

MATLAB[®]

The Language of Technical Computing

Computation

Visualization

Programming

Building GUIs with MATLAB

Version 5

How to Contact The MathWorks:



508-647-7000 Phone



508-647-7001 Fax



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



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



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

Building GUIs with MATLAB (December 1996)

© COPYRIGHT 1984 - 1996 by The MathWorks, Inc. All Rights Reserved.

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

U.S. GOVERNMENT: If Licensee is acquiring the software on behalf of any unit or agency of the U. S. Government, the following shall apply:

(a) for units of the Department of Defense:

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013.

(b) for any other unit or agency:

NOTICE - Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, the computer software and accompanying documentation, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Clause 52.227-19(c)(2) of the FAR.

Contractor/manufacture is The MathWorks Inc., 24 Prime Park Way, Natick, MA 01760-1500.

MATLAB, SIMULINK, and Handle Graphics are registered trademarks and Real-Time Workshop is a trademark of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History: December 1996 First printing for MATLAB 5

Introduction

1

Introduction	1-2
How to Use This Book	1-2
The Big Picture	1-3
A Quick Start with Guide	1-4
Modifying Properties with the Property Editor	1-4
Adding Buttons with the Control Panel	1-6
Setting Callbacks with the Callback Editor	1-7
Activating the Figure	1-8
The Bottom Line	1-9

GUI Design

2

Introduction	2-2
Design Principles	2-3
Simplicity	2-4
Consistency	2-7
Familiarity	2-7
The Dynamic Interface	2-9
Keep It Simple	2-11
Design Process	2-12
A Complete Example: Precipitation Files	2-15

Introduction	3-2
Handle Graphics and the Property Editor	3-3
The Property Editor	3-3
GUI Layout with Guide	3-9
The Control Panel	3-10
The Property Editor	3-12
The Callback Editor	3-12
The Alignment Tool	3-14
The Menu Editor	3-15
A Complete Example	3-16
Writing Callbacks	3-25
Layout vs. Callbacks	3-25
Callback Tips	3-25
A Complete Example: Temperature Conversion	3-35

Introduction

1-2 Introduction

1-2 How to Use This Book

1-3 The Big Picture

1-4 A Quick Start with Guide

1-4 Modifying Properties with the Property Editor

1-6 Adding Buttons with the Control Panel

1-8 Setting Callbacks with the Callback Editor

1-9 Activating the Figure

1-9 The Bottom Line

Introduction

Good tools make for pleasant work.

This book is about how to make MATLAB-based Graphical User Interface (GUI) tools. It falls naturally into two parts:

- GUI design, or *how to make something that's useful*
- GUI implementation, or *how to make something that works*

The principles of good GUI design are, for the most part, timeless and universal. They apply in MATLAB as much as they apply anywhere else, though we consider them here with respect to specific examples. The guidelines for GUI implementation, on the other hand, tend to be much more specific—many of the implementation details we discuss here weren't possible in MATLAB until recently. Accordingly, in this manual there is a trend from the general to the specific. Part of the second half of the manual is devoted to the use of MATLAB's GUI-building tool, Guide (Graphical User Interface Development Environment). A quick start example of how to use Guide also closes out this introductory chapter.

How to Use This Book

The goal of this book is to address both the design of the GUI, and its rapid and robust implementation in the MATLAB environment using Guide. If you only want to know how to use Guide, read through the rest of this introduction and then go to Chapter 3, "GUI Implementation." If you also want to learn about designing GUIs, read Chapter 2 as well.

Examples Online The M-files behind many of the examples in this manual can be found in `ftp://ftp.mathworks.com/pub/mathworks/toolbox/building-guis/`.

The Big Picture

This manual includes many specific recommendations about details of implementation, design, and so on, but keep in mind the big picture: the user interface you create, whether big or small, simple or extravagant, is a *single entity* with a certain *personality* to help a *specific user* perform a *specific task*. Who is that user? What is that task? What is that personality?

The Big Picture

The user interface you create is a ***single entity*** with a ***certain personality*** to help a ***specific user*** perform a ***specific task***.



To see if your interface is doing a good job, here are two good generic questions:

- 1 Do the users always know where they are?
- 2 Do they always know where to go next?

Keep returning to these questions, keep thinking about the big picture, and the details fall into place much easier.

A Quick Start with Guide

MATLAB is built around a programming language, and as such it's really designed with tool-building in mind. Guide extends MATLAB's support for rapid coding into the realm of building GUIs.

Guide is a set of MATLAB tools designed to make building GUIs easier and faster. Just as writing math in MATLAB is much like writing it on paper, building a GUI with Guide is much like drawing one on paper. As a result, you can lay out a complex graphical tool in minutes. Once your buttons and plots are in place, the Guide Callback Editor lets you set up the MATLAB code that gets executed when a particular button is pressed.

Modifying Properties with the Property Editor

The five tools that together make up Guide are:

- The Property Editor
- The Guide Control Panel
- The Callback Editor
- The Alignment Tool
- The Menu Editor

We consider the top three in this Quick Start introduction. Let's start with the Property Editor. In MATLAB Handle Graphics, every graphical object has a handle and a number of properties. For instance, a figure window has a `Color` property and a `Visible` property, and so on. To view and change these properties from the command line, you need to use the `set` and `get` commands together with the handle for the figure. The Property Editor is designed to let you modify these properties interactively. For instance, plot the curve $t \sin(t)$:

