Introduced before R2006a

mf2mf

Translate parameters between membership functions

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:

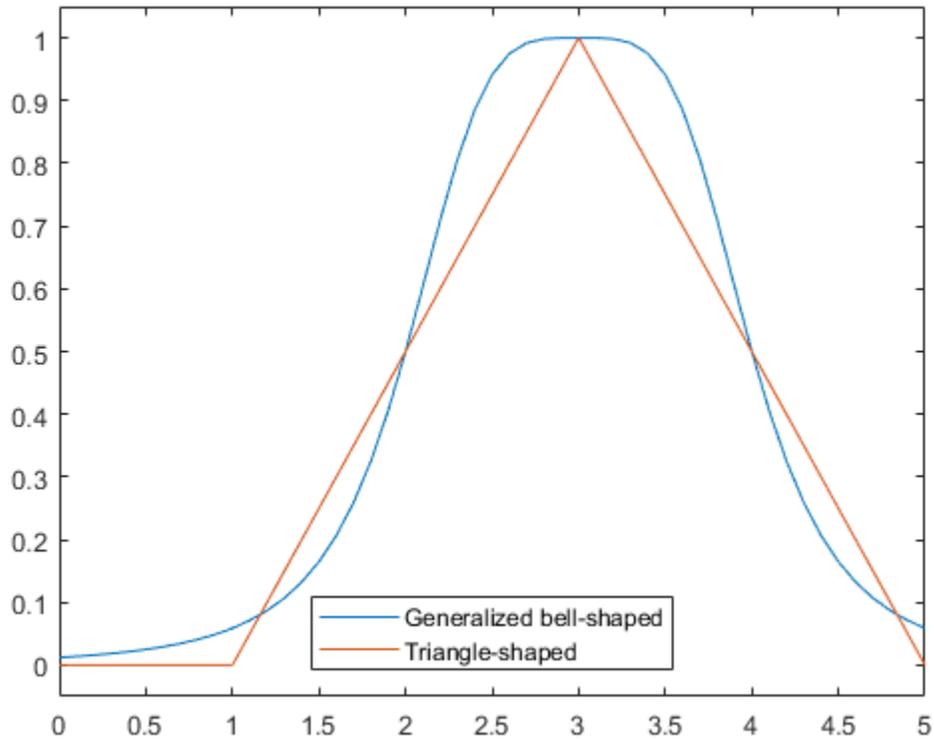
Membership function type	Description	For more information
'gbellmf'	Generalized bell-shaped membership function	gbellmf
'gaussmf'	Gaussian membership function	gaussmf

Membership function type	Description	For more information
'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])
```



See Also

See Also

`dsigmf` | `evalmf` | `gauss2mf` | `gaussmf` | `gbellmf` | `pimf` | `psigmf` | `sigmf` | `smf`
| `trapmf` | `trapmf` | `trimf` | `trimf` | `zmf`

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

Introduced before R2006a

mfedit

Open Membership Function Editor

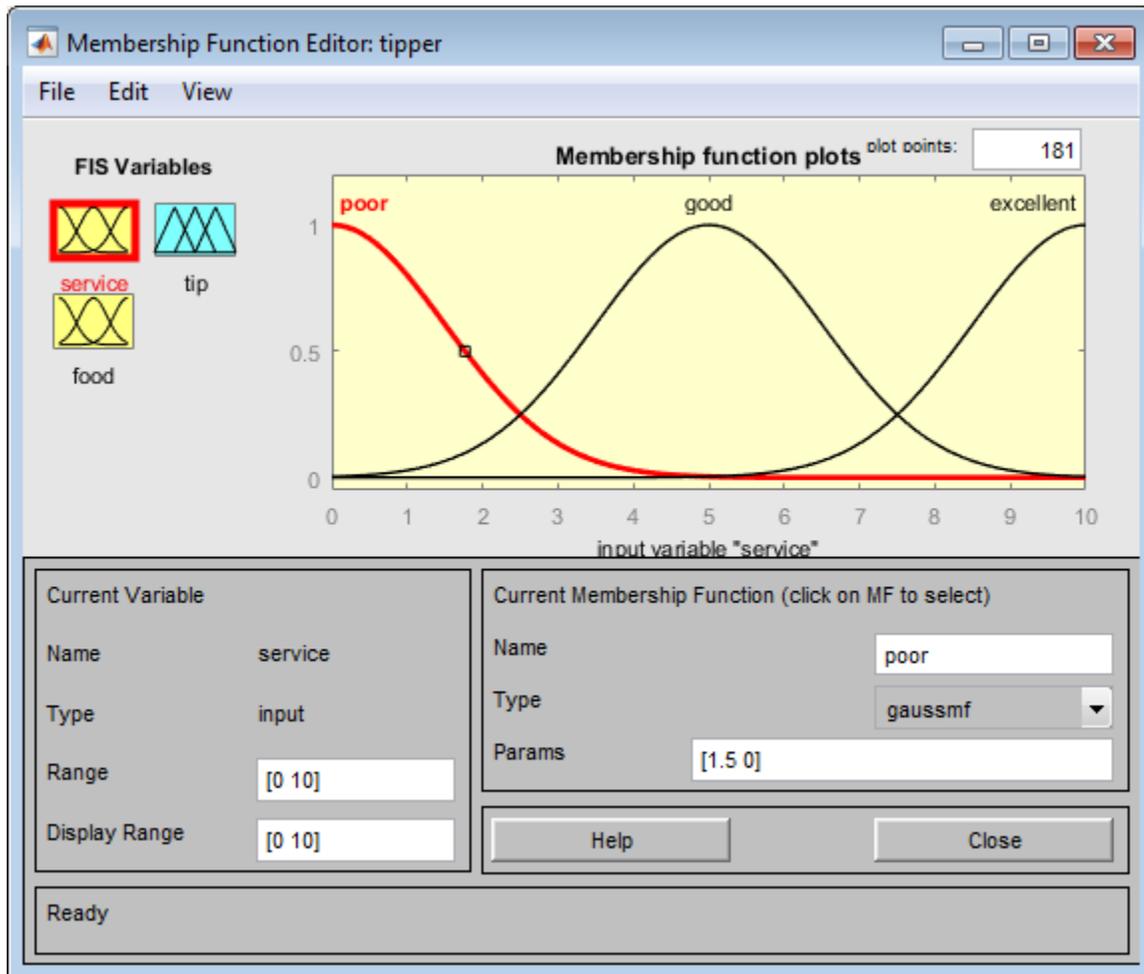
Syntax

```
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 MATLAB workspace variable, for an FIS structure, `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 2-39 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.

See Also

See Also

Apps

Fuzzy Logic Designer

Functions

addmf | plotmf | ruleedit | ruleview | surfview

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

Introduced before R2006a

newfis

Create new fuzzy inference system

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 single 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 2-20.

