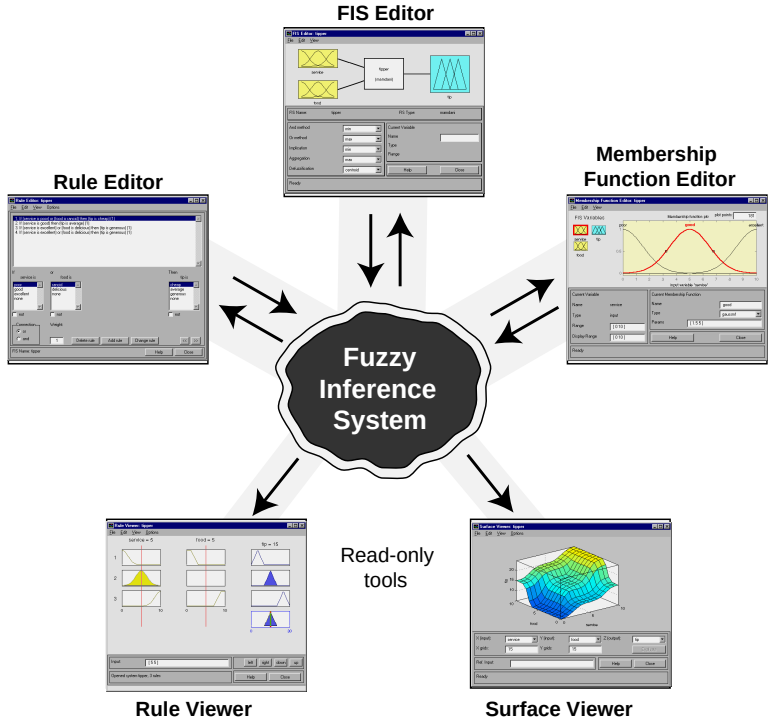
Building Systems with the Fuzzy Logic Toolbox

Dinner for Two, from the Top

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

In addition to these five primary GUIs, the toolbox includes the graphical ANFIS Editor GUI, which is used for building and analyzing Sugeno-type adaptive neural fuzzy inference systems. The ANFIS Editor GUI is discussed later in the section, "Sugeno-Type Fuzzy Inference" on page 2-77.



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

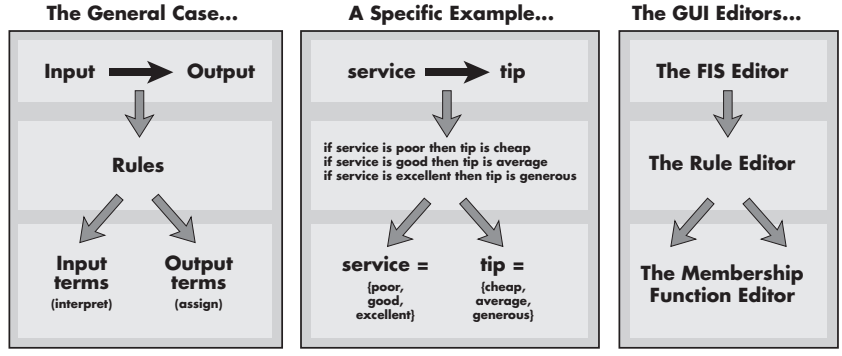
The Membership Function Editor is used to define the shapes of all the membership functions associated with each variable.

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

The Rule Viewer and the Surface Viewer are used for looking at, as opposed to editing, the FIS. They are strictly read-only tools. The Rule Viewer is a MATLAB based display of the fuzzy inference diagram shown at the end of

the last section. Used as a diagnostic, it can show (for example) which rules are active, or how individual membership function shapes are influencing the results. The Surface Viewer is used to display the dependency of one of the outputs on any one or two of the inputs — that is, it generates and plots an output surface map for the system.

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



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

Getting Started

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

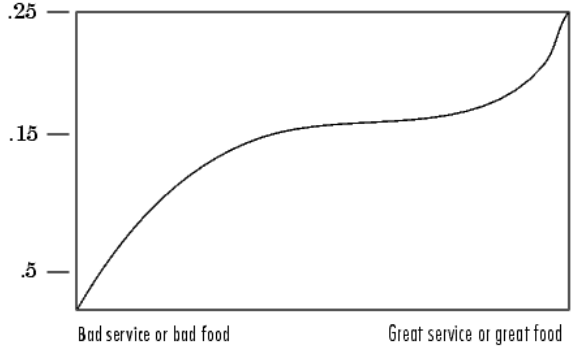
The Basic Tipping Problem

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

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

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

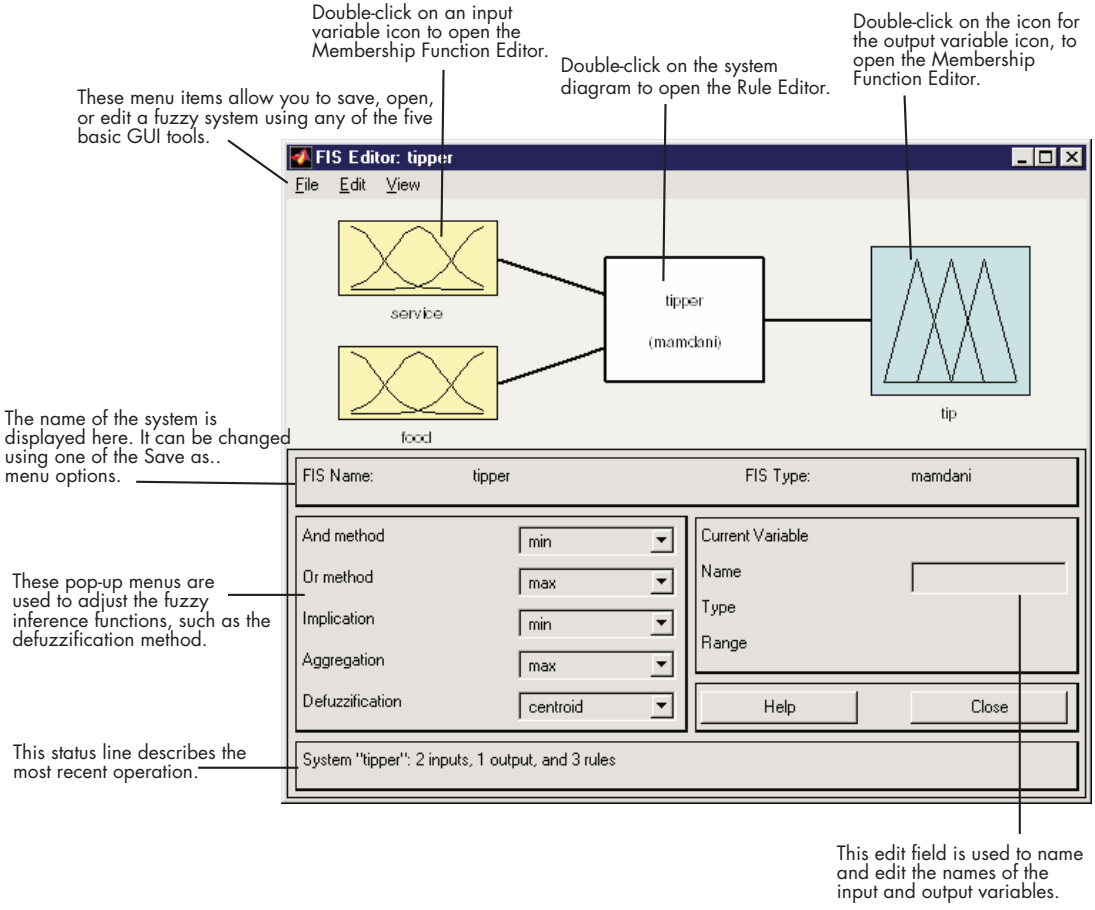
We'll assume that an average tip is 15%, a generous tip is 25%, and a cheap tip is 5%. it is also useful to have a vague idea of what the tipping function should look like this.



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

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

The FIS Editor



The following discussion tells you how to build a new fuzzy inference system from scratch. If you want to save time and follow along quickly, you can load the pre-built system by typing

```
fuzzy tipper
```

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

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

Below the diagram is the name of the system and the type of inference used. The default, Mamdani-type inference, is what we've been describing so far and what we'll continue to use for this example. Another slightly different type of inference, called Sugeno-type inference, is also available. This method is explained in "Sugeno-Type Fuzzy Inference" on page 2-77. Below the name of the fuzzy inference system, on the left side of the figure, are the pop-up menus that allow you to modify the various pieces of the inference process. On the right side at the bottom of the figure is the area that displays the name of either an input or output variable, its associated membership function type, and its range. The latter two fields are specified only after the membership functions have been. Below that region are the **Help** and **Close** buttons that call up online help and close the window, respectively. At the bottom is a status line that relays information about the system.

To start this system from scratch, type

```
fuzzy
```

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

1. Click once on the box (yellow) on the left marked **input1** (the box will be highlighted in red).
2. In the white edit field on the right, change `input1` to `service` and press **Return.**

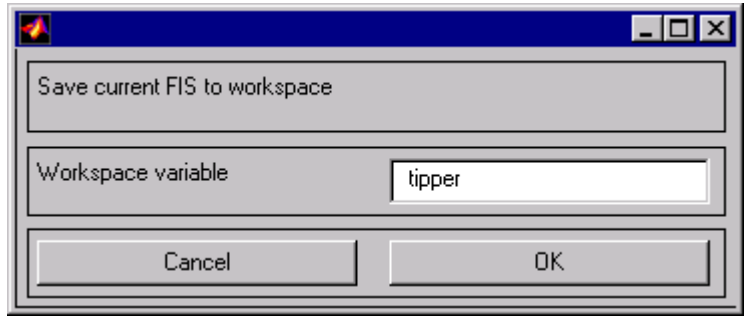
3 Click once on the box (yellow) marked **input2** (the box will be highlighted in red).

4 In the white edit field on the right, change `input2` to `food` and press **Return**.

5 Click once on the box (blue) on the right marked **output1**.

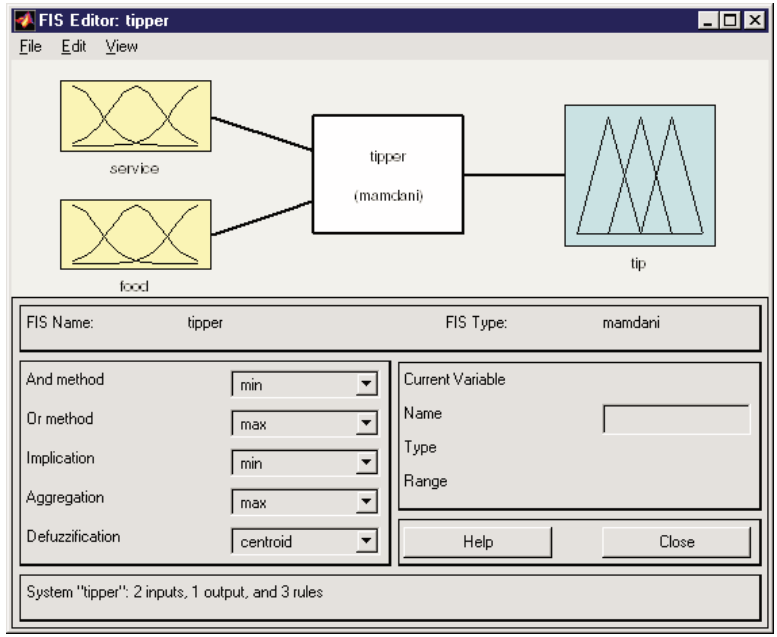
6 In the white edit field on the right, change `output1` to `tip`.

7 From the **File** menu, select **Export and then To Workspace...**



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

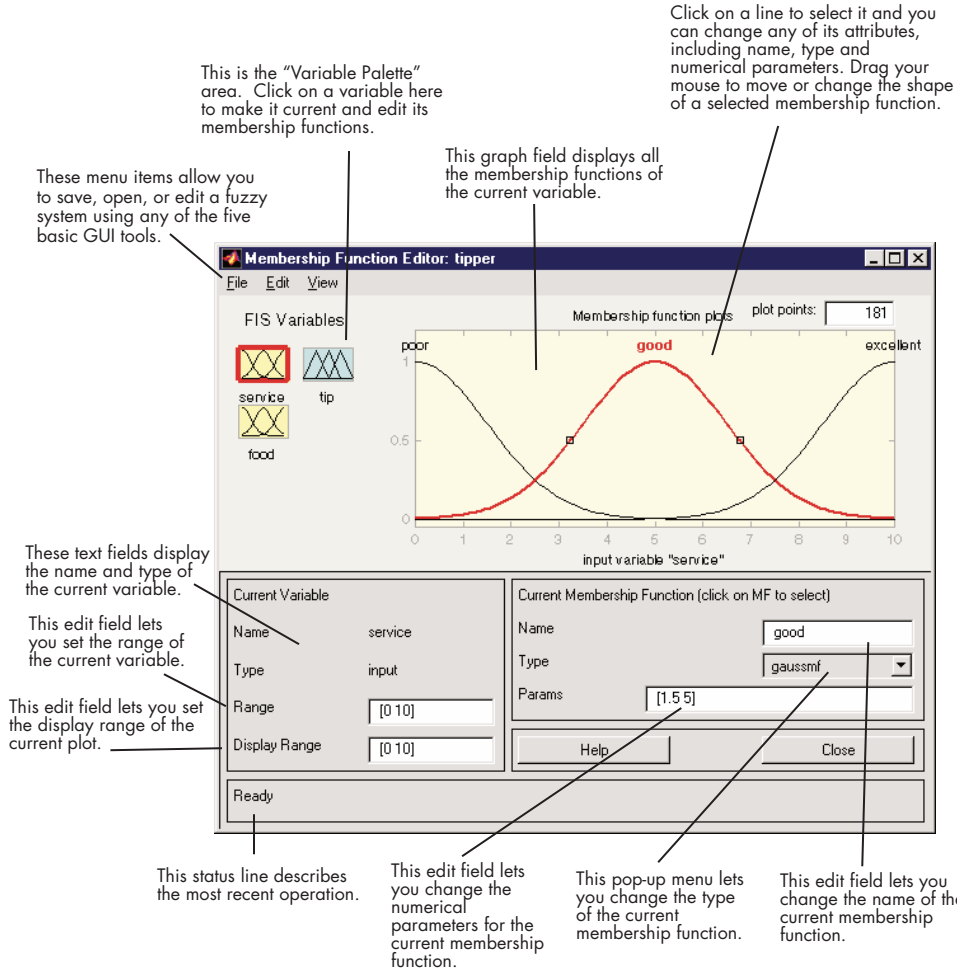
You will see the diagram updated to reflect the new names of the input and output variables. There is now a new variable in the workspace called `tipper` that contains all the information about this system. By saving to the workspace with a new name, you also rename the entire system. Your window will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



Leave the inference options in the lower left in their default positions for now. You've entered all the information you need for this particular GUI. Next define the membership functions associated with each of the variables. To do this, open the Membership Function Editor. You can open the Membership Function Editor in one of three ways:

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

The Membership Function Editor



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

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

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

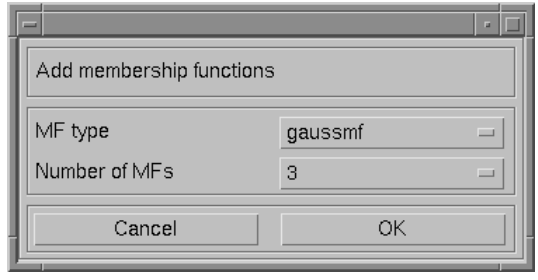
Next select the **Edit** pull-down menu, and choose **Add MFS ...**. A new window will appear, which allows you to select both the membership function type and the number of membership functions associated with the selected variable. In the lower right corner of the window are the controls that let you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

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

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

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

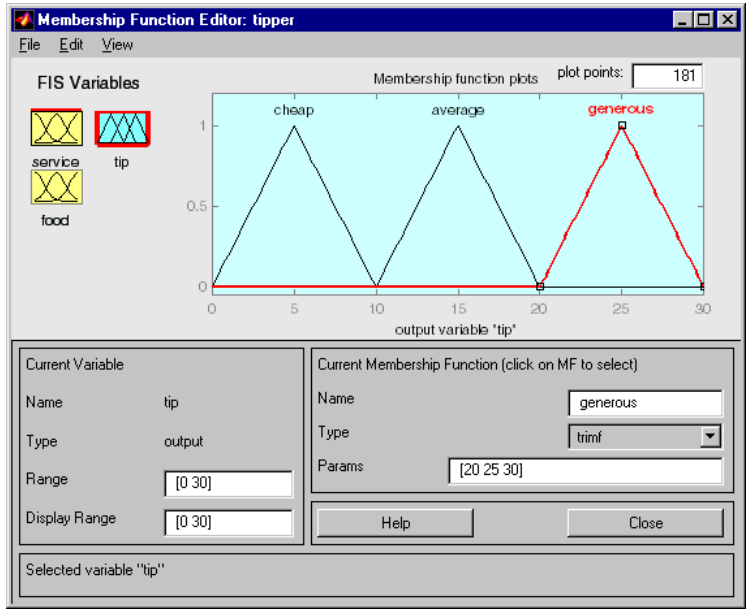
1. Select the **input** variable, `service`, by double-clicking on it. Set both the **Range** and the **Display Range** to the vector `[0 10]`.
2. Select **Add MFs...** from the **Edit** menu. The window below opens.