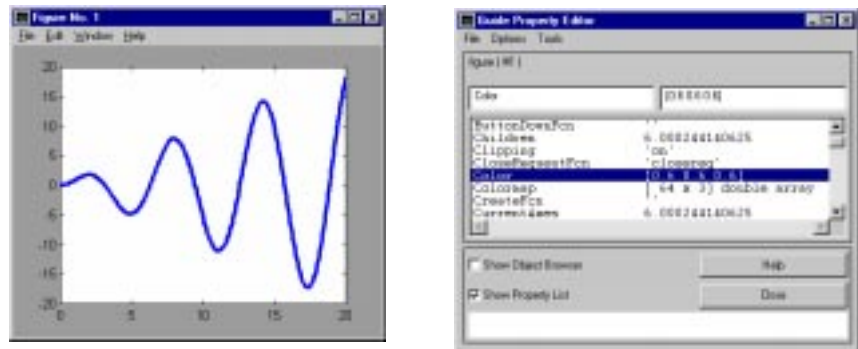
```
>> t = 0:0.1:20;
>> plot(t, t.*sin(t))
>> get(gcf, 'Color')
ans =
0.8000 0.8000 0.8000
```

Here we're using the `get` command to query the figure's `Color` property (recall that `gcf` is a function that returns the handle of the current

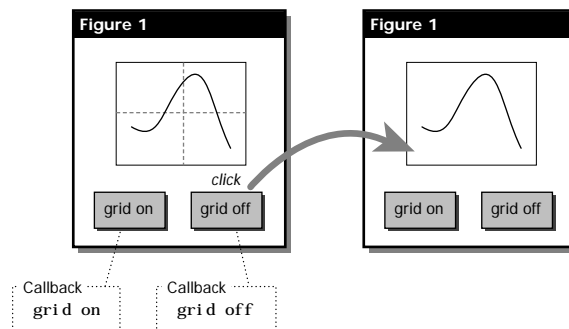
figure). All colors in MATLAB are stored as RGB (red, green, and blue) triplets, where $[1\ 1\ 1]$ is white, and $[0\ 0\ 0]$ is black. We can use the Property Editor to find and modify the same information. Invoke the Property Editor with the command `propedit(gcf)`.

```
>> propedit(gcf)
```

The Property Editor can be used to change the properties of the figure. Click in the **Show Property List** checkbox and scroll down until you see the word **Color** in the first column of the Property List. Click on that line in the listbox, and the current color of the figure quickly appears in the edit field just to the right. Now you can change to something darker, say $[0.6\ 0.6\ 0.6]$. You can use the Property Editor to query or change any property.



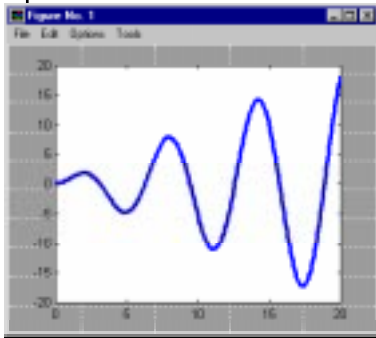
As a simple GUI for getting started, let's add two buttons to this figure: one to turn the axis grid off and one to turn it back on.



Adding Buttons with the Control Panel

Typing `guide` in the MATLAB Command Window brings up the Guide Control Panel and “controls” (or puts into edit mode) the sine wave figure. Here’s what you should see.

Dashed white lines in the figure indicate the figure is controlled by Guide.



The Figure List in the Control Panel also shows which figures are controlled.



The dashed white lines in the controlled figure shown on the left above indicate its controlled state. The Guide Control Panel is divided into three main parts:

- list of Guide Tools (at the top)
- Guide-Controlled Figure List (in the middle)
- New Object Palette (at the bottom)

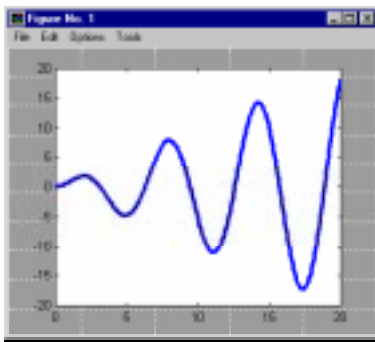
The Guide Tools include the already-mentioned Property Editor, the Callback Editor, the Alignment Tool, and the Menu Editor. In this example, the Property Editor button is selected (“pushed in”) because that tool is currently open on the desktop.

In the middle of the Control Panel, the Guide-Controlled Figure List shows what figures are open and whether or not they are being “controlled” by Guide. When a figure is in a Guide-controlled state, you can move buttons, axes, and other objects inside that figure simply by clicking on them and dragging them—a welcome relief from calculating

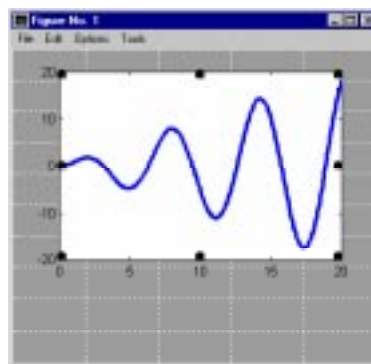
numerical positions in MATLAB code. Once everything is as you like it, then you can “activate” the figure to return it to normal behavior (i.e. no clicking and dragging).

At the bottom of the Control Panel is the New Object Palette. To add an object like a push button, you click on the appropriate region and then click and drag in the Guide-controlled figure. Click once on the “button” field in the Guide Control Panel and you can add a button to the figure by clicking in the controlled figure and dragging until the button is the size you want.

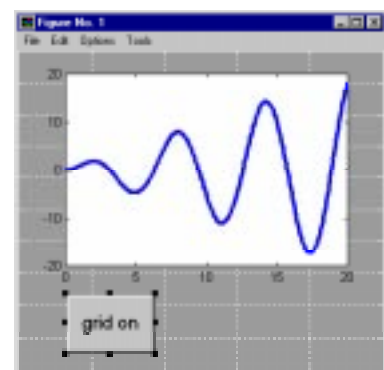
The new button is still selected, so use the Property Editor to set its String property to 'grid on' (it's important to include the single quotes). Now you have a figure that looks like the rightmost figure below.