'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 Mamdani Systems Using Custom Functions” on page 2-55.

For more information on fuzzy operators and the fuzzy inference process, see “Fuzzy Inference Process” on page 2-22.

'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 Mamdani Systems Using Custom Functions” on page 2-55.

For more information on fuzzy operators and the fuzzy inference process, see “Fuzzy Inference Process” on page 2-22.

'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 Mamdani Systems Using Custom Functions” on page 2-55.

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 2-22.

'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 Mamdani Systems Using Custom Functions” on page 2-55.

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 2-22.

'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 Mamdani Systems Using Custom Functions” on page 2-55.

For more information on defuzzification and the fuzzy inference process, see “Fuzzy Inference Process” on page 2-22.

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.

See Also

See Also

`readfis` | `writefis`

Topics

“Foundations of Fuzzy Logic” on page 2-2

“Fuzzy Inference Process” on page 2-22

Introduced before R2006a

parsrule

Parse fuzzy rules

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','français');
```

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 single 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)	==

Rule Component	Symbol
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

Rule Component	English	French	German
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`.

See Also

See Also

`addrule` | `ruleedit` | `showrule`

Introduced before R2006a

pimf

Π-shaped membership function

Syntax

```
y = 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 \mathbf{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:

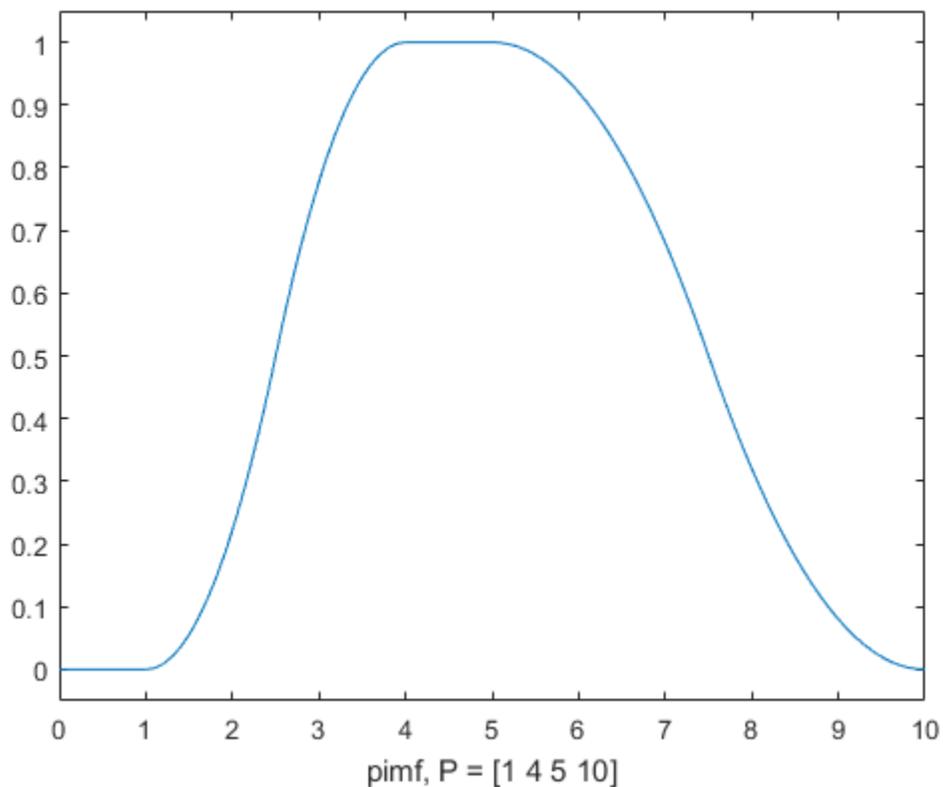
$$f(x;a,b,c,d) = \begin{cases} 0, & x \leq a \\ 2\left(\frac{x-a}{b-a}\right)^2, & a \leq x \leq \frac{a+b}{2} \\ 1 - 2\left(\frac{x-b}{b-a}\right)^2, & \frac{a+b}{2} \leq x \leq b \\ 1, & b \leq x \leq c \\ 1 - 2\left(\frac{x-c}{d-c}\right)^2, & c \leq x \leq \frac{c+d}{2} \\ 2\left(\frac{x-d}{d-c}\right)^2, & \frac{c+d}{2} \leq x \leq d \\ 0, & x \geq d \end{cases}$$

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])
```



See Also

See Also

dsigmf | evalmf | gauss2mf | gaussmf | gbellmf | mf2mf | psigmf | sigmf | smf
| trapmf | trapmf | trimf | trimf | zmf

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

Introduced before R2006a

plotfis

Plot Fuzzy Inference System

Syntax

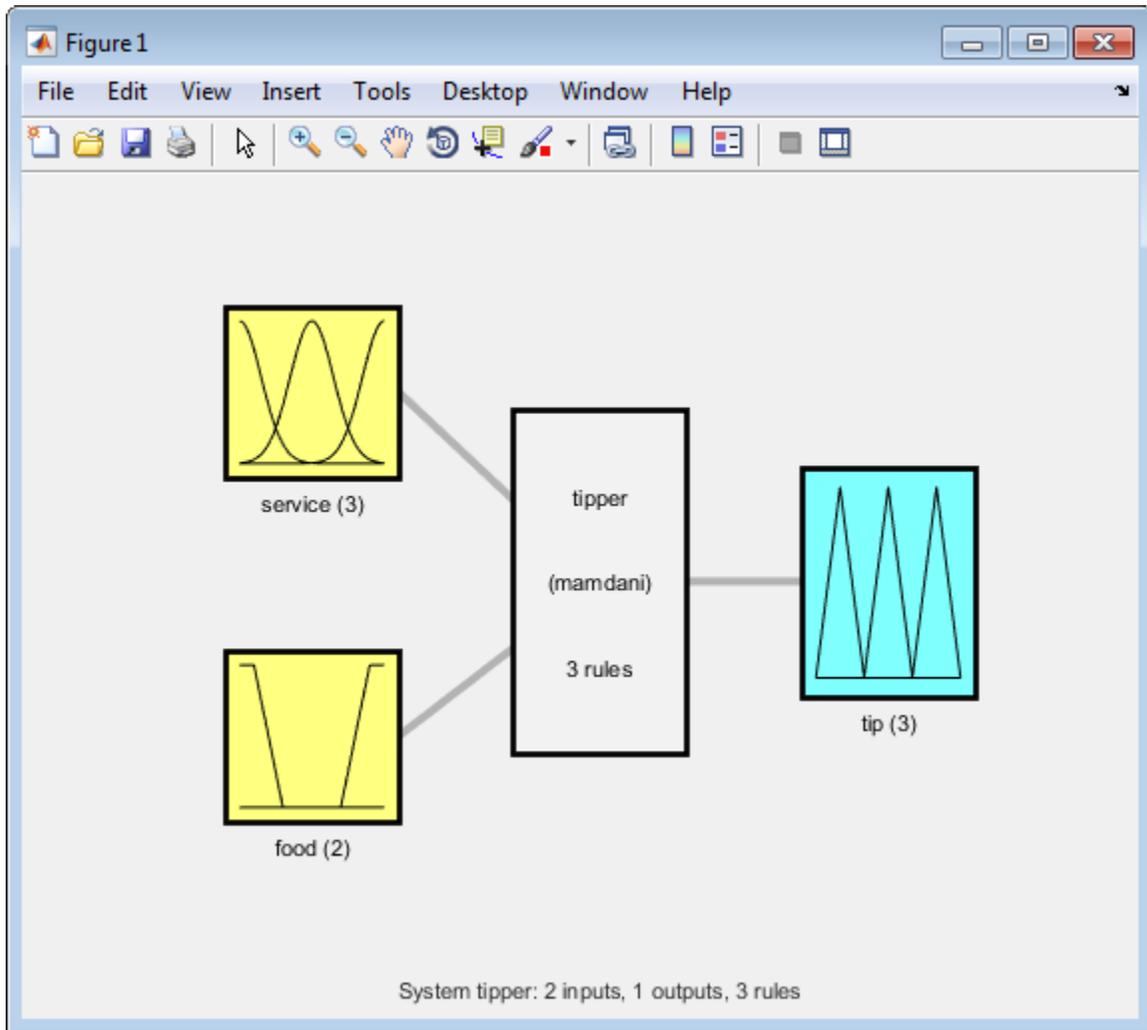
```
plotfis(fismat)
```

Description

This function displays a high level diagram of an FIS, `fismat`. Inputs and their membership functions appear to the left of the FIS structural characteristics, while outputs and their membership functions appear on the right.

Examples

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



See Also

See Also

`evalmf` | `plotmf`

Introduced before R2006a

plotmf

Plot all membership functions for given variable

Syntax