3. Use the tab to choose `gaussmf` for **MF Type** and `3` for **Number of MFs**. This adds three Gaussian curves to the input variable service.
4. Click once on the curve with the leftmost *hump*. Change the name of the curve to `poor`. To adjust the shape of the membership function, either use the mouse, as described above, or type in a desired parameter change, and then click on the **membership function**. The default parameter listing for this curve is `[1.5 0]`.
5. Name the curve with the middle hump, `good`, and the curve with the rightmost hump, `excellent`. Reset the associated parameters if desired.
6. Select the **input** variable, **food**, by clicking on it. Set both the **Range** and the **Display Range** to the vector `[0 10]`.
7. Select **Add MFs...** from the **Edit** menu and add two `trapmf` curves to the input variable `food`.
8. Click once directly on the curve with the leftmost trapezoid. Change the name of the curve to `rancid`. To adjust the shape of the membership function, either use the mouse, as described above, or type in a desired parameter change, and then click on the **membership function**. The default parameter listing for this curve is `[0 0 1 3]`.
9. Name the curve with the rightmost trapezoid, `delicious`, and reset the associated parameters if desired.

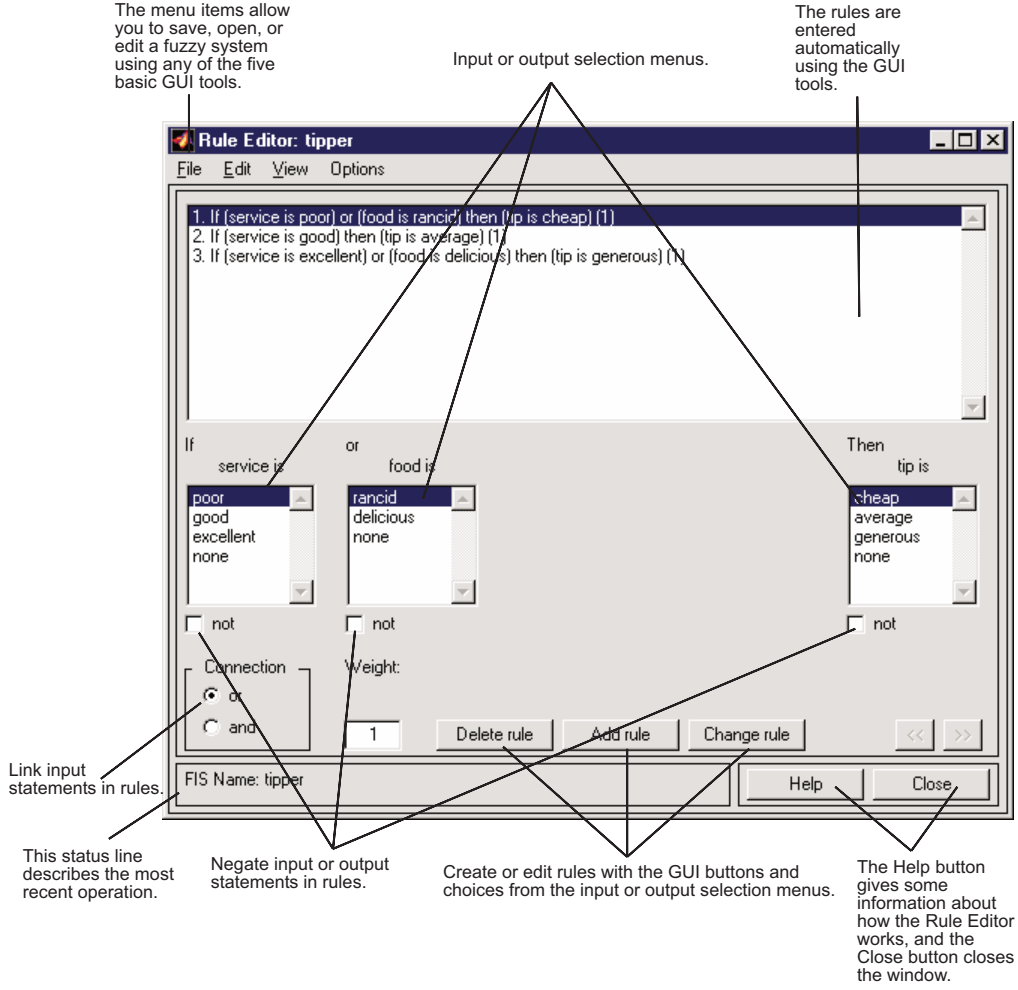
Next you need to create the membership functions for the output variable, **tip**. To create the output variable membership functions, use the Variable Palette on the left, selecting the output variable, **tip**. The inputs ranged from 0 to 10, but the output scale is going to be a tip between 5 and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

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



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

The Rule Editor



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

associated quality. Rules may be changed, deleted, or added, by clicking on the appropriate button.

The Rule Editor also has some familiar landmarks, similar to those in the FIS Editor and the Membership Function Editor, including the menu bar and the status line. The **Format** pop-up menu is available from the **Options** pull-down menu from the top menu bar—this is used to set the format for the display. Similarly, **Language** can be set from under **Options** as well. The **Help** button will bring up a MATLAB Help window.

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

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

The resulting rule is

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

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

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

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

To change a rule, first click on the rule to be changed. Next make the desired changes to that rule, and then click **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're looking at the verbose form of the rules. Try changing it to `symbolic`. You will see

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

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

1, 1 (1) : 1
2, 2 (1) : 1
3, 3 (1) : 1

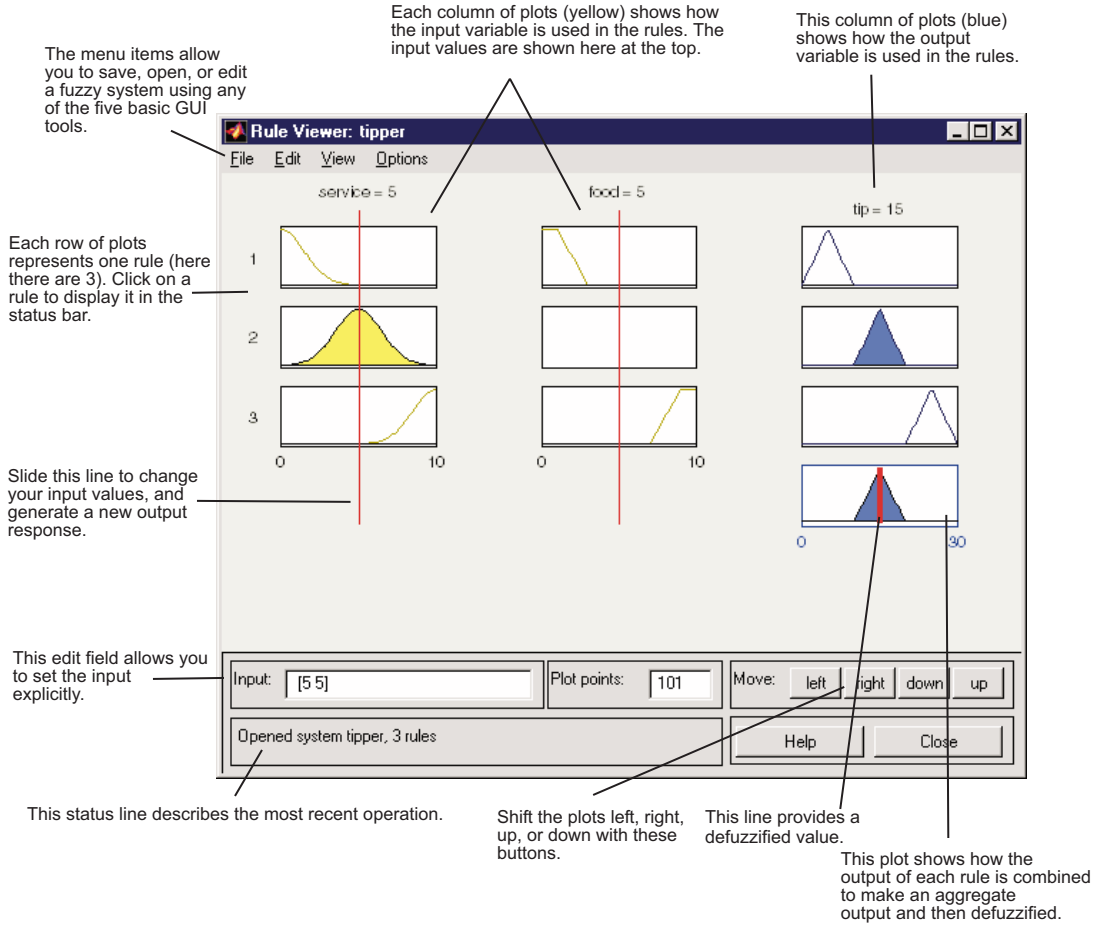
This is the version that the machine deals with. The first column in this structure corresponds to the input variable, the second column corresponds to the output variable, the third column displays the weight applied to each rule, and the fourth column is shorthand that indicates whether this is an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

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

At this point, the fuzzy inference system has been completely defined, in that the variables, membership functions, and the rules necessary to calculate tips

are in place. 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. This is exactly the purpose of the Rule Viewer, the next of the GUI tools we'll look at. From the **View** menu, select **View rules....**

The Rule Viewer



The Rule Viewer displays a roadmap of the whole fuzzy inference process. it is based on the fuzzy inference diagram described in the previous section. You see a single figure window with 10 small plots nested in it. The three small

plots across the top of the figure represent the antecedent and consequent of the first rule. Each rule is a row of plots, and each column is a variable. The first two columns of plots (the six yellow plots) show the membership functions referenced by the antecedent, or the if-part of each rule. The third column of plots (the three blue plots) shows the membership functions referenced by the consequent, or the then-part of each rule. If you click once on a rule number, the corresponding rule will be displayed at the bottom of the figure. Notice that under **food**, there is a plot which is blank. This corresponds to the characterization of none for the variable **food** in the second rule. The fourth plot in the third column of plots represents the aggregate weighted decision for the given inference system. This decision will depend on the input values for the system.

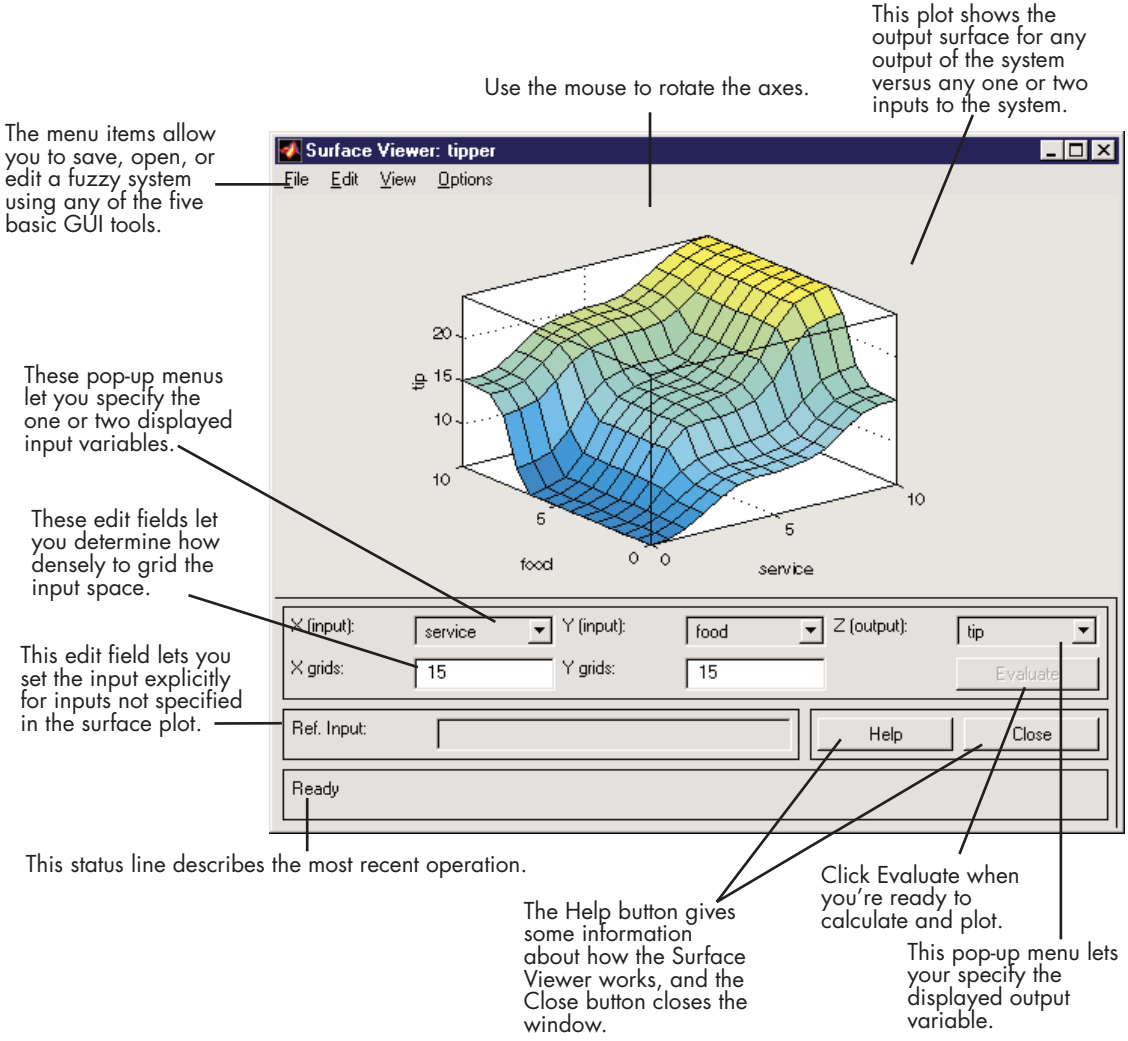
There are also the now familiar items like the status line and the menu bar. In the lower right there is a text field into which you can enter specific input values. For the two-input system, you will enter an input vector, `[9 8]`, for example, and then click on **Input**. You can also adjust these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

The Rule Viewer allows you to interpret the entire fuzzy inference process at once. The Rule Viewer also shows how the shape of certain membership functions influences the overall result. Since it plots every part of every rule, it can become unwieldy for particularly large systems, but, for a relatively small number of inputs and outputs, it performs well (depending on how

much screen space you devote to it) with up to 30 rules and as many as 6 or 7 variables.