First the figure is controlled,



then the axes are resized
by clicking and dragging,

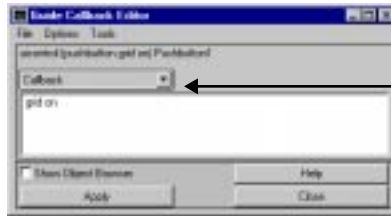


then the button is added
using the Control Panel
and modified with the
Property Editor.

Setting Callbacks with the Callback Editor

We've positioned a button where we want it, but we haven't defined what happens when it gets clicked. This is a job for the Callback Editor. With the button still selected, open up the Callback Editor by clicking the second tool in the Control Panel, select Callback from the pop-up menu,

and enter the callback: `grid on` (No need for quotes when using the Callback Editor).

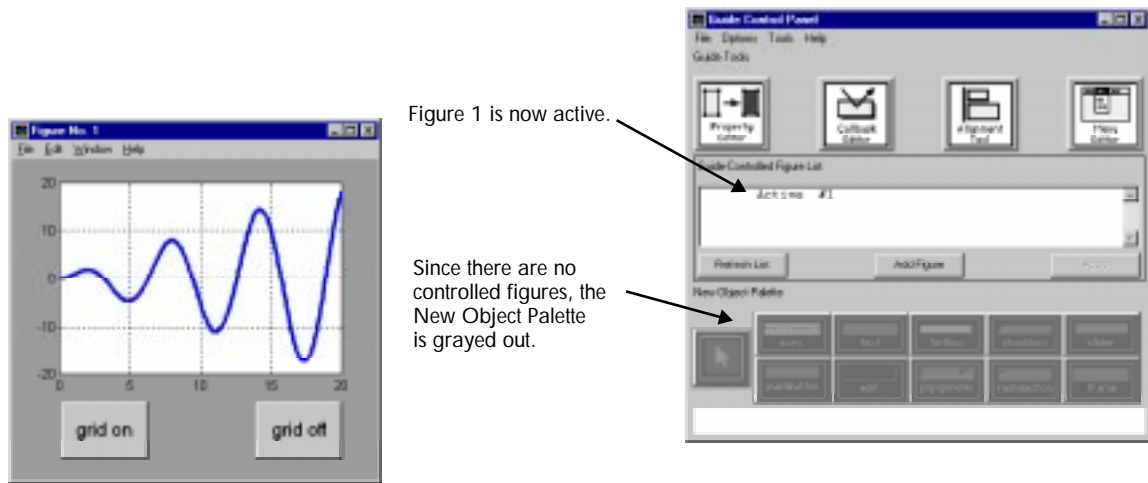


The "grid on" button is selected. Notice the pop-up menu must be set to "Callback."

Once the figure is activated, whenever the button is pushed, MATLAB runs this code, and the grid gets turned on. The code you choose to put in here can be as simple or complex as you like. In this case, the code is quite simple. Before we finish up, we want to add another button similar to the first one. Copy and paste the button shown above using the **Edit** menu in the controlled figure, change both the string and the callback to say `grid off`, and you're ready to run.

Activating the Figure

The last step is to pull the figure out of edit mode. This is known as "activating" the figure. First, click on the Controlled Figure List in the Guide Control Panel where it says Figure No. 1. The Controlled Figure List indicates that Figure 1 is about to be activated. Now press the **Apply** button, and the figure is activated (you are asked if you want to save the figure first). The result is a brand new MATLAB GUI.



The Bottom Line

It will always be hard to design good GUIs, but it shouldn't be so hard to implement them. Guide is designed to take the tedium out of the process. But in addition to being a layout tool for GUIs, Guide gives you powerful tools like the Property Editor that can be used any time you're dealing with MATLAB graphics. The Property Editor alone can be thought of as a debugging tool, a coding assistant (for remembering all those property names and values), and a layout tool. When combined with the Guide Control Panel, the Callback Editor, the Menu Editor and the Alignment Tool, the resulting combination gives you unprecedented command of MATLAB graphics. Good tools make for pleasant work, and Guide makes building good tools easier than ever.

GUI Design

2-2 Introduction

2-3 Design Principles

2-4 Simplicity

2-7 Consistency

2-7 Familiarity

2-9 The Dynamic Interface

2-11 Keep It Simple

2-12 Design Process

2-15 A Complete Example: Precipitation Files

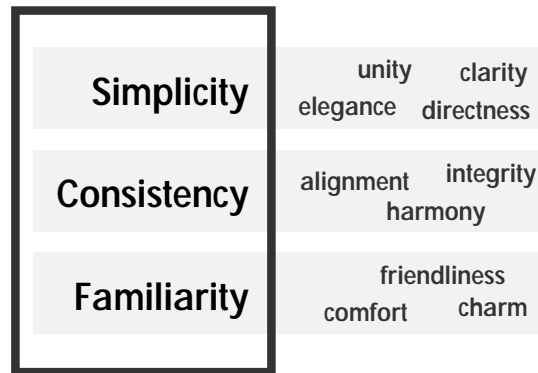
Introduction

This chapter is about the first half of the GUI creation process: designing the GUI. Designing means everything you do before you start to write the code that results in a final working GUI. Since GUI design is not a simple recipe, this chapter is itself divided into two parts: guiding principles for design, and a suggested process to follow for design.

Design Principles

GUI design has been around for a short time, but the universal qualities of good design have remained unchanged by its arrival. This disappoints some people who believe GUI design ought to be a completely unexplored territory. But it can also be extremely comforting to tackle a GUI design problem and find centuries of accumulated wisdom waiting patiently to help you.

Hundreds of books have been written about design, and the best we can expect to do here is emphasize fundamental themes. The good news is that the same ideas keep coming up over and over again. Here is the short list: Simplicity, Consistency, and Familiarity.