```
plotmf(fis,varType,varIndex)
```

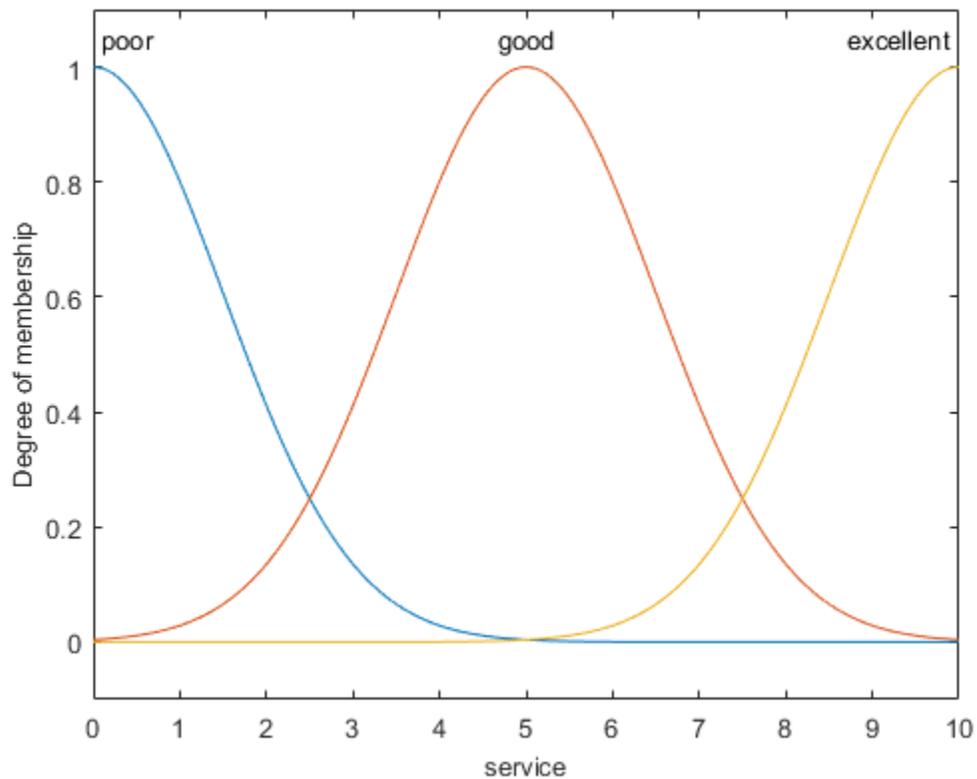
Description

This function plots all of the membership functions in the FIS called `fis` associated with a given variable whose type and index are respectively given by `varType` (must be 'input' or 'output'), and `varIndex`. This function can also be used with the MATLAB function, `subplot`.

Examples

Plot Membership Functions for FIS Input Variable

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



See Also

See Also

`evalmf` | `plotfis`

Introduced before R2006a

probor

Probabilistic OR

Syntax