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

The Surface Viewer



Upon opening the Surface Viewer, we are presented with a two-dimensional curve that represents the mapping from service quality to tip amount. Since this is a one-input one-output case, we can see the entire mapping in one plot. Two-input one-output systems also work well, as they generate three-dimensional plots that MATLAB can adeptly manage. When we move

beyond three dimensions overall, we start to encounter trouble displaying the results. Accordingly, the Surface Viewer is equipped with pop-up menus that let you select any two inputs and any one output for plotting. Just below the pop-up menus are two text input fields that let you determine how many *x-axis* and *y-axis* grid lines you want to include. This allows you to keep the calculation time reasonable for complex problems. Clicking the **Evaluate** button initiates the calculation, and the plot comes up soon after the calculation is complete. To change the *x*-axis or *y*-axis grid after the surface is in view, simply change the appropriate text field, and click either **X-grids** or **Y-grids**, according to which text field you changed, to redraw the plot.

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

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

Importing and Exporting from the GUI Tools

When you save a fuzzy system to disk, you're saving an ASCII text FIS file representation of that system with the file suffix `.fis`. This text file can be edited and modified and is simple to understand. When you save your fuzzy system to the MATLAB workspace, you're creating a variable (whose name

you choose) that will act as a MATLAB structure for the FIS system. FIS files and FIS structures represent the same system.

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

Customizing Your Fuzzy System

If you want to include customized functions as part of your use of the Fuzzy Logic Toolbox, you must follow a few guidelines. You may substitute customized functions for the AND, OR, aggregation, and defuzzification methods, provided your customized functions work in a similar way to `max`, `min`, or `prod` in MATLAB. That is, they must be able to operate down the columns of a matrix.

In MATLAB, for a matrix `x`, `min(x)` returns a row vector containing the minimum element from each column. For N-D arrays, `min(x)` operates along the first non-singleton dimension. The function `min(x,y)`, on the other hand, returns an array the same size as `x` and `y` populated with the smallest elements from `x` or `y`. Either one can be a scalar. Functions such as `max`, `prod`, and `mean` operate in a similar manner.

In the Fuzzy Logic Toolbox, the implication method performs an element by element matrix operation, similar to the `min(x,y)` function in MATLAB, as in

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

After you have defined your custom function using the procedure described in the next section, use the FIS Editor to substitute your custom function for a standard function. To do this:

1. Open the FIS Editor by typing `fuzzy` at the command line prompt.
2. In the lower left panel, locate the method you want to replace.

3 In the drop-down menu, select `Custom`. A dialog box appears.

4 Enter the name of your custom function and click **OK**.

Your custom function then replaces the standard function in all subsequent operations.

Custom Membership Functions

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

To define a custom membership function named `custmf`:

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

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

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

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

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

6 Click **OK**.

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

```
function out = testmf1(x, params)
for i=1:length(x)
if x(i)<params(1)
 y(i)=params(1);
elseif x(i)<params(2)
 y(i)=params(2);
elseif x(i)<params(3)
```

```
 y(i)=params(3);
elseif x(i)<params(4)
 y(i)=params(4);
elseif x(i)<params(5)
 y(i)=params(5);
elseif x(i)<params(6)
 y(i)=params(6);
elseif x(i)<params(7)
 y(i)=params(7);
elseif x(i)<params(8)
 y(i)=params(8);
else
 y(i)=0;
end
end
out=.1*y';
```

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

Working from the Command Line

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

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

MATLAB will respond with

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

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

```
a.type
```

for example. MATLAB will respond with

```
ans =
mamdani
```

The function

```
getfis(a)
```

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

`getfis(a)` returns

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

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

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

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

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

The `structure.field` syntax also generates this information. For more information on the syntax for MATLAB structures and cell arrays, see the MATLAB documentation.

For example, type

```
a.input
```

or

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

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

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

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

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

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

```
showfis(a)
```

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

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

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

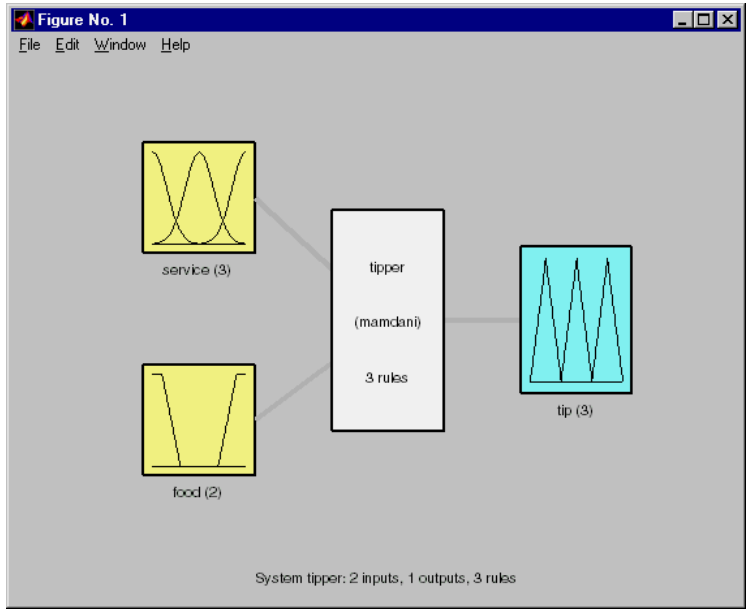
If, in addition, `a` is a Sugeno-type FIS, then `anfisedit(a)` will display the ANFIS Editor GUI.

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

System Display Functions

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

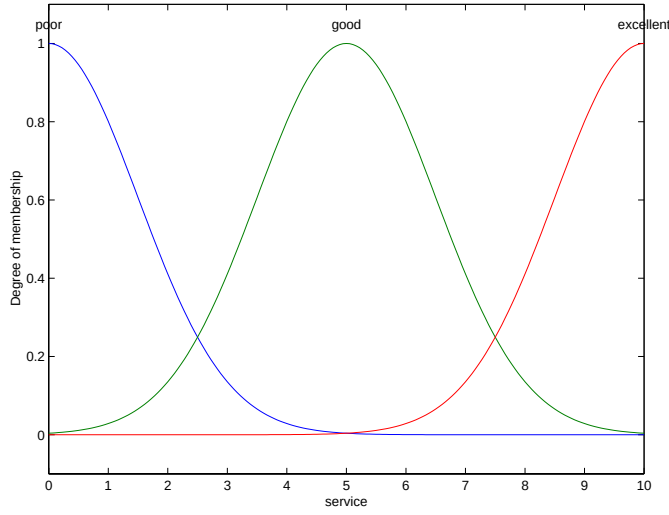
```
plotfis(a)
```



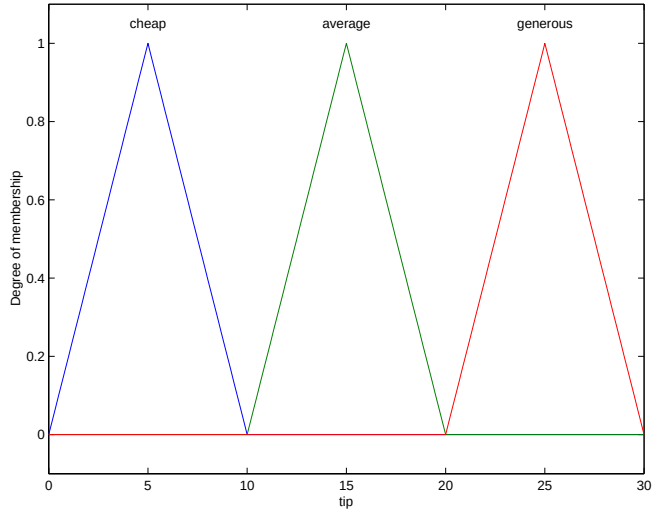
After closing any open MATLAB figures or GUI windows, the function `plotmf` plots all the membership functions associated with a given variable as follows.

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

returns



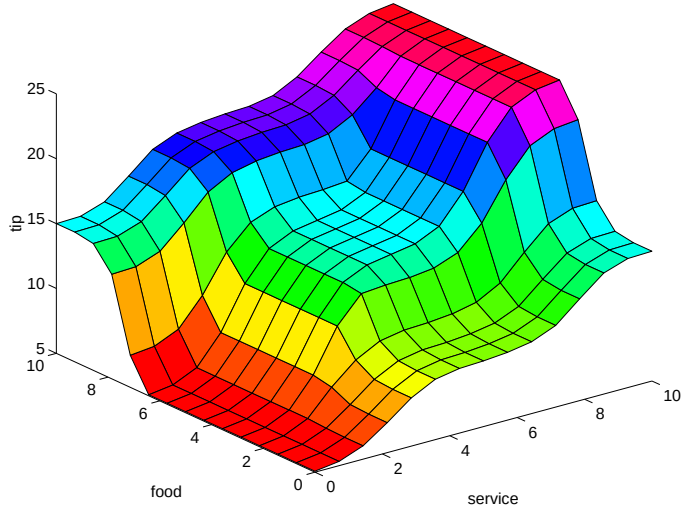

```
plotmf(a,'output',1)
```



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

Finally, the function `gensurf` will plot any one or two inputs versus any one output of a given system. The result is either a two-dimensional curve, or a three-dimensional surface. Note that when there are three or more inputs, `gensurf` must be generated with all but two inputs fixed, as is described in the description of `genfis` in Chapter 4, “Functions — Alphabetical List”.

```
gensurf(a)
```



Building a System from Scratch

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

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

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

If input1 is MF1 or input2 is MF3, then output1 is MF2 (weight = 0.5)

This rule is turned into a structure according to the following logic. If there are *m* inputs to a system and *n* outputs, then the first *m* vector entries of the rule structure correspond to inputs 1 through *m*. The entry in column 1 is the index number for the membership function associated with input 1. The entry in column 2 is the index number for the membership function associated with input 2, and so on. The next *n* columns work the same way for the outputs. Column *m* + *n* + 1 is the weight associated with that rule (typically 1) and column *m* + *n* + 2 specifies the connective used (where AND = 1 and OR = 2). The structure associated with the rule shown above is

```
1 3 2 0.5 2
```

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

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

```
a.output(1).name='tip';
a.output(1).range=[0 30];
a.output(1).mf(1).name='cheap'
a.output(1).mf(1).type='trimf';
a.output(1).mf(1).params=[0 5 10];
a.output(1).mf(2).name='average';
a.output(1).mf(2).type='trimf';
a.output(1).mf(2).params=[10 15 20];
a.output(1).mf(3).name='generous';
a.output(1).mf(3).type='trimf';
a.output(1).mf(3).params=[20 25 30];
a.rule(1).antecedent=[1 1];
a.rule(1).consequent=[1];
a.rule(1).weight=1;
a.rule(1).connection=2;
a.rule(2).antecedent=[2 0];
a.rule(2).consequent=[2];
a.rule(2).weight=1;
a.rule(2).connection=1;
a.rule(3).antecedent=[3 2];
a.rule(3).consequent=[3];
a.rule(3).weight=1;
a.rule(3).connection=2
```

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

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

```
1 1 1 1 2
2 0 2 1 1
3 2 3 1 2 ];
a=addrule(a,ruleList);
```

FIS Evaluation

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

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

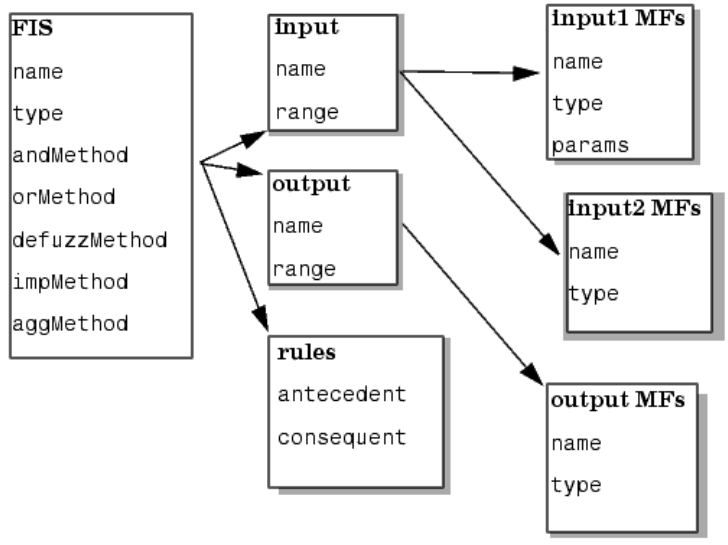
This function can also be used for multiple collections of inputs, since different input vectors are represented in different parts of the input structure. By doing multiple evaluations at once, you get a tremendous boost in speed.

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

The FIS Structure

The FIS structure is the MATLAB object that contains all the fuzzy inference system information. This structure is stored inside each GUI tool. Access functions such as `getfis` and `setfis` make it easy to examine this structure.

All the information for a given fuzzy inference system is contained in the FIS structure, including variable names, membership function definitions, and so on. This structure can itself be thought of as a hierarchy of structures, as shown in the following diagram.