Each word is really the center of a natural group, since many words can be used to describe the same basic ideas. But of these concepts, large and small, the undisputed king is this: Simplicity.

To the quantitatively inclined, these ideals may sound too fuzzy. How can we be more specific? With respect to GUI design, two metrics inform everything we do:

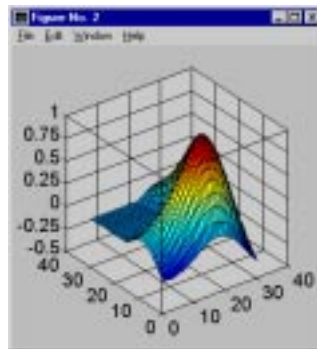
- 1 How long does it take to perform a task the first time?
- 2 How long does it take to perform a task once the interface is familiar?

Each of the themes above bears directly on these two measures.

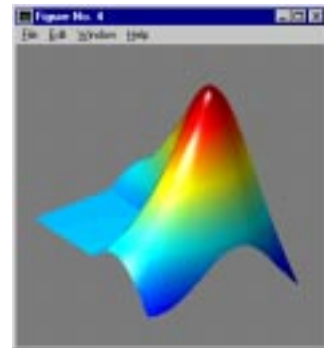
Simplicity

Simplicity in design is our chief goal. A simple GUI has a clean look and a sense of unity. It's very easy to add functionality to the GUI you're building, but if that functionality really doesn't belong, take it out. Avoid screen clutter, and only present users with choices that advance them toward the completion of the task.

Emphasize Form, not Number Clutter obscures valuable information. Since visualization is inherently more qualitative than quantitative, concentrate on the shape and let the labeling vanish.



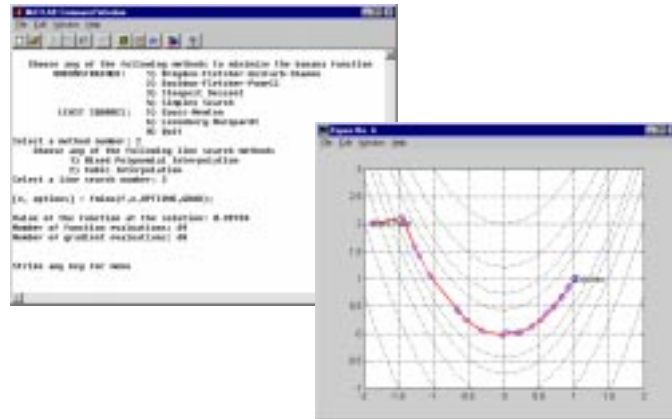
The grid lines on the left don't really add value to the image.



Once you let yourself remove a piece of the GUI that doesn't absolutely need to be there, you may find that you can eliminate a lot of supporting machinery that no longer has any purpose.

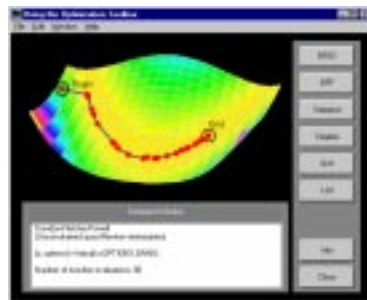
Minimize the Area of Interaction Don't use two figures when one will do. If you're demonstrating input-output relationships, put the input right next to the output.

This Optimization Toolbox demo (bandem.m) shows how different optimization methods find the lowest point in a complex surface.



The original Banana Function demo. You have to shift your attention from the Command Window to the figure and back.

The figure above shows a contour plot of a function we're trying to minimize. We are presented with the starting point for the minimization and the correct endpoint at the very lowest point of the surface. We are given, from the command line, several numbered options to choose from. Type in a number and you see an animation of the minimization as it occurs.



The redesigned Banana Function demo. Now everything is on one figure.

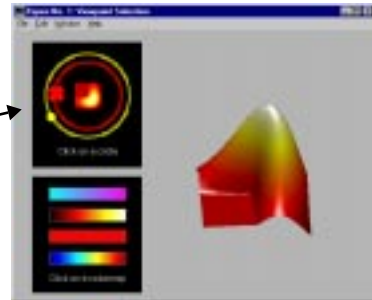
The figure above shows an improved version of the demo. We've unified the presentation into one figure to minimize the area of interaction. We've also replaced entering a number between one and six with clicking on a button. This is another way to minimize the area of interaction. The method labels are right on the buttons. Finally, since the demo is quali-

tative (the goal is to find the lowest point), we can throw away the numbers on the plot and turn up the contrast on the colors, thereby making the graph much more readable.

Use Graphical Input Rather Than Numeric Rather than typing in numbers, let the user enter data by directly “touching” the graphic. Let the graphic index into itself.

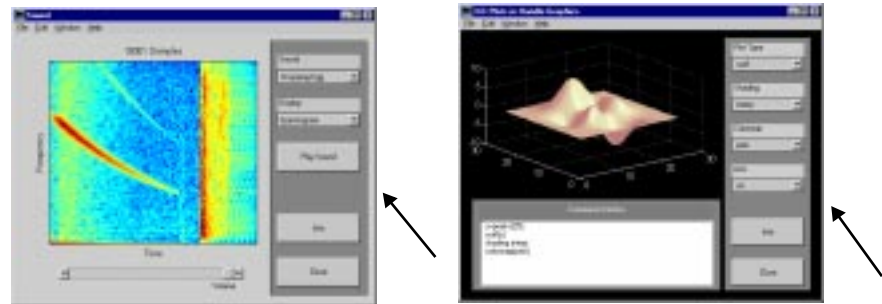
In the example shown below, you can “walk around” the three-dimensional shape shown on the right side of the figure by setting the camera position of the viewpoint. Camera position is given by three Cartesian coordinates that can be hard to determine, but this little GUI sidesteps the problem by letting the user place the camera (represented by an X) on a viewing circle around a small representation of the shape.