```
y = probor(x)
```

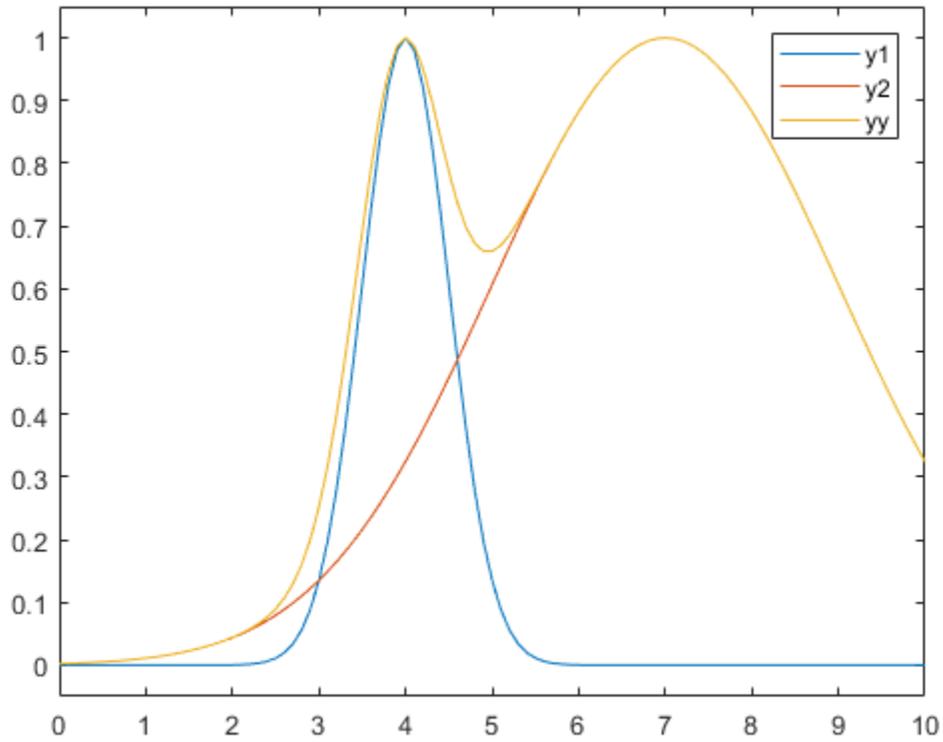
Description

`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

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

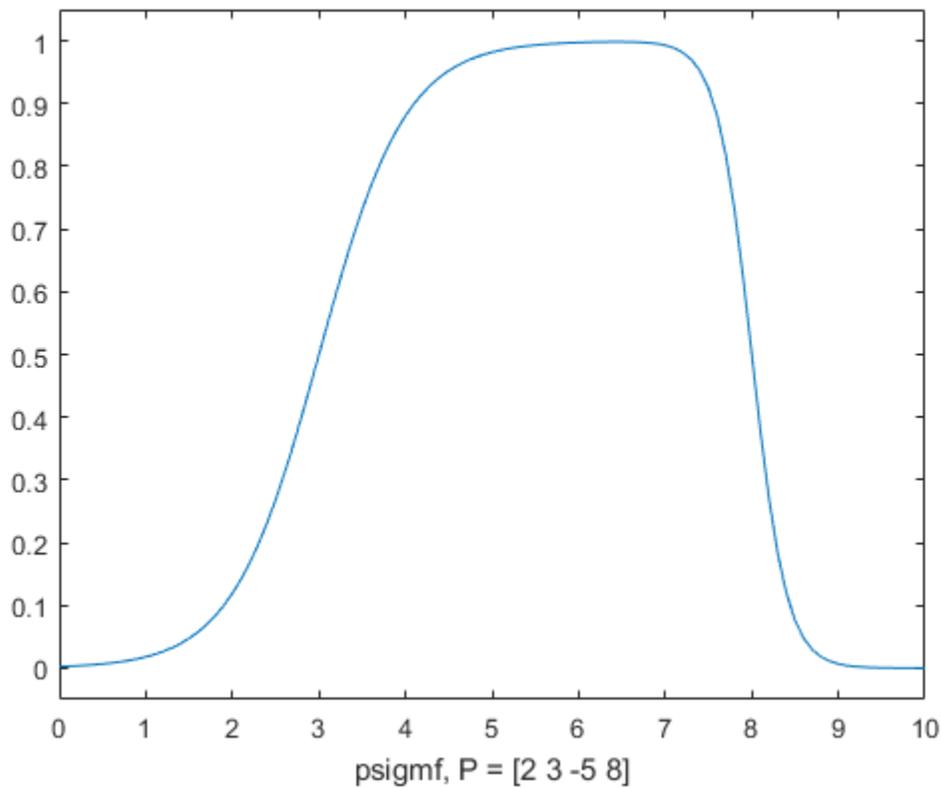
`psigmf` is simply the product of two such curves plotted for the values of the vector x
 $f_1(x; a_1, c_1) \times f_2(x; a_2, c_2)$

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])
```



See Also

See Also

`dsigmf` | `evalmf` | `gauss2mf` | `gaussmf` | `gbellmf` | `mf2mf` | `pimf` | `sigmf` | `smf` | `trapmf` | `trapmf` | `trimf` | `trimf` | `zmf`

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

Introduced before R2006a

readfis

Load Fuzzy Inference System from file

Syntax

```
fis = readfis(fileName)
```

```
fis = readfis
```

Description

`fis = readfis(fileName)` reads a fuzzy inference system from a file. Specify `fileName` as a character vector or string with or without the `.fis` extension.

`fis = readfis` opens a dialog box for retrieving files to select the fuzzy inference system file.

Examples

Load Fuzzy Inference System From File

Load the fuzzy system stored in the file `tipper.fis`.

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

View the properties of the FIS.

```
getfis(fis)
```

```
Name      = tipper
Type      = mamdani
NumInputs = 2
InLabels  =
    service
    food
NumOutputs = 1
OutLabels =
```

```
tip
NumRules = 3
AndMethod = min
OrMethod = max
ImpMethod = min
AggMethod = max
DefuzzMethod = centroid
```

See Also

See Also

writefis

Introduced before R2006a

rmmf

Remove membership function from fuzzy inference system

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.

See Also

See Also

`addmf` | `addrule` | `addvar` | `plotmf` | `rmvar`

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

Introduced before R2006a

rmvar

Remove variables from fuzzy inference system

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.

See Also

See Also

[addmf](#) | [addrule](#) | [addvar](#) | [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 2-47 for more information about how to use `ruleedit`.

`ruleedit(fis)` operates on a MATLAB workspace variable, for an FIS structure, `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 " $A \& B \Rightarrow C$."

Indexed mirrors how the rule is stored in the FIS structure.