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

```
showfis(a)
1.  Name             tipper
2.  Type             mamdani
3.  Inputs/Outputs   [ 2 1 ]
4.  NumInputMFs      [ 3 2 ]
5.  NumOutputMFs     3
6.  NumRules         3
7.  AndMethod        min
8.  OrMethod         max
9.  ImpMethod        min
10. AggMethod        max
11. DefuzzMethod     centroid
12. InLabels         service
13.                  food
14. OutLabels        tip
15. InRange          [ 0 10 ]
16.                  [ 0 10 ]
17. OutRange         [ 0 30 ]
```

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

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

Saving FIS Files on Disk

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

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

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

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

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

[Input2]
Name='food'
Range=[0 10]
NumMFs=2
MF1='rancid':'trapmf',[0 0 1 3]
MF2='delicious':'trapmf',[7 9 10 10]
```

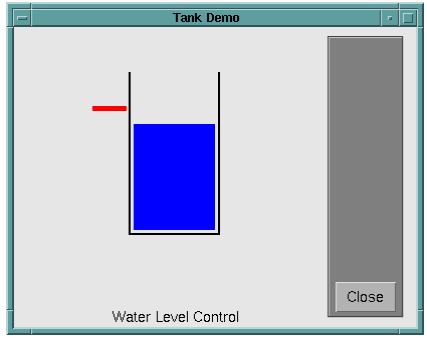
```
[Output1]
Name='tip'
Range=[0 30]
NumMFs=3
MF1='cheap':'trimf',[0 5 10]
MF2='average':'trimf',[10 15 20]
MF3='generous':'trimf',[20 25 30]

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

Working with Simulink

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

An Example: Water Level Control

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



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

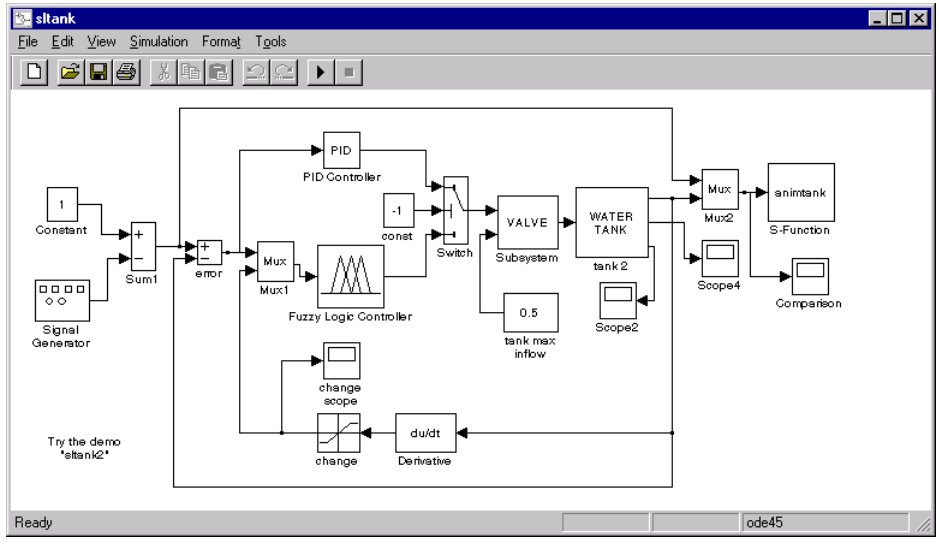
1. *If (level is okay) then (valve is no_change) (1)*
2. *If (level is low) then (valve is open_fast) (1)*
3. *If (level is high) then (valve is close_fast) (1)*

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

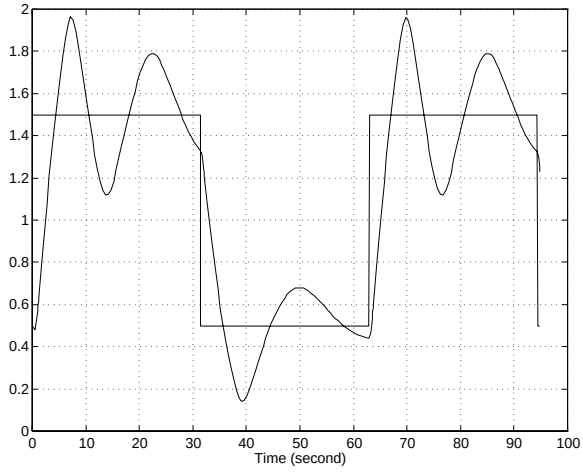
It contains a Simulink block called the Fuzzy Logic Controller block. The Simulink block diagram for this system is `sltank`. Typing

```
sltank
```

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



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

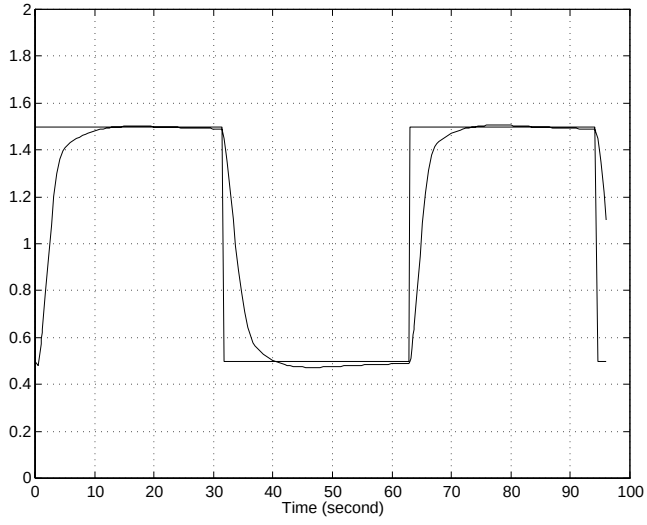


We need to add another input, the water level's rate of change, to slow down the valve movement when we get close to the right level.

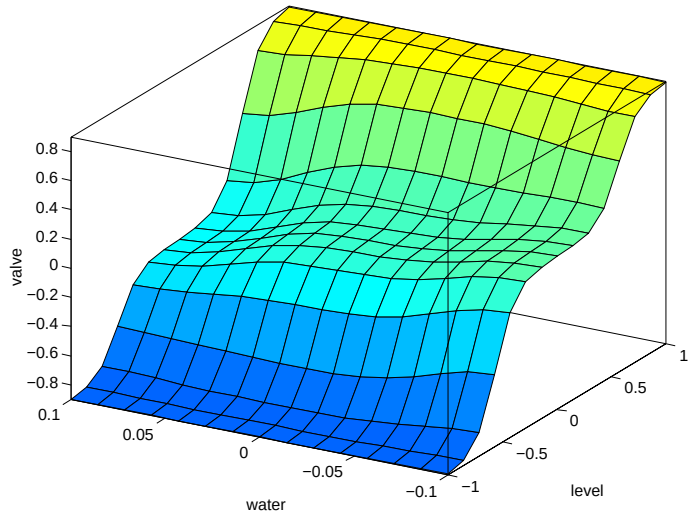
4. *If (level is good) and (rate is negative), then (valve is close_slow) (1)*

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

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



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



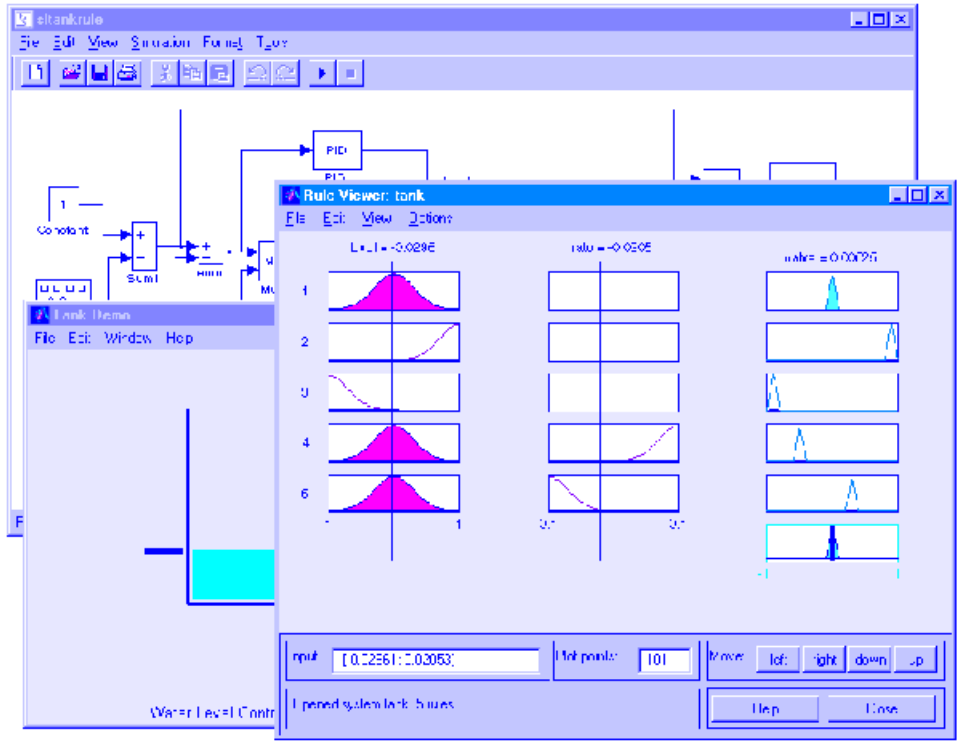
Because the MATLAB technical computing environment supports so many tools (like the Control System Toolbox, the Neural Network Toolbox, and so on), you can, for example, easily make a comparison of a fuzzy controller versus a linear controller or a neural network controller.

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

```
sltankrule
```

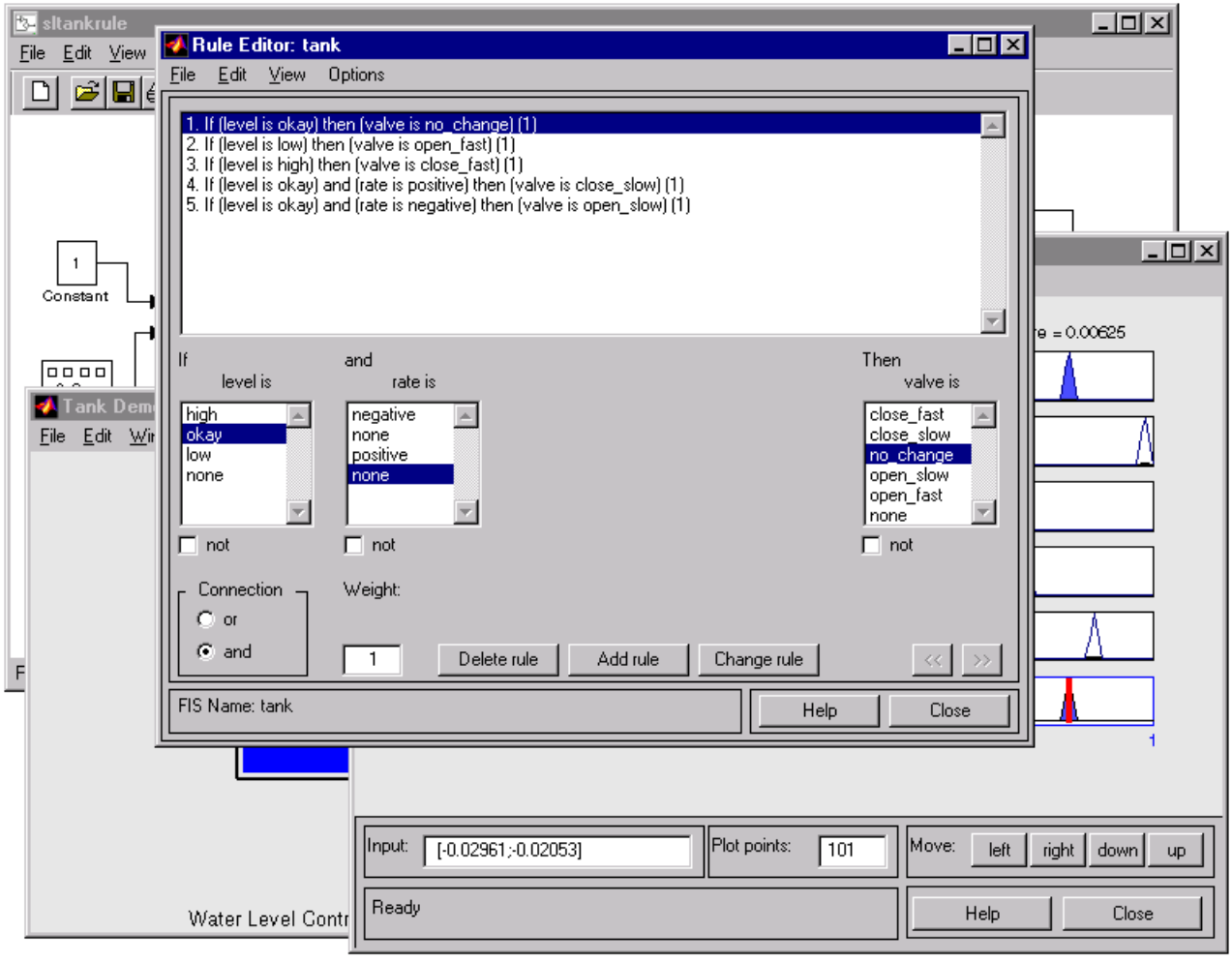
This demo contains a block called the Fuzzy Controller With Rule Viewer block.

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



The Rule Viewer that opens during the simulation can be used to access the Membership Function Editor, the Rule Editor, or any of the other GUIs, (see “The Membership Function Editor” on page 2-37, or “The Rule Editor” on page 2-41, for more information).

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



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

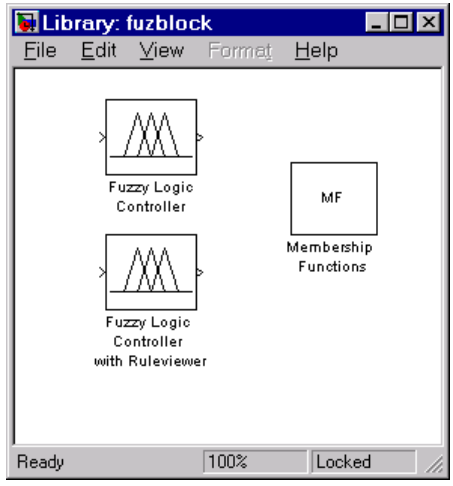
Building Your Own Fuzzy Simulink Models

To build your own Simulink systems that use fuzzy logic, simply copy the Fuzzy Logic Controller block out of `sltank` (or any of the other Simulink demo systems available with the toolbox) and place it in your own block diagram. You can also find the Fuzzy Logic Controller block in the Fuzzy Logic Toolbox library, which you can open either by selecting **Fuzzy Logic Toolbox** in the Simulink Library Browser, or by typing

```
fuzblock
```

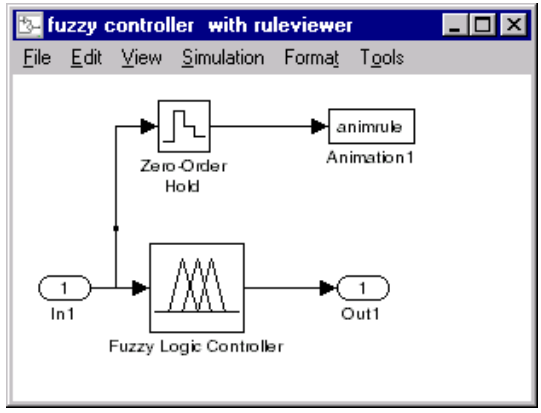
at the MATLAB prompt.

The following library appears.



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.

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



To initialize the Fuzzy Logic Controller blocks (with or without the Rule Viewer), double-click on the block and enter the name of the structure variable describing your FIS. This variable must be located in the MATLAB workspace.

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. For more information about the Fuzzy Logic Controller block, see the `fuzblock` reference page.

The Fuzzy Wizard cannot handle 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. For more information, see the `sffis` reference page.

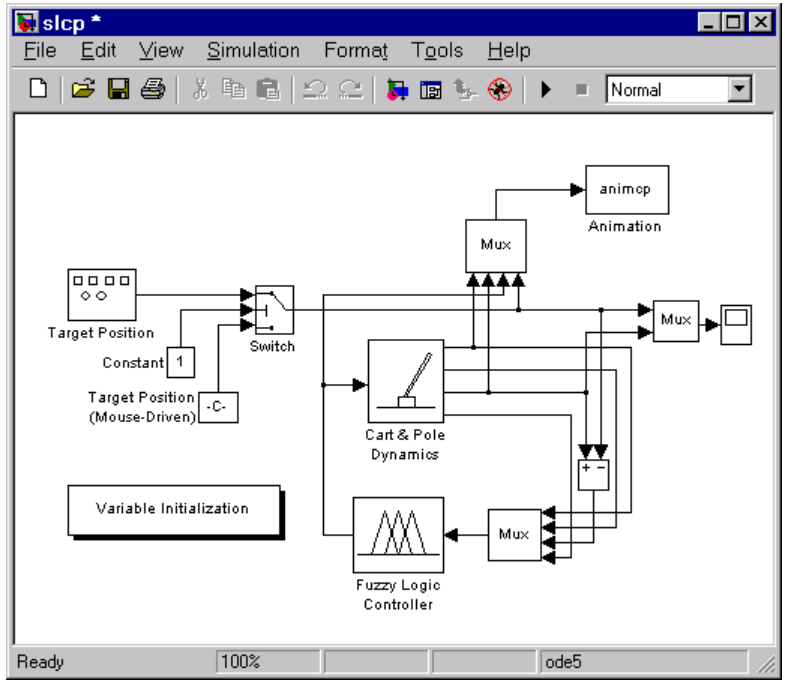
Example: Cart and Pole Simulation

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

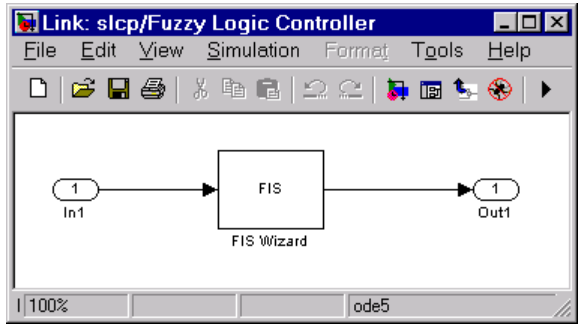
```
slcp
```

at the MATLAB prompt to open the simulation.