In this example you click on the X to change the position of the camera. Click on the dot to change the position of the light source.



Consistency

The further users are from their base of experience, the more likely they are to feel disoriented. Anything you can do to keep the user from feeling confused is extraordinarily important.



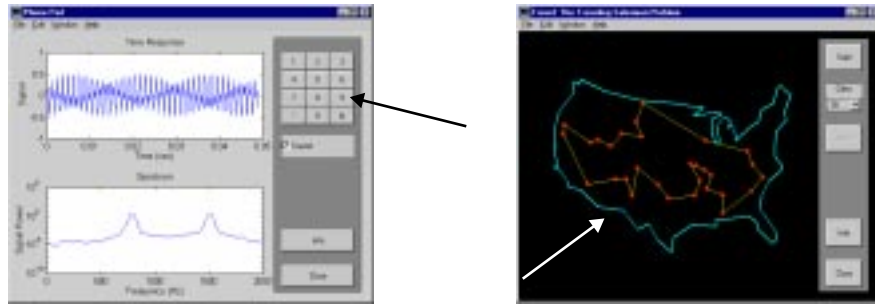
These two demos are about very different things, but both respect the convention of putting controls on the right side grouped inside a frame. Notice the consistent placement of the **Info** and **Close** buttons.

The GUI on the left shows sound analysis capability (filename `xpsound.m`), and the one on the right introduces some graphics programming concepts (filename `graf3d.m`). They are in two very different parts of the MATLAB demo software, but they share (along with a great many other MATLAB demos) a number of basic layout characteristics. This consistency makes it easier to move from one demo to the next by generating a sense of familiarity, which also leads to the next theme.

Familiarity

If the GUI that you create is in some sense familiar to its users, then they can generally learn how to use it more quickly. This is value of basing the GUI on a good metaphor. They might not know how to do a given task, but the metaphor helps them make a good guess.

Use the Familiar to Draw Users into the GUI Even a small connection to the user's real world experience can bridge the gap between the onscreen experience and the data it represents.



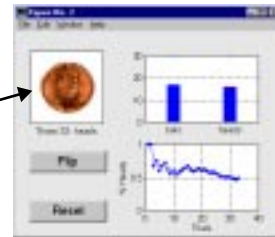
The touch-tone phone pad in the example on the left and the outline of the U.S. in the example on the right both provide some familiar context for the GUI.

These two GUIs are examples of drawing on experience outside of MATLAB to enrich understanding within it. On the left, we're looking at the spectral analysis of the tones generated by a touch-tone phone (filename `phone.m`, from the Signal Processing Toolbox). The upper plot shows the sound waveform, and the lower plot shows the spectral content. To generate the tone, we use a representation of the phone touch-tone keypad itself.

The figure on the right shows a demonstration of the traveling salesman problem (filename `travel.m`), which goes like this: Given a list of cities to be visited, what route between them results in the shortest overall trip? In this demo, the cities are really just random points on the plane.

But by superimposing an outline of the United States, this mathematical curiosity suddenly feels more relevant.

A photograph of a penny in this demo helps communicate the basic nature of this statistical experiment.



The GUI shown above illustrates the MATLAB random number generator by simulating the tossing of a coin. The image of a penny in the upper left part of the figure emphasizes the content of the two plots (a histogram and a history plot). Familiarity draws people into the GUI and makes them feel comfortable.

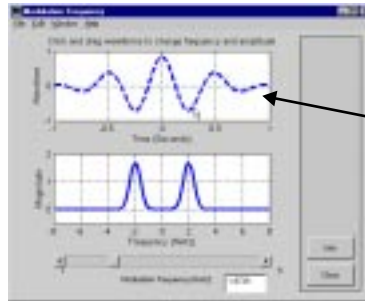
The Dynamic Interface

Much of what we've discussed so far is biased by the static world of graphic design. But the world mediated by the computer is a dynamic one, and there are a few more key words to mention. Where you act using the GUI, your actions should be: Immediate, Continuous, and Reversible.

Immediate	snappy direct
Continuous	smooth physical
Reversible	friendly encouraging

Permit Direct Manipulation Whenever possible, let the user *grab* the data as though it were a solid object.

With respect to immediacy and continuity, calculation time is the most important consideration. If you can make a calculation and display the result instantly, by all means do so. But if calculation time is significant, you may be better off using a button to invoke the action.



The fact that you can “grab” the line in the top part of this GUI and instantly see its Fourier transform in the bottom greatly enhances the GUI’s value.

In the example above (file name `sigdemo2.m` in the Signal Processing Toolbox), the cursor is shown in the upper plot dragging a waveform. The Fourier transform of the signal is instantly updated in the lower plot. The result is an insightful illustration of the relationship between a signal and its Fourier transform.



In this demo, you can change the angle of attack of the airfoil by literally grabbing the wing and rotating it. The corresponding lift shows up as a vector and a call-out on the lift plot.

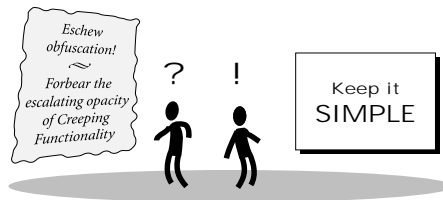
The GUI above shows the lift generated by an airfoil as a function of angle of attack. If you click anywhere on the lift curve, the airfoil on the right side immediately rotates to the selected angle and the lift arrow shows the magnitude of the lift generated. You can also set the angle of attack by clicking on the wing and rotating it to a new angle. This kind of immediacy and continuity promotes quick learning and a high degree of interaction.

Finally, reversibility is most often embodied by the **Undo** menu. Undoing is sometimes difficult to implement, but it’s always appreciated

by users. A well-built undo capability encourages experimentation and a sense of comfort in working with your GUI.