See Also

See Also

Apps

Fuzzy Logic Designer

Functions

addrule | mfedit | parsrule | ruleview | showrule | surfview

Topics

"The Rule Editor" on page 2-47

Introduced before R2006a

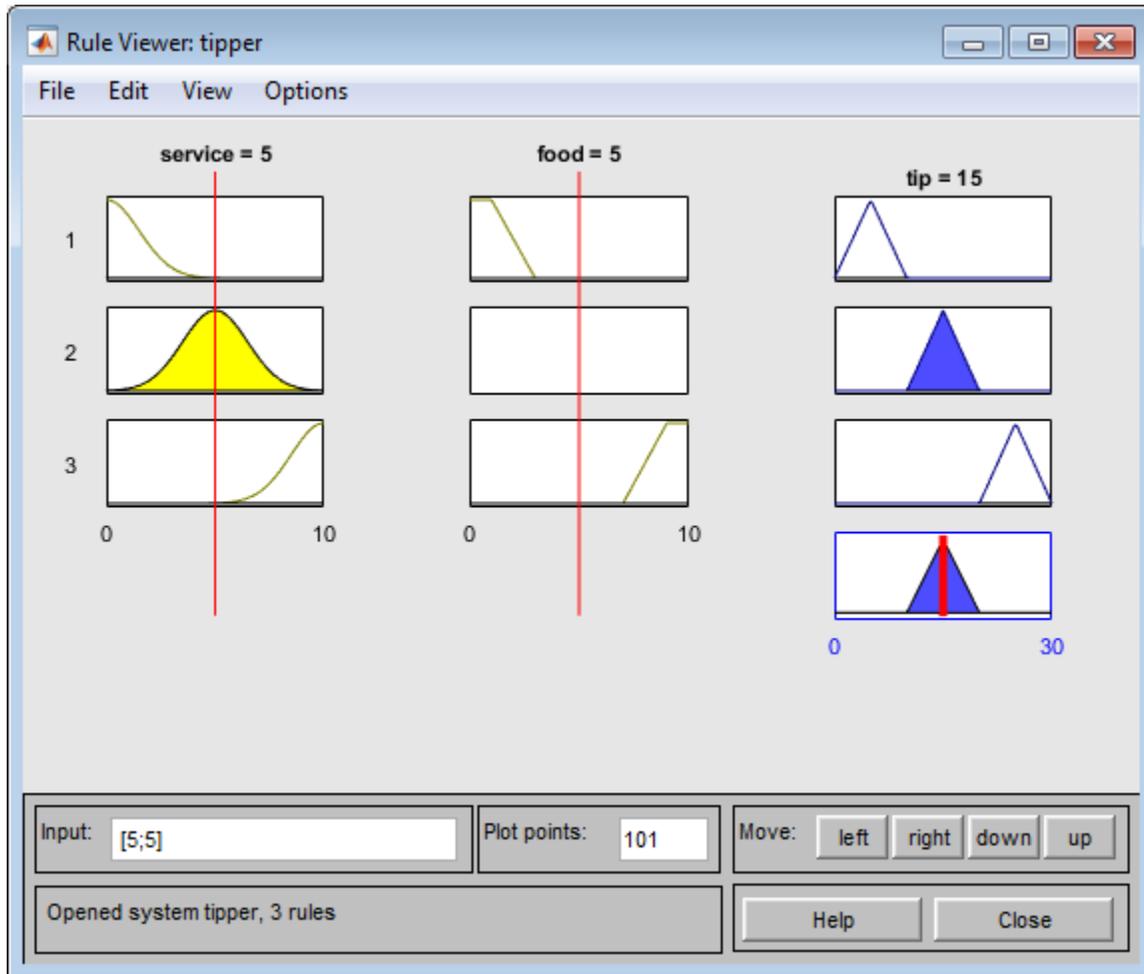
ruleview

Open Rule Viewer

Syntax

```
ruleview(fis)  
ruleview(fileName)
```

Description



`ruleview(fis)` opens the Rule Viewer for the fuzzy inference system, `fis`. It is used to view the entire implication process from beginning to end. You can move around the line indices that correspond to the inputs and then watch the system readjust and compute the new output. Refer to “The Rule Viewer” on page 2-50 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.

See Also

See Also

Apps

Fuzzy Logic Designer

Functions

`addrule` | `mfedit` | `ruleedit` | `showrule` | `surfview`

Topics

“The Rule Viewer” on page 2-50

Introduced before R2006a

setfis

Set fuzzy system properties

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]

See Also

See Also

getfis

Introduced before R2006a

showfis

Display annotated Fuzzy Inference System

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	mandani
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

```

See Also

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 =
```

```
3×78 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 =
```

```
3×65 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 =
```

```
3×15 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 =
```

```
2×78 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 =
```

```
3×85 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

FIS structure

Fuzzy inference system, specified as an FIS structure.

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 single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example:

'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 `0`.

'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

See Also

See Also

addrule | parsrule | 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 .

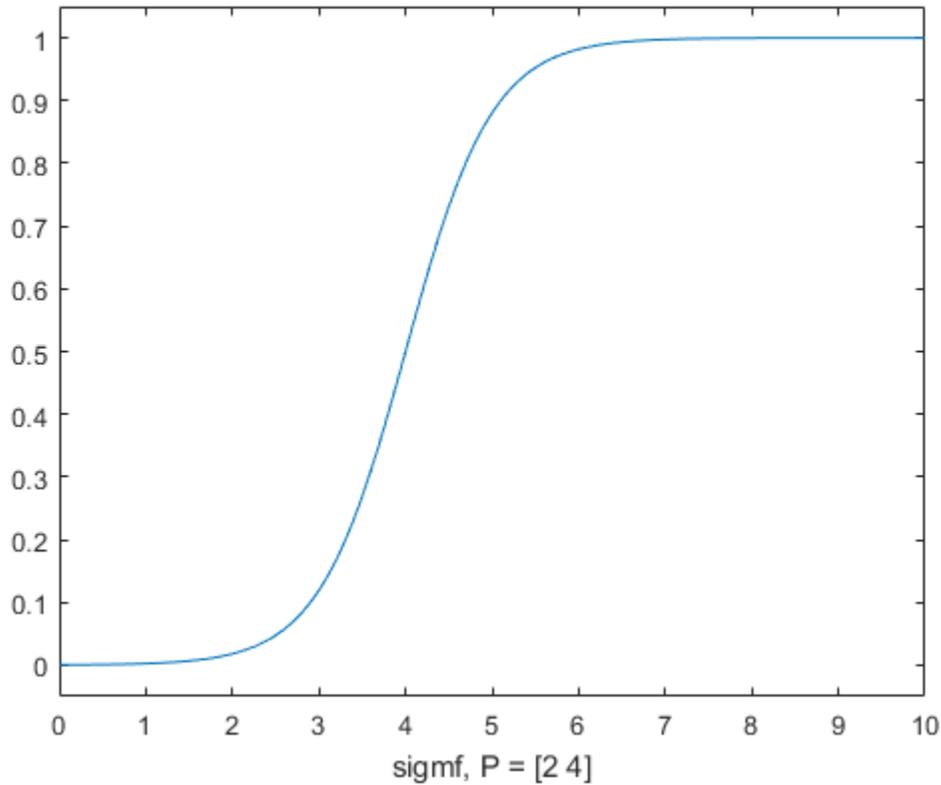
$$f(x,a,c) = \frac{1}{1 + e^{-a(x-c)}}$$

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 `dsigmoid` and `psigmoid`.

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])
```



See Also

See Also

[dsigmf](#) | [evalmf](#) | [gauss2mf](#) | [gaussmf](#) | [gbellmf](#) | [mf2mf](#) | [pimf](#) | [psigmf](#) | [smf](#)
| [trapmf](#) | [trapmf](#) | [trimf](#) | [trimf](#) | [zmf](#)