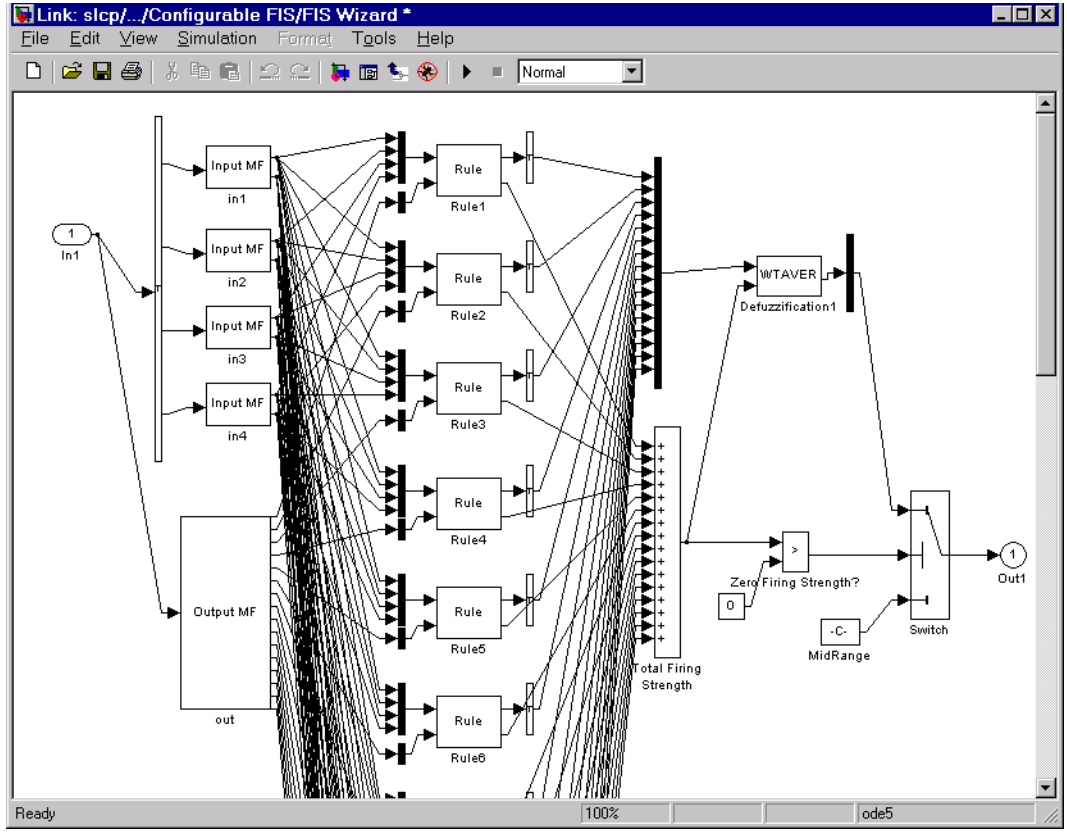
This model appears.



Right-click on the Fuzzy Logic Controller block and select **Look under mask** from the right-click menu. This subsystem opens.



Follow the same procedure to look under the mask of the FIS Wizard subsystem to see the implementation of your FIS. This following figure shows part of the implementation (the entire model is too large to show in this document).



As the figure shows, 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.

Sugeno-Type Fuzzy Inference

The fuzzy inference process we've been referring to so far is known as Mamdani's fuzzy inference method, the most common methodology. In this section, we discuss the so-called Sugeno, or Takagi-Sugeno-Kang, method of fuzzy inference. Introduced in 1985 [Sug85], it 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 exactly 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 = x and Input 2 = y, then Output is $z = ax + by + c$

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

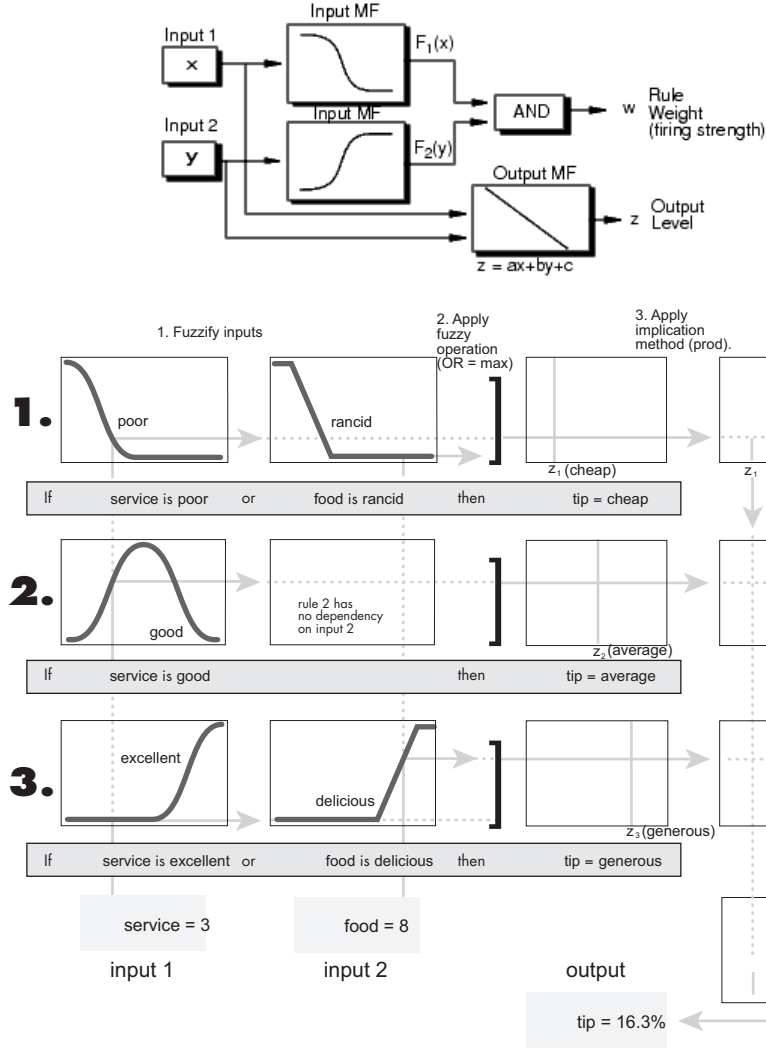
The output level z_i of each rule is weighted by the firing strength w_i of the rule. For example, for an AND rule with Input 1 = x and Input 2 = y, the firing strength is

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

where $F_{1,2}(.)$ 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

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

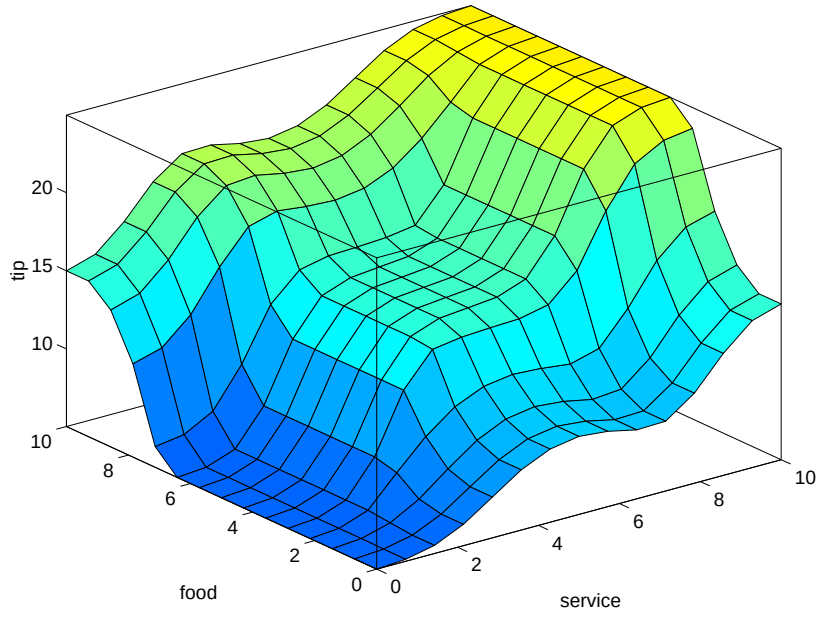
A Sugeno rule operates as shown in the following diagram.



The figure above 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 completely 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 will see it is almost the same as the Mamdani system we've been looking at.

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



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 very compact and efficient. Higher order Sugeno fuzzy models are possible, but they introduce significant complexity with little obvious merit. Sugeno fuzzy models whose output membership functions are greater than first order are not supported by the Fuzzy Logic Toolbox.

Because of the linear dependence of each rule on the input variables of a system, the Sugeno method is ideal for acting as an interpolating supervisor of multiple linear controllers that are to be applied, respectively, to different operating conditions of a dynamic nonlinear system. For example, the performance of an aircraft may change dramatically with altitude and Mach

number. Linear controllers, though easy to compute and well-suited to any given flight condition, must be updated regularly and smoothly to keep up with the changing state of the flight vehicle. A Sugeno fuzzy inference system is extremely well suited to the task of smoothly interpolating the linear gains that would be applied across the input space; it is a natural and efficient gain scheduler. Similarly, a Sugeno system is suited for modeling nonlinear systems by interpolating between multiple linear models.

An Example: Two Lines

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

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

The output variable has two membership functions.

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

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

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

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

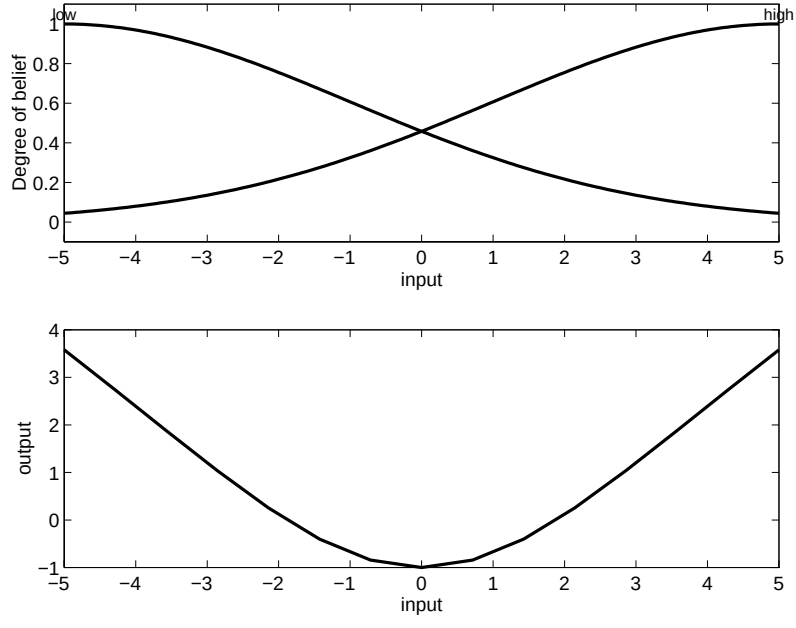
$$output = (1)*input + (-1)$$

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

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

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

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



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

Conclusion

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

Note 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. See Chapter 4, "Functions — Alphabetical List" for details.

Here 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.

anfis and the ANFIS Editor GUI

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

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

A Modeling Scenario

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

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

Model Learning and Inference Through ANFIS

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

What Is ANFIS?

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

FIS Structure and Parameter Adjustment

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

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

Familiarity Breeds Validation: Know Your Data

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

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

Model Validation Using Checking and Testing Data Sets

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

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

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

Constraints of anfis

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

- Be first or zeroth order Sugeno-type systems.
- Have a single output, obtained using weighted average defuzzification. All output membership functions must be the same type and either be linear or constant.
- Have no rule sharing. Different rules cannot share the same output membership function, namely the number of output membership functions must be equal to the number of rules.
- Have unity weight for each rule.

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

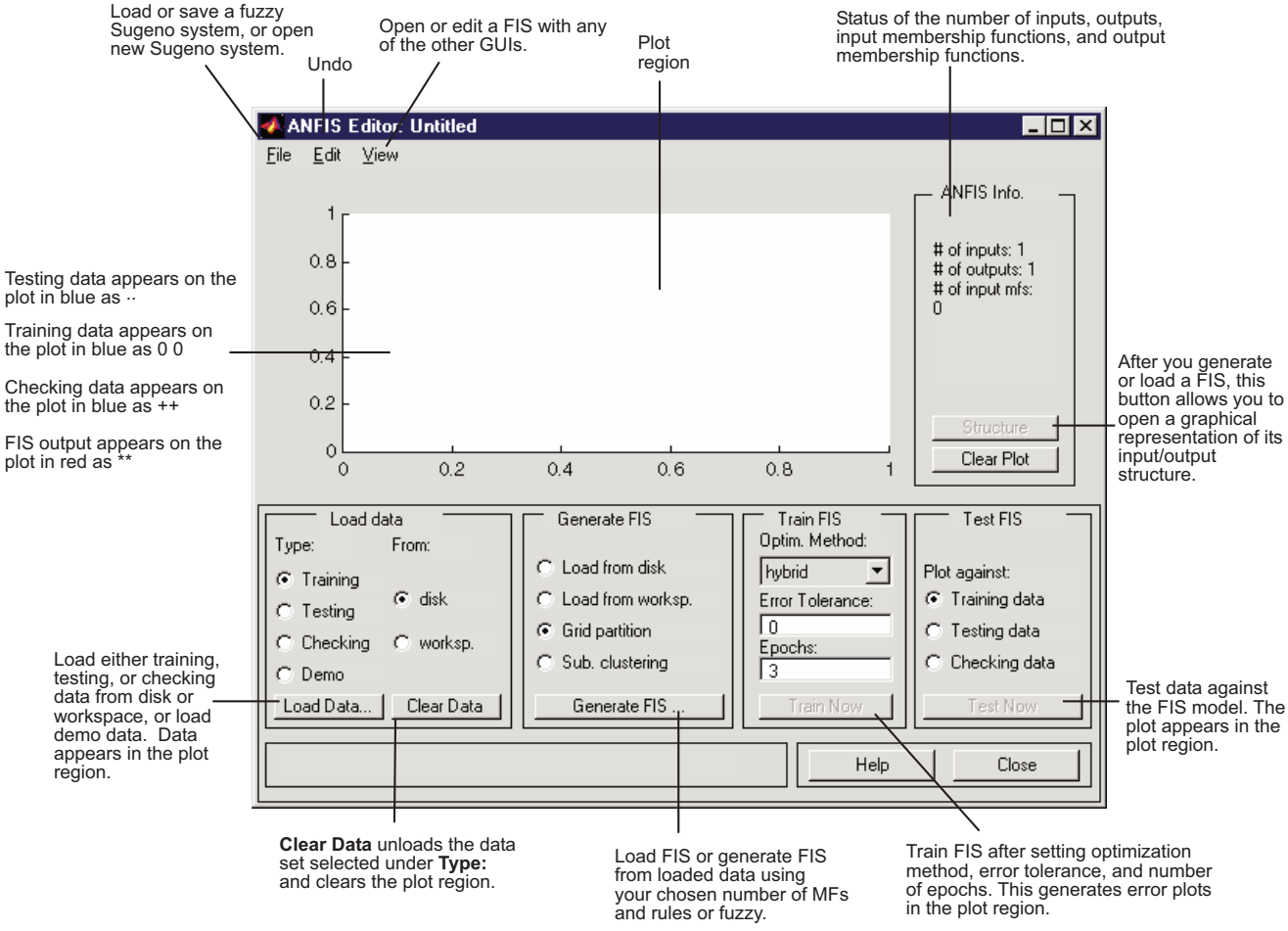
Moreover, `anfis` cannot accept all the customization options that basic fuzzy inference allows. That is, you cannot make your own membership functions and defuzzification functions; you must use the ones provided.