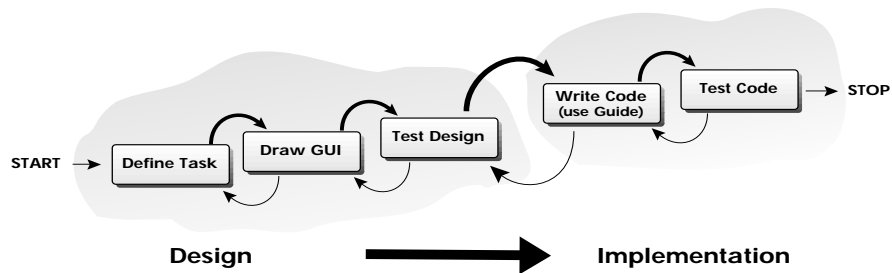
Keep It Simple

Lists of design guidelines grow quickly, and they can eventually obscure the very ideas they were meant to promote. Remember to learn from GUIs that you like, and when it doubt, keep it simple.



Design Process

In this section, we suggest a road map for creating GUIs in MATLAB. Like everything associated with design, this really amounts to some suggestions about how to think, rather than a step-by-step recipe. It's helpful to think about the GUI creation process as breaking into a design phase and an implementation phase. This concept is illustrated in the diagram below.



This is the most important fact presented in this chapter: complete the design of your GUI before you begin to implement your GUI.

Don't Start Coding Until You're Done Designing If you start working on the implementation too soon, you tend to converge much slower on a final design.

Of course the design may change once you start coding, and there may well be design decisions that you can't properly make until you've written some code. Still, take the design part just as far as you can on paper, because you are guaranteed to save yourself a lot of time.

Notice the arrows in this diagram point in both directions—sometimes you move forward and sometimes back. In fact, some have drawn this diagram as a spiral, noting that inevitably your first design needs to go through the whole process multiple times.

Start with the Ideal Don't make compromises too soon. The best final designs grow from an idealized initial vision.

The ideal design is something that exists in your head after you've spent a long time thinking about the task and the user. This design might be expensive in terms of development effort or computing power required. But for whatever reason, it's the best way to approach the task. Think of it as a three-dimensional shape that must now be projected down into the two dimensions of the final (and achievable) implementation. If you start out with the merely achievable, the final result will be half-hearted. If you start with the ideal, it's often surprising what you can make the technology do for you.

Test the GUI on Paper Drawing and testing your GUI on paper keeps you and any testers from over-focusing on the detailed aspects of the GUI.

Early on in the design process, there's no point in playing with font sizes and button colors. What matters is overall behavior and appearance of the GUI. Paper prototyping is a good exercise for keeping things in perspective. The idea is to build your entire interface out of paper and try it out. Since there's no code involved, you play the role of the computer while a cooperative user sitting across from you performs a task. It's an excellent way to see if the GUI does what you (and the user) want it to do.



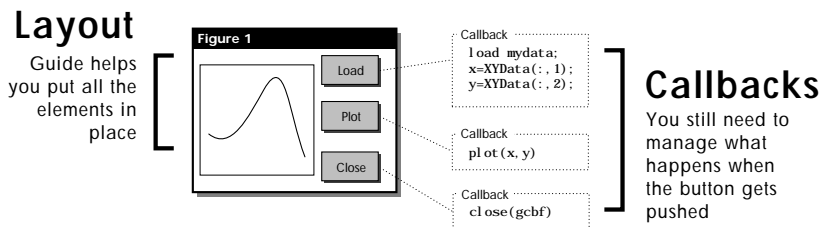
Paper prototyping: these are still video-images from paper prototype usability testing at The MathWorks. Notice the simulated “watch” cursor in the rightmost image.

Paper prototyping tests the thoroughness of your design and can resolve disagreements about tough design questions that might otherwise be intractable.

By the time you're ready to start writing code, you should have a complete diagram or set of diagrams of the GUI (the layout) and an exact description of what functionality is associated with each part of the GUI (the callbacks). Actually, the term "write the code" is somewhat misleading now that Guide is available to make your job easier. Part of the coding process is about building up the GUI with MATLAB components like uicontrols and axes, putting everything in the right place. You can either do this directly with pages of initialization code that look like this.

```
ui control (' Style', ' pushbutton', ...
' String', ' Pl ot', ...
' Cal l back', ' pl ot (x, y)', ...
' Uni ts', ' poi nt', ...
' Posi ti on', [50 65 75 35])
```

or you can use Guide (MATLAB's GUI Development Environment). Guide is described in more detail in Chapter 3, but the basic idea is that it lets you do all your layout graphically.



As shown in the illustration above, even after you've used Guide to get the layout right, you still need to write the MATLAB language callbacks.

A Complete Example: Precipitation Files

In this section we go through the design process from beginning to end, noting each step along the way. One thing that becomes obvious when you start to work with a real example is how much the different tasks on our process checklist spill into each other. In this example, the process breaks into three natural clusters.

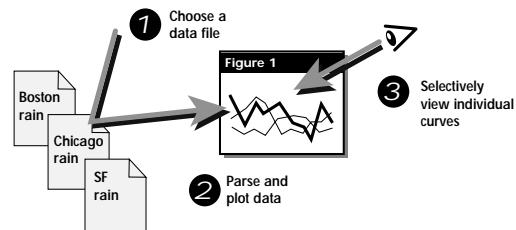
We want to use MATLAB to view weather data that we found on the net. Specifically, these files are monthly precipitation datasets available from the US National Climatic Data Center (<http://www.ncdc.noaa.gov>) as part of the NCDC digital database TD3220.