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

Introduced before R2006a

smf

S-shaped membership function

Syntax

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

Description

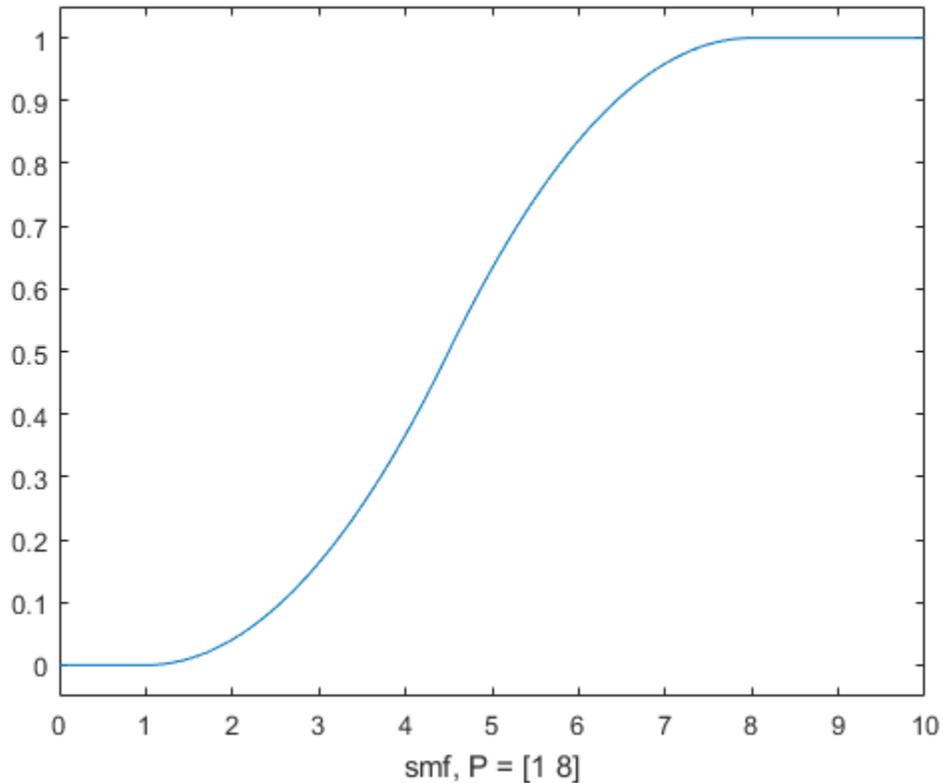
This spline-based curve is a mapping on the vector x , and is named because of its S-shape. The parameters a and b locate the extremes of the sloped portion of the curve, as given by:

$$f(x;a,b) = \begin{cases} 0, & x \leq a \\ 2\left(\frac{x-a}{b-a}\right)^2, & a \leq x \leq \frac{a+b}{2} \\ 1-2\left(\frac{x-b}{b-a}\right)^2, & \frac{a+b}{2} \leq x \leq b \\ 1, & x \geq b \end{cases}$$

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])
```



See Also

See Also

`dsigmf` | `evalmf` | `gauss2mf` | `gaussmf` | `gbellmf` | `mf2mf` | `pimf` | `psigmf` | `sigmf` | `trapmf` | `trapmf` | `trimf` | `trimf` | `zmf`

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

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 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 =  
    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 single

quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

Example:

'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);
```

Extract the input and output variable data.

```
inputData = data(:,1:M);  
outputData = data(:,M+1:end);
```

- 2 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. 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

See Also

genfis

Topics

"Fuzzy Clustering" on page 4-2

"Model Suburban Commuting Using Subtractive Clustering" on page 4-12

Introduced before R2006a

surfview

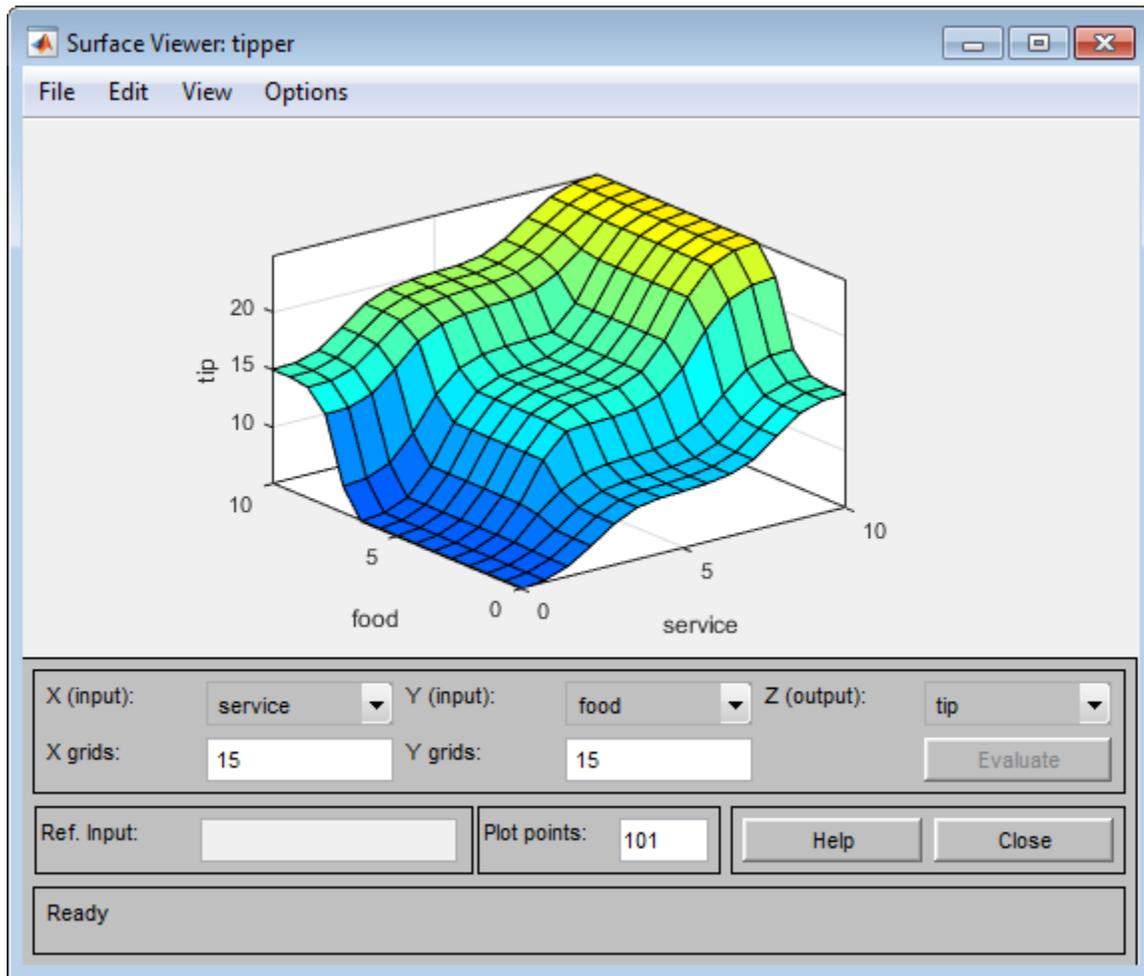
Open Surface Viewer

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 an FIS structure.