The ANFIS Editor GUI

To get started with the ANFIS Editor GUI, type

```
anfisedit
```

The following GUI will appear on your screen.



From this GUI you can:

- Load data (training, testing, and checking) by selecting appropriate radio buttons in the **Load data** portion of the GUI and then clicking **Load Data**

  The loaded data is plotted on the plot region.
- Generate an initial FIS model or load an initial FIS model using the options in the **Generate FIS** portion of the GUI
- View the FIS model structure once an initial FIS has been generated or loaded by clicking the **Structure** button
- Choose the FIS model parameter optimization method: backpropagation or a mixture of backpropagation and least squares (hybrid method)
- Choose the number of training epochs and the training error tolerance
- Train the FIS model by clicking the **Train Now** button

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

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

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

Data Formalities and the ANFIS Editor GUI: Checking and Training

To start training an FIS using either `anfis` or the ANFIS Editor GUI, first you need to have a training data set that contains desired input/output data pairs of the target system to be modeled. Sometimes you also want to have the optional testing data set that can check the generalization capability of the resulting fuzzy inference system, and/or a checking data set that helps with model overfitting during the training. The use of a testing data set and a checking data set for model validation are discussed in “Model Validation Using Checking and Testing Data Sets” on page 2-85. As we mentioned previously, overfitting is accounted for by testing the FIS trained on the training data against the checking data, and choosing the membership

function parameters to be those associated with the minimum checking error if these errors indicate model overfitting. You will have to examine your training error plots fairly closely in order to determine this. These issues are discussed later in an example. Usually these training and checking data sets are collected based on observations of the target system and are then stored in separate files.

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

ANFIS Editor GUI Example 1: Checking Data Helps Model Validation

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

Loading Data

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

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

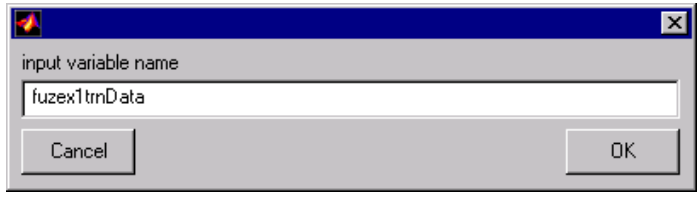
```
load fuzex1trnData.dat
load fuzex2trnData.dat
load fuzex1chkData.dat
load fuzex2chkData.dat
```

from the command line.

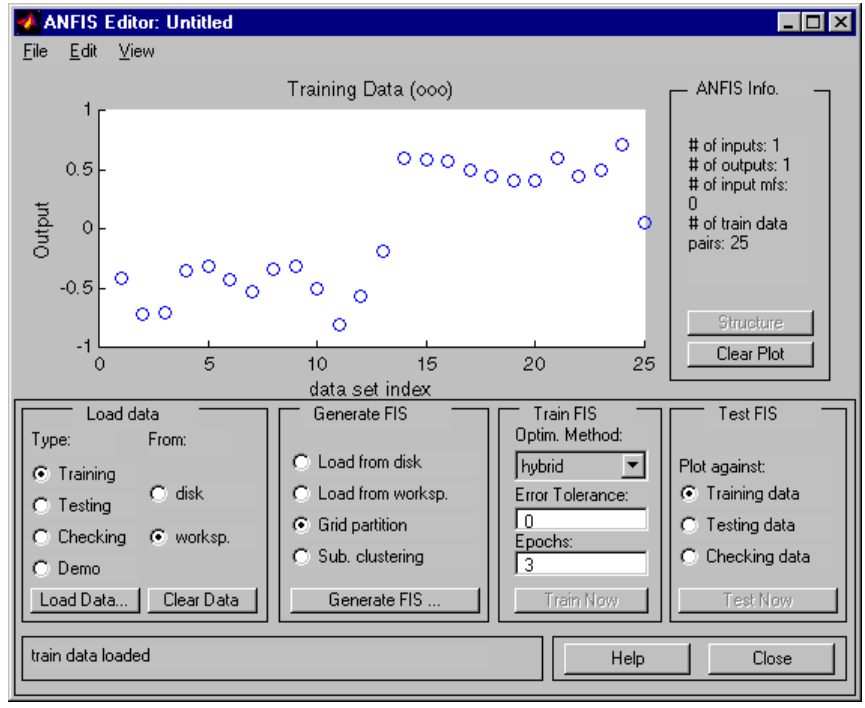
Note You may also want to load your data set from the `fuzzydemos` or any other directory on the disk, using the ANFIS Editor GUI, directly.

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

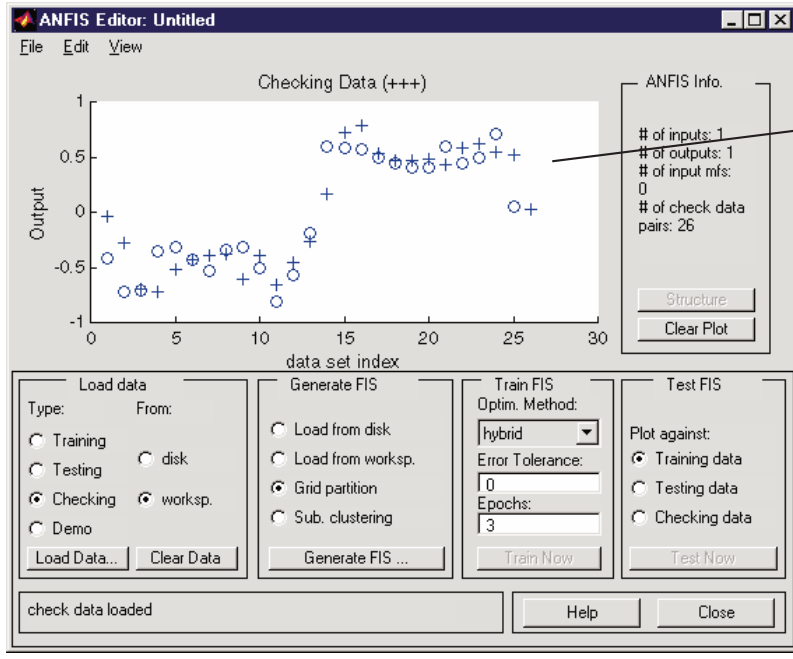
The small GUI window that opens allows you to type in a variable name from the workspace. Type in `fuzex1trnData`, as shown below.



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



Notice the horizontal axis is marked **data set index**. This index indicates the row from which that input data value was obtained (whether or not the input is a vector or a scalar). Next select the **Checking** check box in the **Type** column of the **Load data** portion of the GUI to load `fuzex1chkData` from the workspace. This data appears in the GUI plot as *plusses* superimposed on the training data.



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

Initializing and Generating Your FIS

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

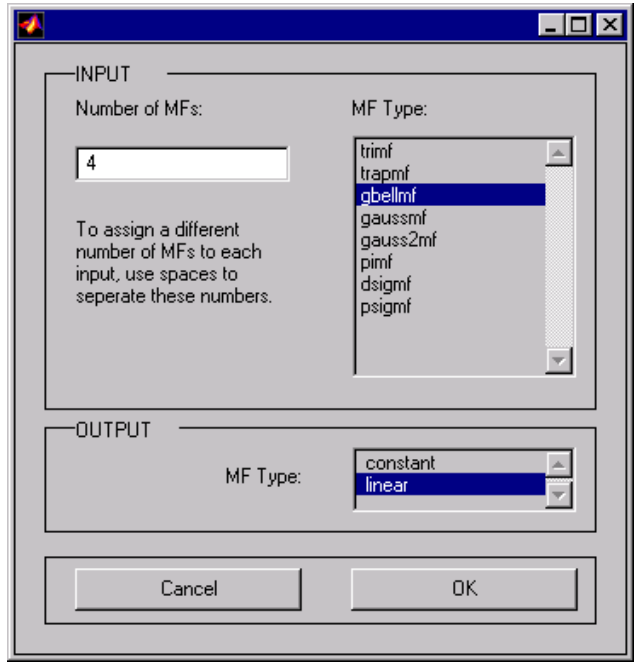
Automatic FIS Structure Generation with ANFIS

To initialize your FIS using `anfis`,

1. Choose **Grid partition,** the default partitioning method. The two partition methods, grid partitioning and subtractive clustering, are described later in “Fuzzy C-Means Clustering” on page 2-113, and in “Subtractive Clustering” on page 2-116.

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

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



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

Specifying Your Own Membership Functions for ANFIS

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

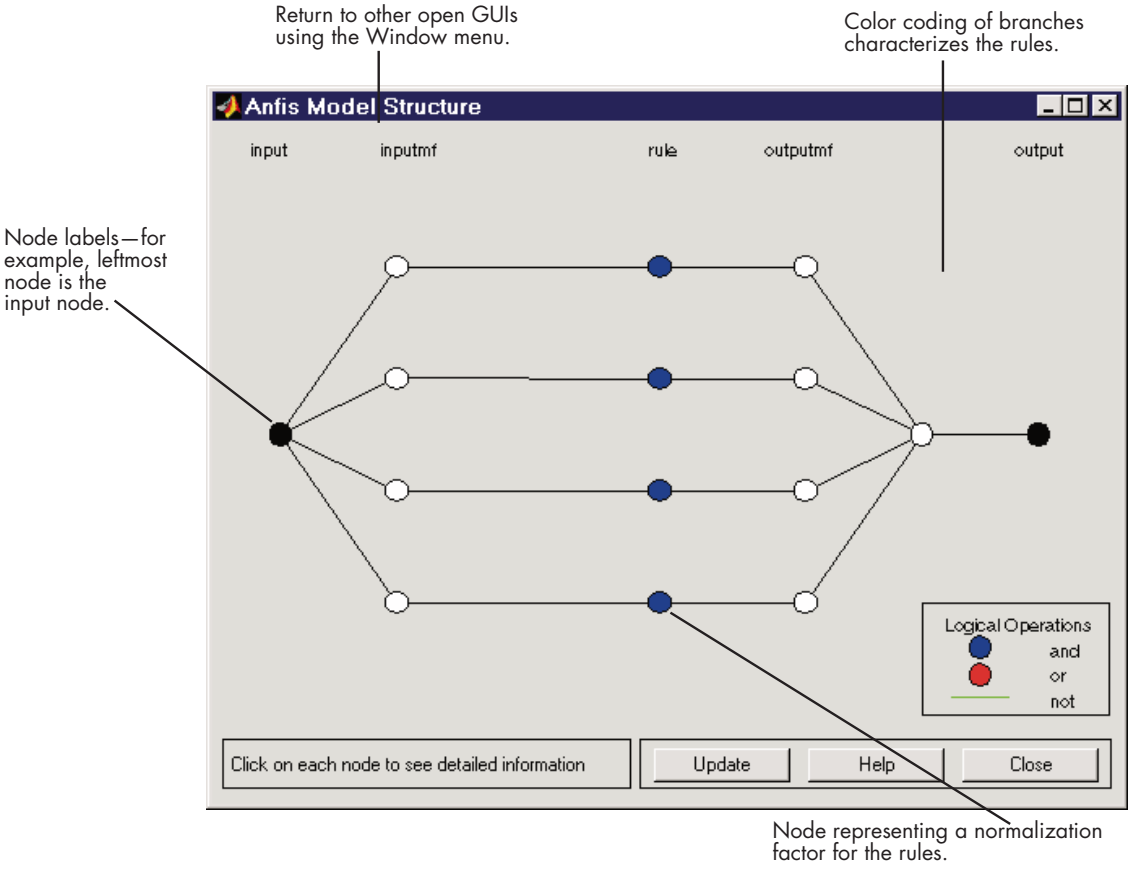
To define your own FIS structure and parameters:

1. Open the **Edit membership functions** menu item from the **View** menu.
2. Add your desired membership functions (the custom membership option will be disabled for `anfis`). The output membership functions must either be all constant or all linear. For carrying out this and the following step, see "The FIS Editor" on page 2-33 and "The Membership Function Editor" on page 2-37.
3. Select the **Edit rules** menu item in the **View** menu. Use the Rule Editor to generate the rules (see "The Rule Editor" on page 2-41).
4. Select the **Edit FIS Properties** menu item from the **View** menu. Name your FIS, and save it to either the workspace or the disk.
5. Use the **View** menu to return to the ANFIS Editor GUI to train the FIS.

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

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 GUI. A new GUI appears, as follows.



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

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

ANFIS Training

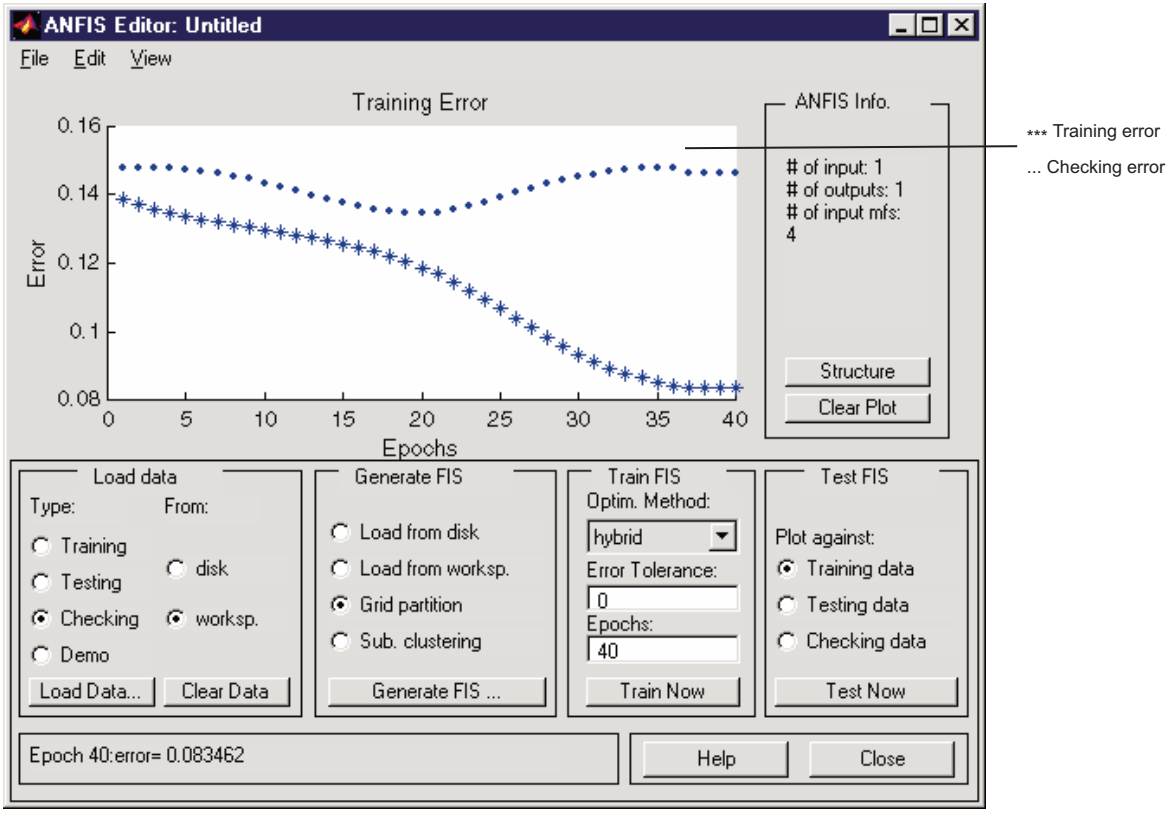
The two `anfis` parameter optimization method options available for FIS training are `hybrid` (the default, mixed least squares and backpropagation) and `backpropa` (backpropagation). **Error Tolerance** is used to create a

training stopping criterion, which is related to the error size. The training will stop after the training data error remains within this tolerance. This is best left set to 0 if you don't know how your training error is going to behave.