1. Define the Task—Draw the GUI

The task is still somewhat vague, and surprisingly enough this is often the case in real world examples. You don't always know what you're looking for until you start to draw up some scenarios. Here is an excerpt from the .prp precipitation data files we want to plot:

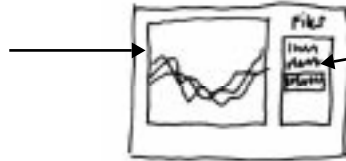
NUMBER	LOCATION	STATE	YEAR	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC	ANNUAL
190770	BOSTON LOGAN INTL AP	MA	1948	511	208	314	262	537	450	453	124	67	484	516	125	4051
190770	BOSTON LOGAN INTL AP	MA	1950	386	381	299	238	155	110	145	314	89	199	617	337	3270
190770	BOSTON LOGAN INTL AP	MA	1951	404	371	441	306	481	431	213	323	200	398	660	469	4697
190770	BOSTON LOGAN INTL AP	MA	1952	431	471	441	441	357	326	52	686	113	161	172	409	4060
190770	BOSTON LOGAN INTL AP	MA	1954	326	337	333	525	1338	278	250	564	831	358	552	540	6232

We have the .prp files for a number of different cities. We want to select the data file for a given city and see a plot of the monthly precipitation figures. Here's simple illustration of what we want to do.



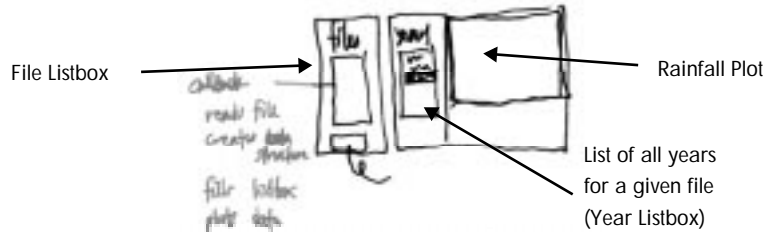
The Task. Select a .prp precipitation file for a given city and interactively view the rainfall figures for each year in that file. Here is sketch number one.

Plot of the yearly rainfall data for a given file
(Rainfall Plot)



List of all *.prp files in the current directory
(File Listbox)

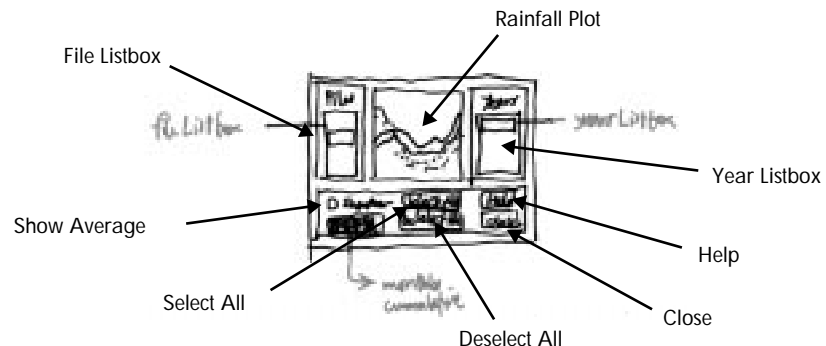
All the .prp files in the current directory are displayed in the listbox on the right. Click on the file listbox and the data for that file are read into a MATLAB variable and plotted in the axes on the left. This is a good start, but we want a way to choose individual years from the list associated with any given location. Here's another sketch that takes that into account.



We've purposely left in some of the scribbled annotations from these sketches so you can see how the design truly proceeded. Here are some observations about what we've got so far:

- The first thing we have to do is select a city file, so it makes sense to keep that on the left, since we're used to reading from left to right.
- For symmetry and separation of the listboxes, let's move the year listbox to the right side (where the file listbox started out in sketch number one).
- Some extra options at the bottom of the figure would be nice. For instance, we almost always want to include a **Close** button and a **Help** button.
- Another nice option would be to plot the average rainfall for the selected city in the Rainfall Plot, along with the ability to turn it off and on with a checkbox.
- We can also add a **Select All Years** button and a **Deselect All Years** button.

Notice how fluid the design is at this point. Move elements all over the place, test them in your mind, try to see what works. By adding the Average Rainfall option, we've expanded on the original task, too. Here's sketch number three, our latest effort. Now we have a reasonable drawing of what the GUI is going to look like. We're not done with the design until we know exactly what each part of the interface will do. Here are the procedures for each of the objects.



File Listbox. Identify the selected file, parse it and load the data into MATLAB, put the year list into the Year Listbox and plot the rainfall data for all years.

Year Listbox. Identify the year selected, and select that line in the Rainfall Plot and deselect all others.

Rainfall Plot. If a line is clicked on, select it, deselect all other lines, then select the appropriate year in the Year Listbox.

Show Average. This is a checkbox that toggles the visibility of the average rainfall line.

Help. Open a dialog box with some help text in it.

Close. Close this figure.

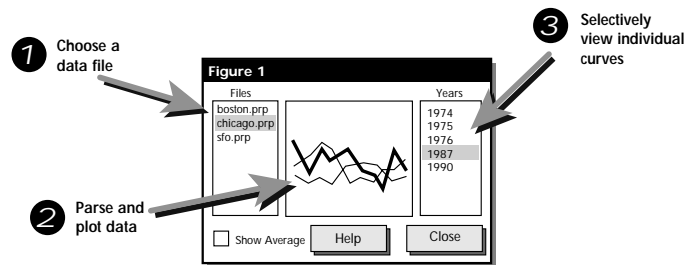
Select All, Deselect All. Either select or deselect all of the curves in the Rainfall Plot. On further consideration, these buttons don't seem very useful. What's the value of selecting every last curve? Let's remove these buttons.

This is our candidate for implementation. Does it meet the criteria set forth in the task? In other words, if we were to test it against our task, would we be happy?