Because it does not alter the fuzzy system or its associated FIS structure 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.

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**.

Click **Evaluate** to perform the calculation and plot the output surface.

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

If there are more than two inputs to your system, you must supply the constant values associated with any unspecified inputs in the reference input section.

Refer to “The Surface Viewer” on page 2-52 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.

See Also

See Also

Apps

Fuzzy Logic Designer

Functions

gensurf | mfedit | ruleedit | ruleview

Topics

“The Surface Viewer” on page 2-52

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

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

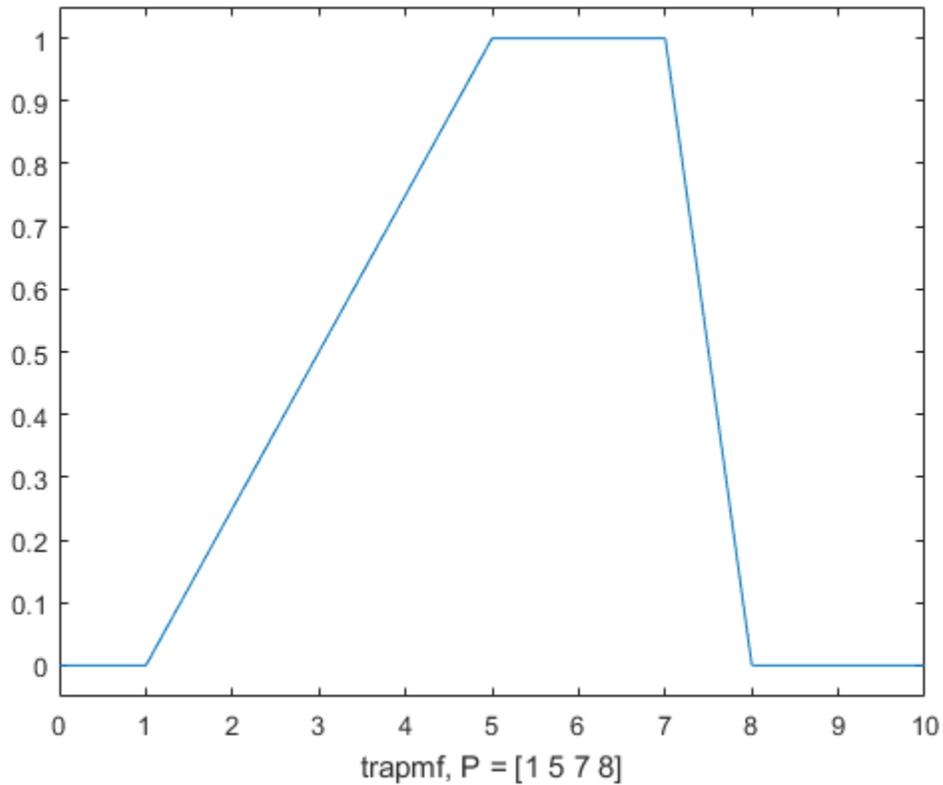
The parameters a and d locate the “feet” of the trapezoid and the parameters b and c locate the “shoulders.”

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])
```



See Also

See Also

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

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

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

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

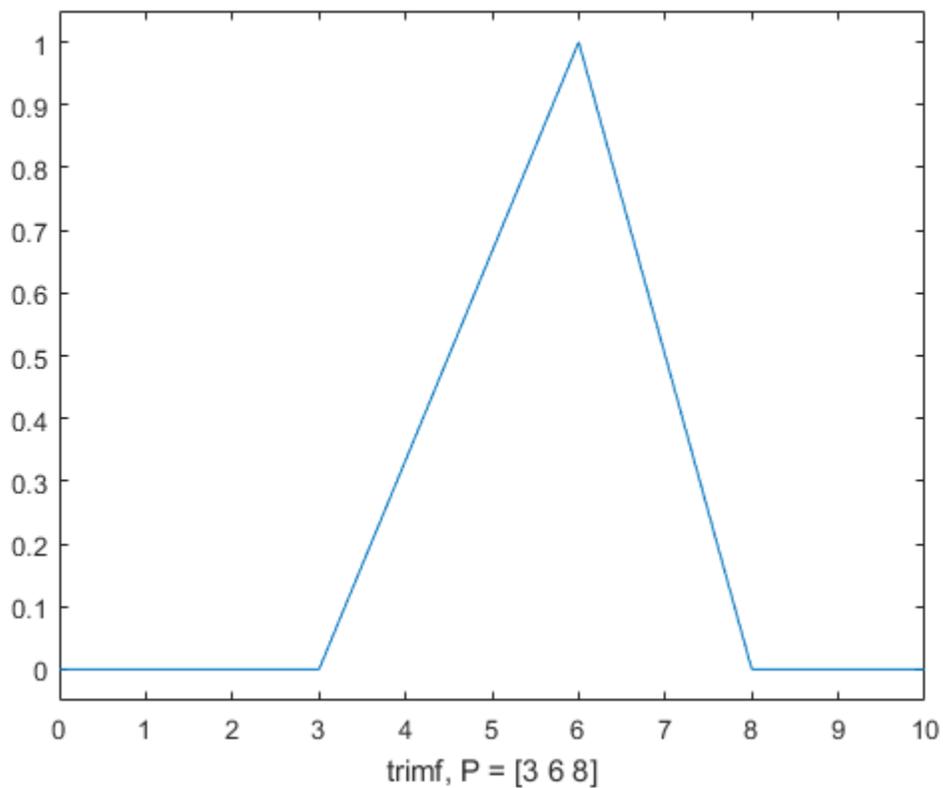
The parameters a and c locate the “feet” of the triangle and the parameter b locates the peak.

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])
```



See Also

See Also

dsigmf | evalmf | gauss2mf | gaussmf | gbellmf | mf2mf | pimf | psigmf |
sigmf | smf | trapmf | trapmf | trimf | zmf

Topics

“Membership Functions” on page 2-6

“The Membership Function Editor” on page 2-39

Introduced before R2006a

writefis

Save fuzzy inference system to file

Syntax

```
writefis(fis)
writefis(fis,fileName)
writefis(fis,fileName,'dialog')
```

Description

`writefis` saves a MATLAB workspace FIS structure, `fis`, as a `.fis` file.

`writefis(fis)` opens a dialog box to assist with the naming and folder location of the file.

`writefis(fis,fileName)` writes a `.fis` file corresponding to the FIS structure, `fis`, to a file called `filename.fis`. Specify `fileName` as a character vector or string. No dialog box appears, and the file is saved to the current folder.

`writefis(fis,fileName,'dialog')` opens a dialog box with the default name `filename.fis` supplied.

The extension `.fis` is only added to `fileName` if it is not already included in the name.