To start the training,

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

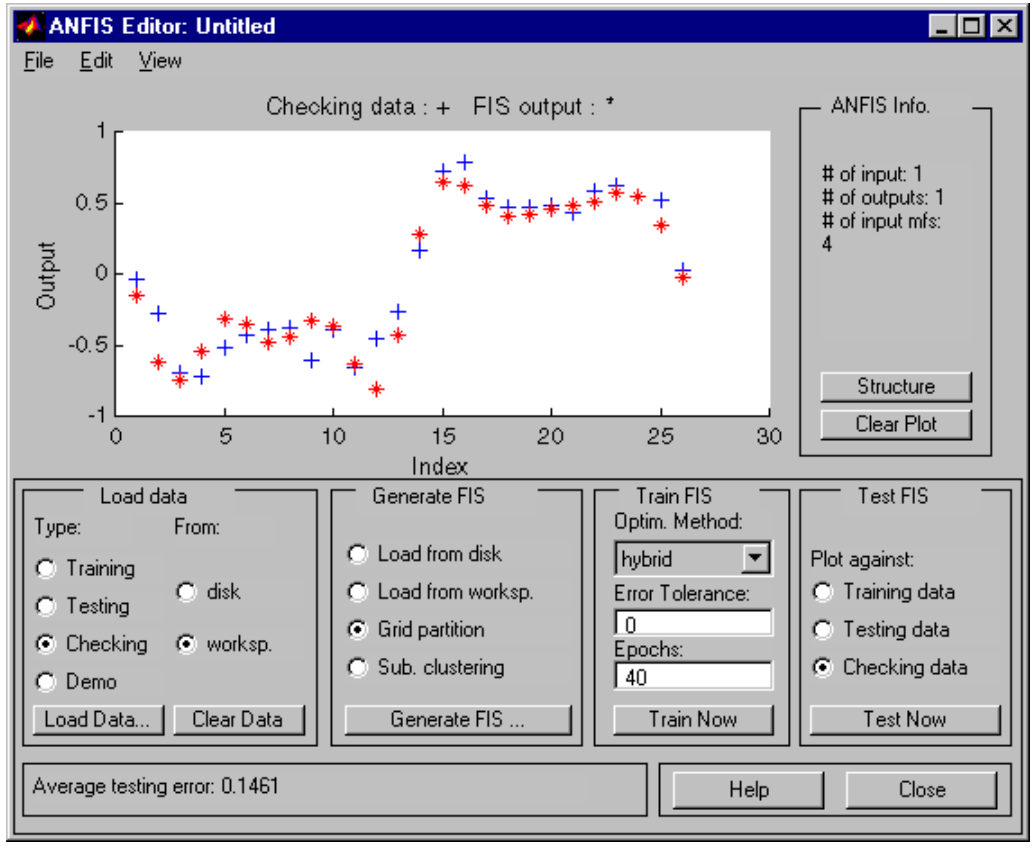
The following should appear on your screen.



Notice how the checking error decreases up to a certain point in the training and then it increases. This increase represents the point of model overfitting. `anfis` chooses the model parameters associated with the minimum checking error (just prior to this jump point). This is an example for which the checking data option of `anfis` is useful.

Testing Your Data Against the Trained FIS

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



Note on loading more data with anfis If you 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 will have to start a new `anfisedit` session from the command line.

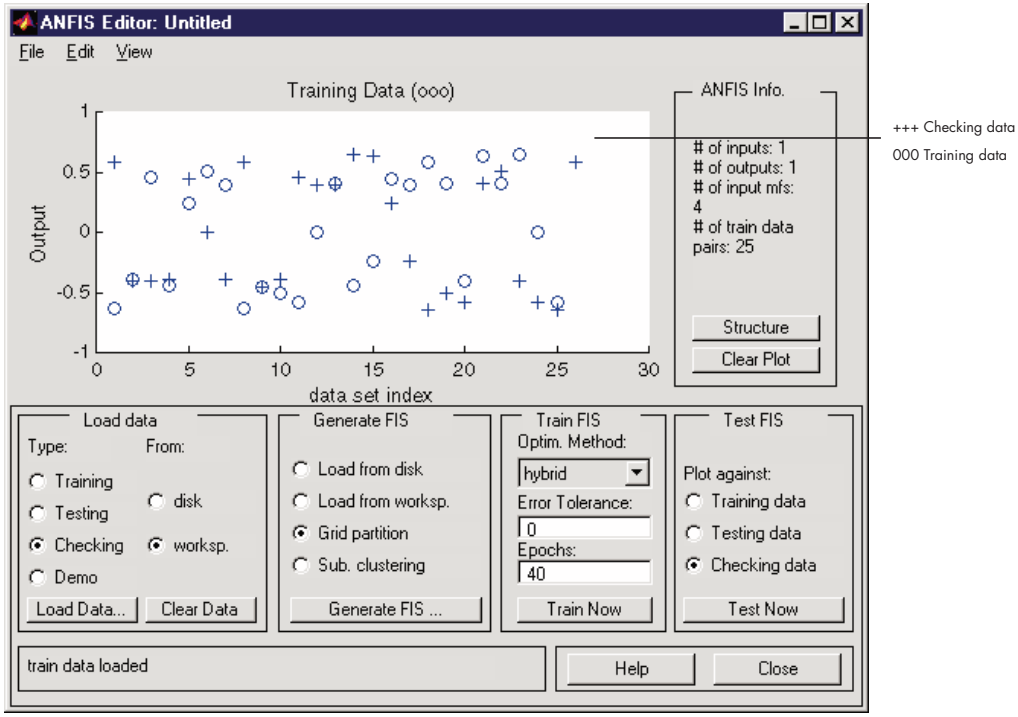
Note on the checking data option and clearing data If you don't want to use the checking data option of `anfis`, don't load any checking data before you train the FIS. If you decide to retrain your FIS with no checking data, you can unload the checking data in one of two ways. One method is to select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

ANFIS Editor GUI Example 2: Checking Data Does not Validate Model

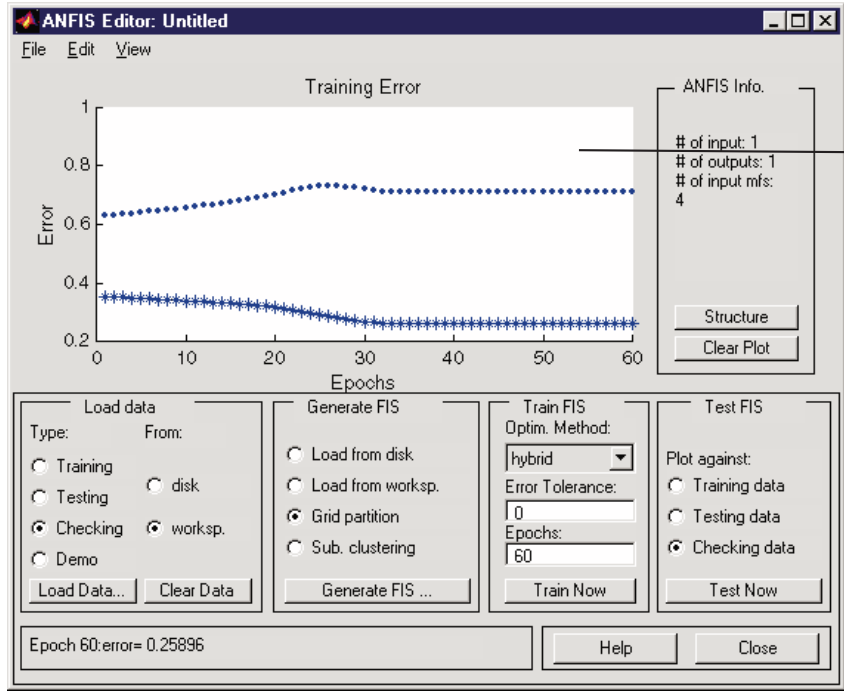
In this example, we examine what happens when the training and checking data sets are sufficiently different. To see how the ANFIS Editor GUI can be used to learn something about data sets and how they differ:

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

You should get something that looks like this.



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

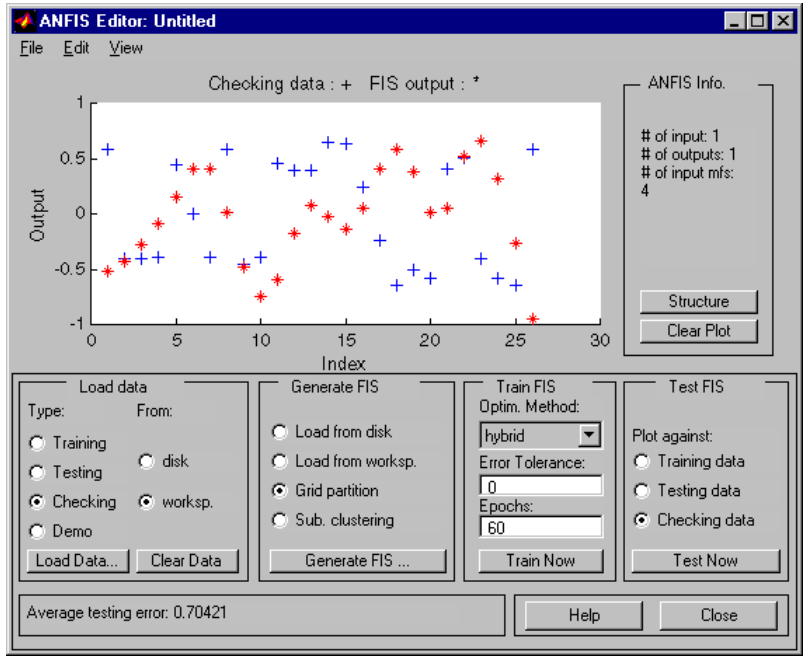


Notice the checking error is quite large. It appears that the minimum checking error occurs within the first epoch. Recall that using the checking data option with `anfis` automatically sets the FIS parameters to be those associated with the minimum checking error. Clearly this set of membership functions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

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

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

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



anfis from the Command Line

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

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

Using anfis for Chaotic Time-Series Prediction

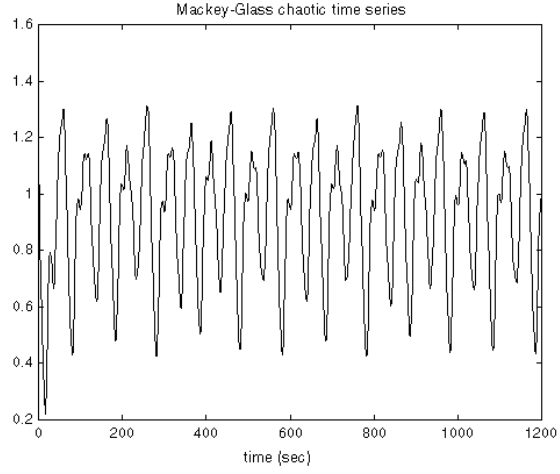
The demo `mgtsdemo` uses `anfis` to predict a time series that is generated by the following Mackey-Glass (MG) time-delay differential equation.

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

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

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

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



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

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

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

The output training data corresponds to the trajectory prediction.

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

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

Here is the code that generates this data.

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

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

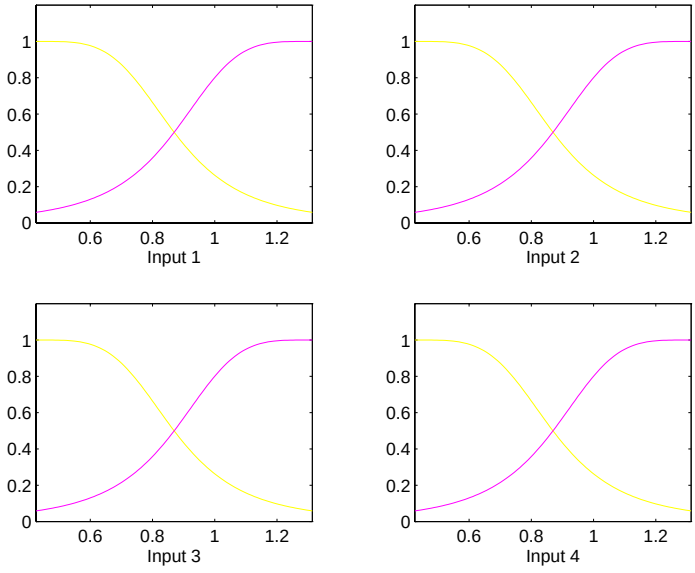
```
fismat = genfis1(trnData);
```

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

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

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

These initial membership functions are shown below.



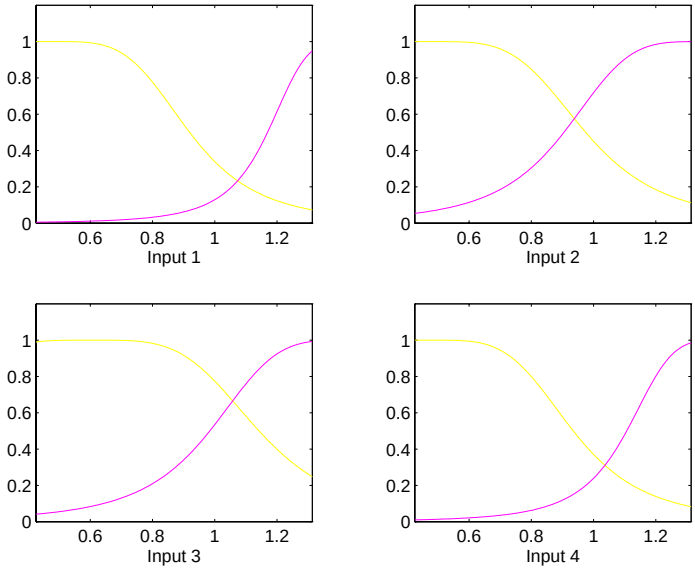
To start the training, type

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

This takes about 4 minutes on a Sun SPARCstation 2 for 10 epochs of training. Because the checking data option of `anfis` was invoked, the final FIS you choose would ordinarily be the one associated with the minimum checking error. This is stored in `fismat2`. The following code will plot these new membership functions.

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

Here is the result.