Examples

Save Fuzzy Inference System to File

Create a fuzzy inference system, and add an input variable with membership functions.

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

Save the fuzzy system in the current working folder in the file `myFile.fis`.

```
writefis(a, 'myFile');
```

See Also

See Also

`newfis` | `readfis`

Introduced before R2006a

zmf

Z-shaped membership function

Syntax

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

Description

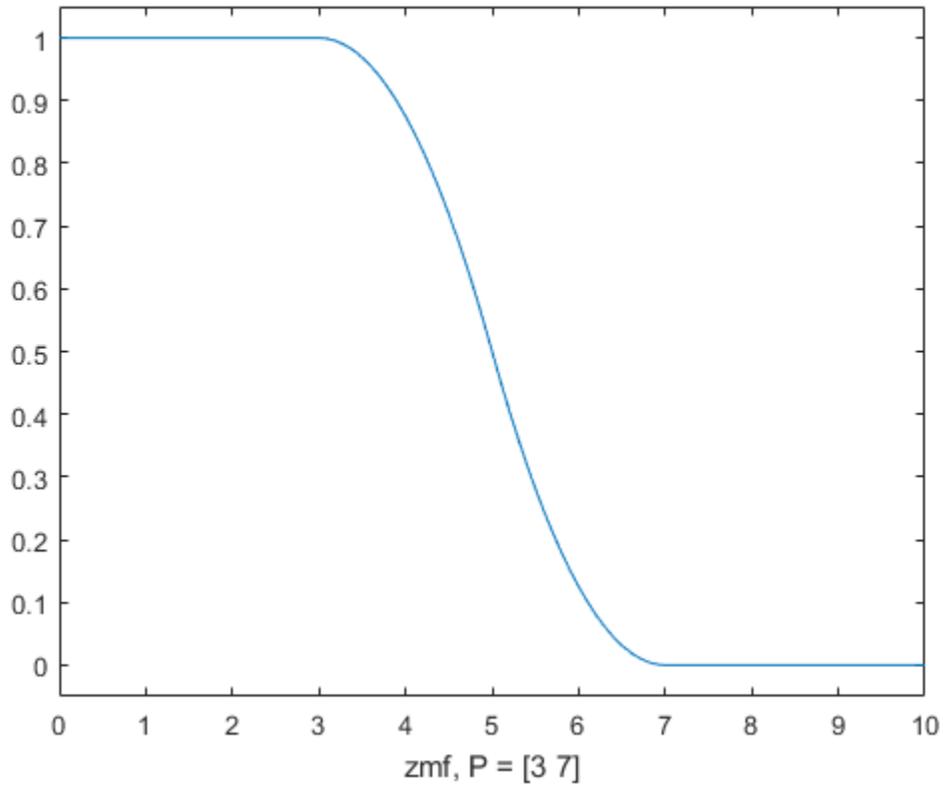
This spline-based function of x is so named because of its Z-shape. The parameters a and b locate the extremes of the sloped portion of the curve as given by.

$$f(x;a,b) = \begin{cases} 1, & x \leq a \\ 1 - 2\left(\frac{x-a}{b-a}\right)^2, & a \leq x \leq \frac{a+b}{2} \\ 2\left(\frac{x-b}{b-a}\right)^2, & \frac{a+b}{2} \leq x \leq b \\ 0, & x \geq b \end{cases}$$

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])
```



See Also

See Also

`dsigmf` | `evalmf` | `gauss2mf` | `gaussmf` | `gbellmf` | `mf2mf` | `pimf` | `psigmf` | `sigmf` | `smf` | `trapmf` | `trapmf` | `trimf` | `trimf`

Topics

“Membership Functions” on page 2-6

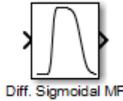
“The Membership Function Editor” on page 2-39

Introduced before R2006a

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

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 2-22.

To display the fuzzy inference process in the Rule Viewer during simulation, use the Fuzzy Logic Controller with Ruleviewer block.

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 name — Fuzzy inference system

structure | file name

Fuzzy inference system to evaluate, specified as one of the following:

- Structure — Specify the name of an FIS structure in the MATLAB workspace.

To create a fuzzy inference system, you can:

- Use the **Fuzzy Logic Designer** app. For an example, see “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31.
- Automatically generate the FIS structure using the `genfis` command.
- Build the system at the command line. For an example, see “Build Mamdani Systems at the Command Line” on page 2-69.
- File name — Specify the name of a `.fis` file in the current working folder or on the MATLAB path. You must include the file extension in the file name.

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`.

Model Examples

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks

Fuzzy Logic Controller with Ruleviewer

Apps

Fuzzy Logic Designer | Neuro-Fuzzy Designer

Functions

evalfis | newfis | readfis

Topics

“Fuzzy Inference Process” on page 2-22

“Simulate Fuzzy Inference Systems in Simulink” on page 2-81

“Build Your Own Fuzzy Simulink Models” on page 2-88

Introduced before R2006a

Fuzzy Logic Controller with Ruleviewer

Evaluate fuzzy inference system and view rules

Library: Fuzzy Logic Toolbox



Fuzzy Logic Controller with Ruleviewer

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 2-22.

To evaluate a fuzzy inference system without displaying the Rule Viewer, use the Fuzzy Logic Controller block.

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

structure

Fuzzy inference system to evaluate, specified as an FIS structure in the MATLAB workspace.

To create a fuzzy inference system, you can:

- Use the **Fuzzy Logic Designer** app. For an example, see “Build Mamdani Systems Using Fuzzy Logic Designer” on page 2-31.
- Automatically generate the FIS structure using the `genfis` command.
- Build the system at the command line. For an example, see “Build Mamdani Systems at the Command Line” on page 2-69.

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.

Model Examples

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

See Also

Blocks

Fuzzy Logic Controller

Apps

Fuzzy Logic Designer | Neuro-Fuzzy Designer

Functions

evalfis | newfis | readfis

Topics

“Fuzzy Inference Process” on page 2-22

“Simulate Fuzzy Inference Systems in Simulink” on page 2-81

“Build Your Own Fuzzy Simulink Models” on page 2-88

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

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

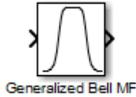
See Also

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

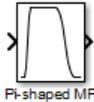
See Also

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

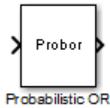
See Also

`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

See Also

Blocks

Probabilistic Rule Agg

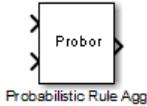
Functions

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

See Also

Blocks

Probabilistic OR

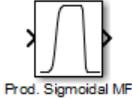
Functions

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

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

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

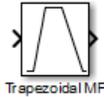
See Also

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

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

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

See Also

zmf

Introduced before R2006a

Bibliography

- [1] Bezdec, 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.

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.