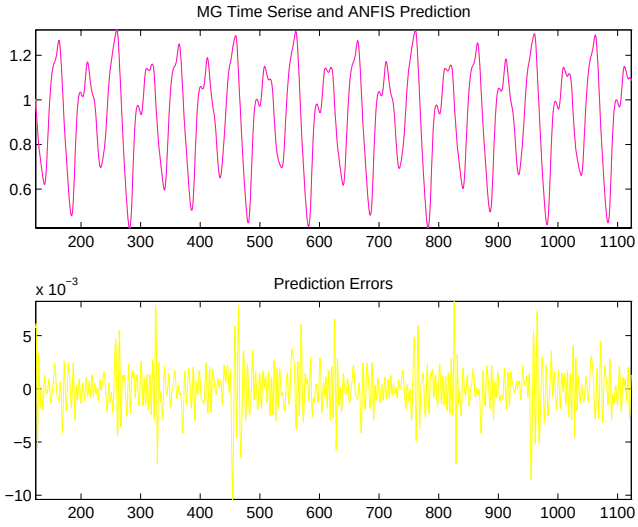
To plot the error signals, type

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

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

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

```
anfis_output = evalfis([trnData; chkData], fismat2);
index = 125:1124;
subplot(211), plot(t(index), [x(index) anfis_output]);
subplot(212), plot(t(index), x(index) - anfis_output);
```



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

More on anfis and the ANFIS Editor GUI

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

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

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

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

Training Data

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

Input FIS Structure

The input FIS structure, `fismat`, can be obtained either from any of the fuzzy editors: the FIS Editor, the Membership Function Editor, and the Rule Editor from the ANFIS Editor GUI, (which allows an FIS structure to be loaded from the disk or the workspace), or from the command line function, `genfis1` (for which you only need to give numbers and types of membership functions). The FIS structure contains both the model structure, (which specifies such items as the number of rules in the FIS, the number of membership functions for each input, etc.), and the parameters, (which specify the shapes of membership functions). There are two *methods* that `anfis` learning employs for updating membership function parameters: backpropagation for all parameters (a steepest descent method), and a hybrid method consisting of backpropagation for the parameters associated with the input membership

functions, and least squares estimation for the parameters associated with the output membership functions. As a result, the training error decreases, at least locally, throughout the learning process. Therefore, the more the initial membership functions resemble the optimal ones, the easier it will be for the model parameter training to converge. Human expertise about the target system to be modeled may aid in setting up these initial membership function parameters in the FIS structure.

Note that `genfis1` produces an FIS structure based on a fixed number of membership functions. This invokes the so-called *curse of dimensionality*, and causes an explosion of the number of rules when the number of inputs is moderately large, that is, more than four or five. The Fuzzy Logic Toolbox offers a method that will provide for some dimension reduction in the fuzzy inference system: you can generate an FIS structure using the clustering algorithm discussed in "Subtractive Clustering" on page 2-116. From the ANFIS Editor GUI, this algorithm is selected with a radio button before the FIS is generated. This subtractive clustering method partitions the data into groups called clusters, and generates an FIS with the minimum number rules required to distinguish the fuzzy qualities associated with each of the clusters.

Training Options

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

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

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

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

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

Display Options

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

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

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

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

Method

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

Output FIS Structure for Training Data

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

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

Training Error

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

Step-Size

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

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

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

Checking Data

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

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

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

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

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

As discussed in “ANFIS Editor GUI Example 2: Checking Data Does not Validate Model” on page 2-98, depending on the behavior of the checking data error, the resulting FIS may or may not be the one you should be using.

Output FIS Structure for Checking Data

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

Checking Error

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

Fuzzy Clustering

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

Fuzzy C-Means Clustering

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

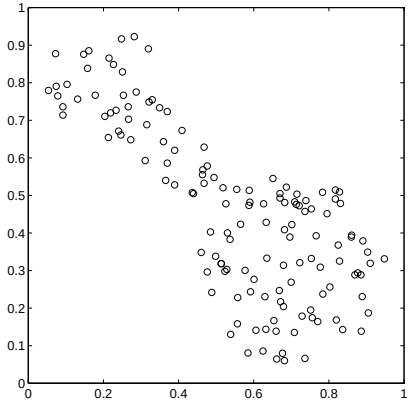
The Fuzzy Logic Toolbox command line function `fcm` starts with an initial guess for the cluster centers, which are intended to mark the mean location of each cluster. The initial guess for these cluster centers is most likely incorrect. Additionally, `fcm` assigns every data point a membership grade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

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

An Example: 2-D Clusters

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

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



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

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

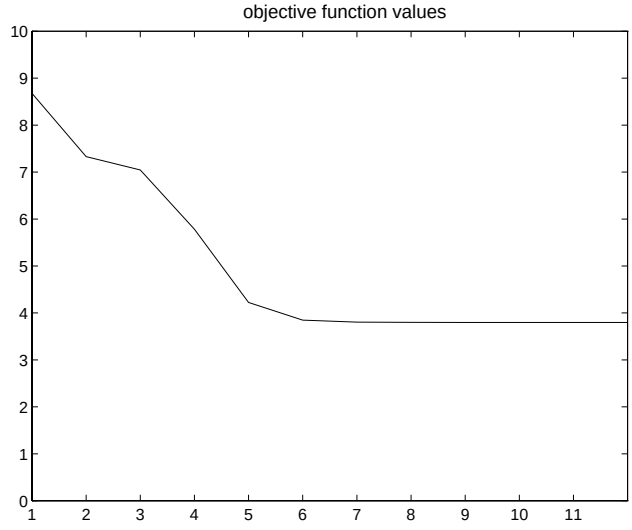
until the objective function is no longer decreasing much at all.

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

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

Plotting the objective function shows the progress of the clustering.

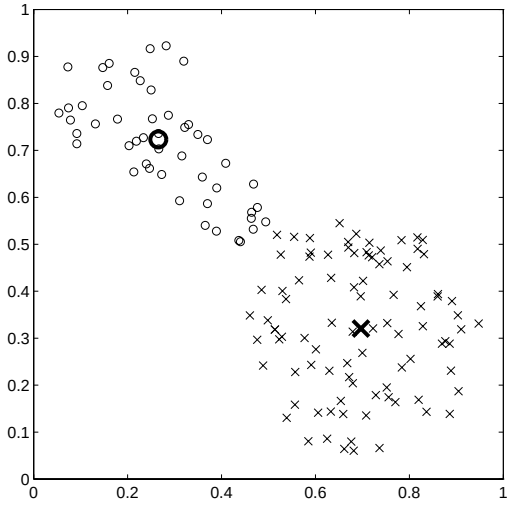
```
plot(objFcn)
```



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

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

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



Subtractive Clustering

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

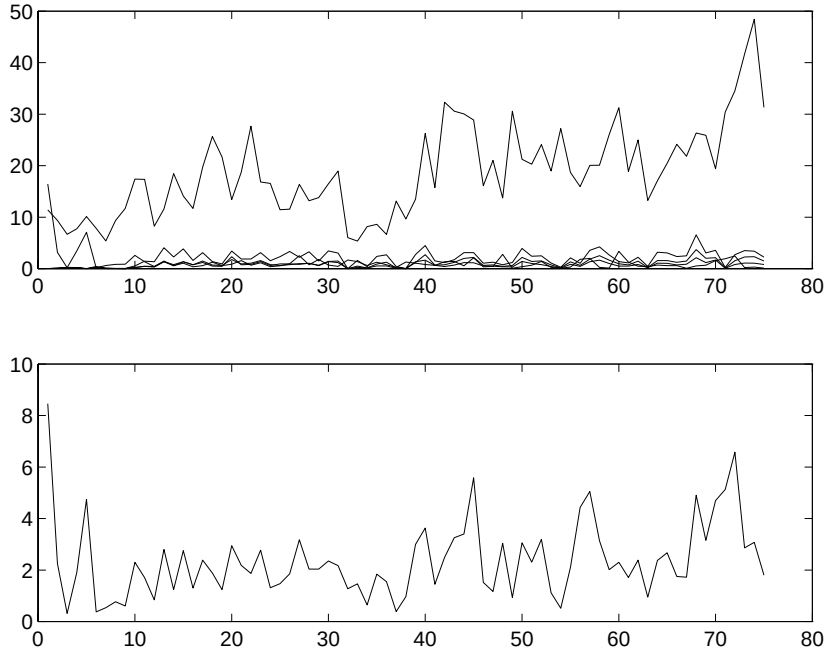
The `genfis2` function builds upon the `subclust` function to provide a fast, one-pass method to take input-output training data and generate a Sugeno-type fuzzy inference system that models the data behavior.

An Example: Suburban Commuting

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

Load the data by typing

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



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

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

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

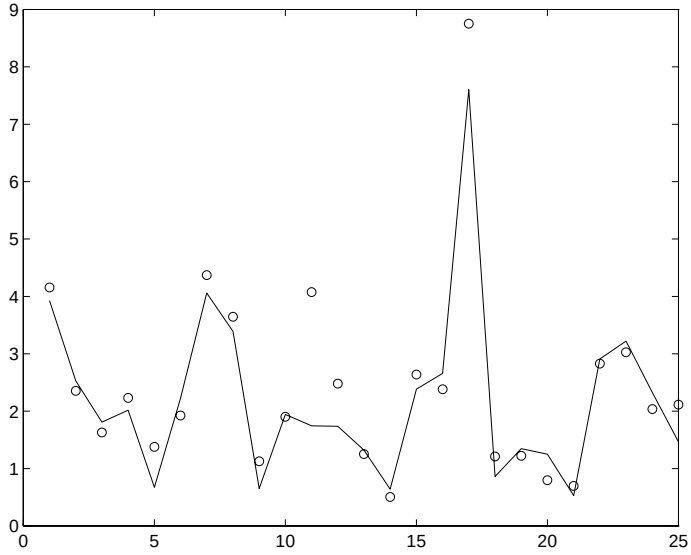
```
fuzout=evalfis(datin,fismat);
trnRMSE=norm(fuzout-datout)/sqrt(length(fuzout))
trnRMSE =
   0.5276
```

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

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

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

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



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

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

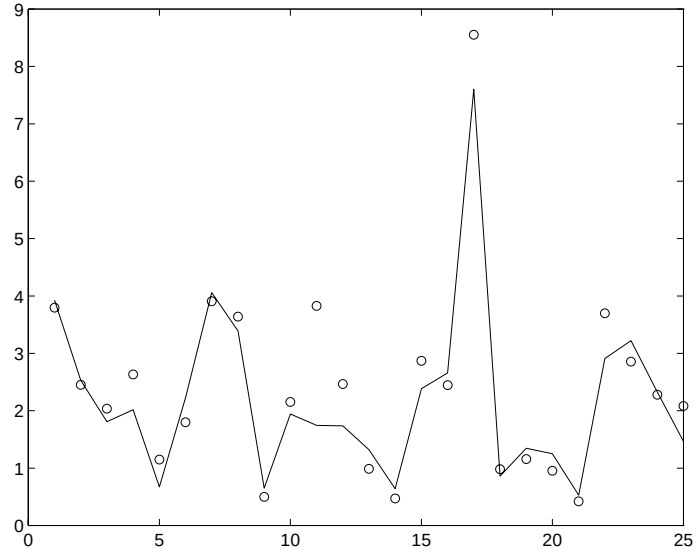
After the training is done, we type

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

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

```
plot(chkdatout)
```

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



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

Overfitting

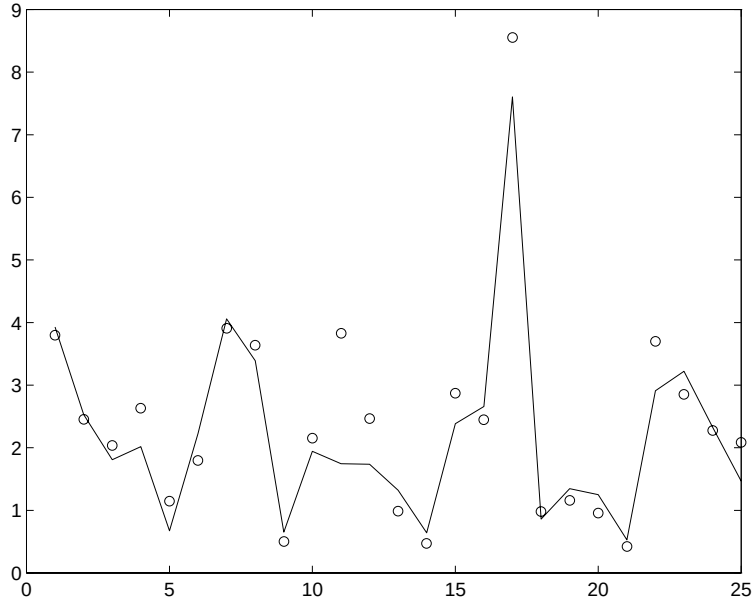
Now let's consider what happens if we carry out a longer (200 epoch) training of this system using `anfis`, including its checking data option.

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

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

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

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



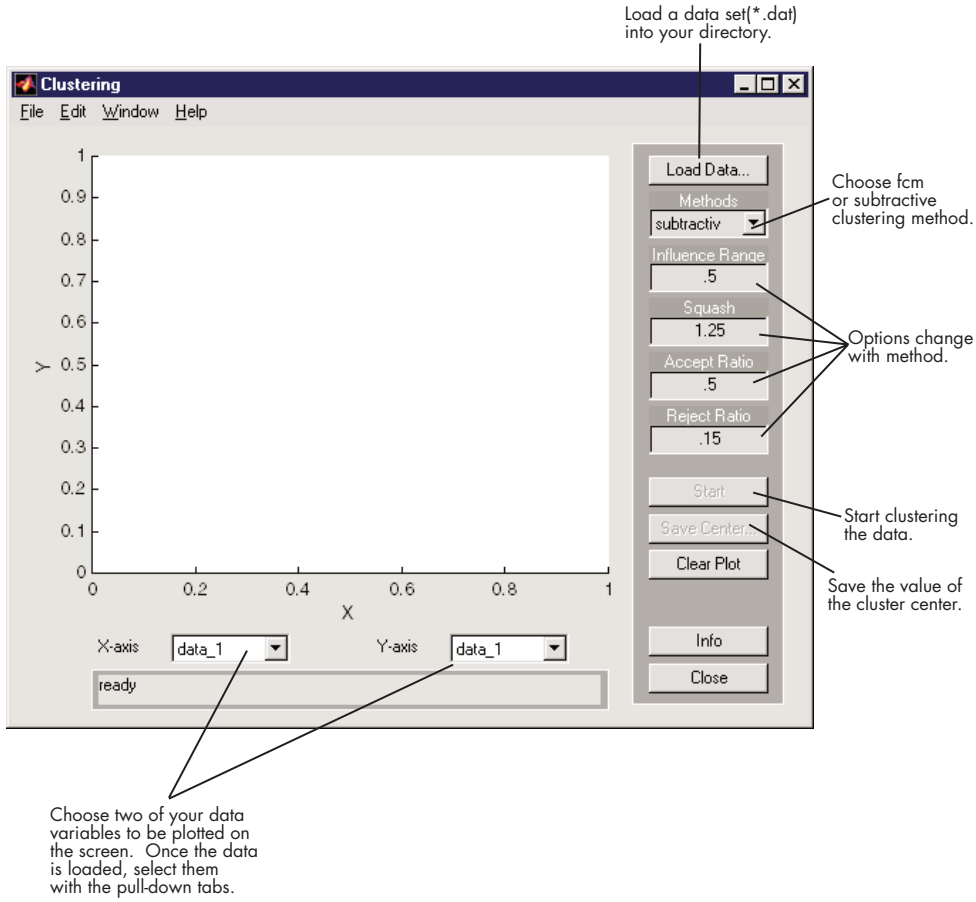
Here we can see that the training error settles at about the 50th epoch point. In fact, the smallest value of the checking data error occurs at epoch 52, after which it increases slightly, even as `anfis` continues to minimize

the error against the training data all the way to epoch 200. Depending on the specified error tolerance, this plot also indicates the model's ability to generalize the test data.

A Clustering GUI Tool

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

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



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

You use the pull-down list under **Methods** to change between `fcm` (fuzzy c-means) and `subtractive` (subtractive clustering). More information on the options can be found in the entries for `fcm`, and `subclust`, respectively.

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