2. Draw the GUI—Test the GUI

Testing the GUI doesn't mean writing code first. It means looking at what you've drawn, putting yourself in the place of the user and seeing if this satisfies the task. This is the stage where we might try building a

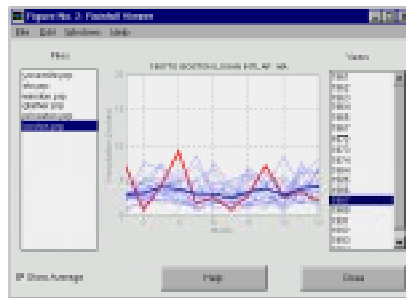
paper prototype of the GUI to see if it holds up well. Shown below is a recap of the task with a cleaned up version of our GUI drawings.



Armed with our knowledge about what each control on the figure does, we can perform a few tests first with ourselves as the subject, then with an office mate or someone down the hall, and finally, if we're serious about doing the best possible job, with real customers.

3. Write the Code—Test the Code

The design is now complete. Since GUI layout and callback writing is the subject of the next chapter, we have reached the end of this example. Here's what rainfall GUI looks like after using Guide to lay out the figure.



This is the GUI, built up with Guide, that represents the final implementation of our design.

GUI Implementation

3-2 Introduction

3-3 Handle Graphics and the Property Editor

3-3 The Property Editor

3-9 GUI Layout with Guide

3-10 The Control Panel

3-12 The Property Editor

3-12 The Callback Editor

3-14 The Alignment Tool

3-15 The Menu Editor

3-16 A Complete Example

3-25 Writing Callbacks

3-25 Layout vs. Callbacks

3-25 Callback Tips

3-35 A Complete Example: Temperature Conversion

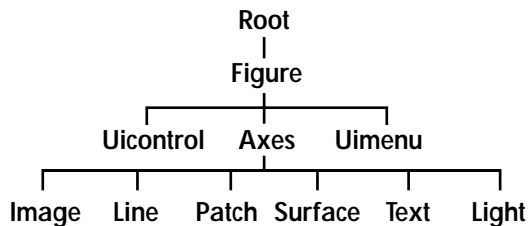
Introduction

This chapter is about the second half of the GUI creation process: implementing the GUI with MATLAB code. We assume here that you already have a good idea of what you want to build, and now you need to know the specifics about to build it.

We start off with a quick introduction to some Handle Graphics® basics and how to use the Property Editor. The rest of the chapter divides into two parts: how to use Guide to lay out a GUI, and how to write the callbacks that do the work behind the scenes.

Handle Graphics and the Property Editor

Anything that appears in a MATLAB figure is an example of Handle Graphics, because every object on the screen has a unique identifier, called a handle, that allows you to go back and modify the object at any time. Generally there's no need to worry about the handles, but they're always there. Shown below is a diagram of the Handle Graphics object hierarchy. These are all the categories of object types in MATLAB.



The Property Editor, one of the Guide tools, is extremely useful for navigating the Handle Graphics hierarchy. If you need to get or set any Handle Graphics property on any object, chances are you can do it most easily with the Property Editor.

The Property Editor

Let's step through an example of using the Property Editor just to get the feel of how it works. Try this:

```
surf(peaks(25))
propedit(gcf)
```

Remember that `gcf`, which stands for “get current figure,” returns the handle of the current figure window. If you're uncertain about functions like `gcf`, `get`, and `set`, read the manual *Using MATLAB Graphics*. The function `peaks(25)` returns some sample data in a 25-by-25 matrix. Now we know that the command

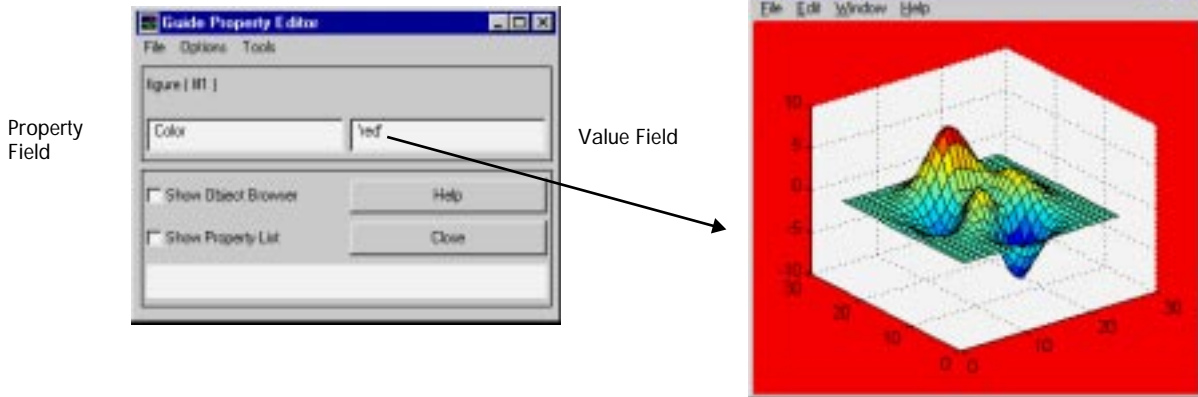
```
set(gcf, 'Color', 'red')
```

will turn the background of the figure red. Since color can also be specified by RGB (red, green, blue) triplets, the command

```
set(gcf, 'Color', [1 0 0])
```

does just as well.

Let's take a look at how we would achieve the same effect using the Property Editor. Below the label figure (#1) in the Property Editor there is an editable text field. This is known as the Property Field. Enter the word `Color` there and press the **Return** key. The RGB value for the default figure background color appears in the editable text field just to the right of it. This is the Value Field. These two text fields let you edit the Property-Value pairs for any object in MATLAB. To make the figure background red, enter the word `'red'` in single quotes in the right-hand, or Value Field as shown in the next figure, just the way you would with the `set` command.

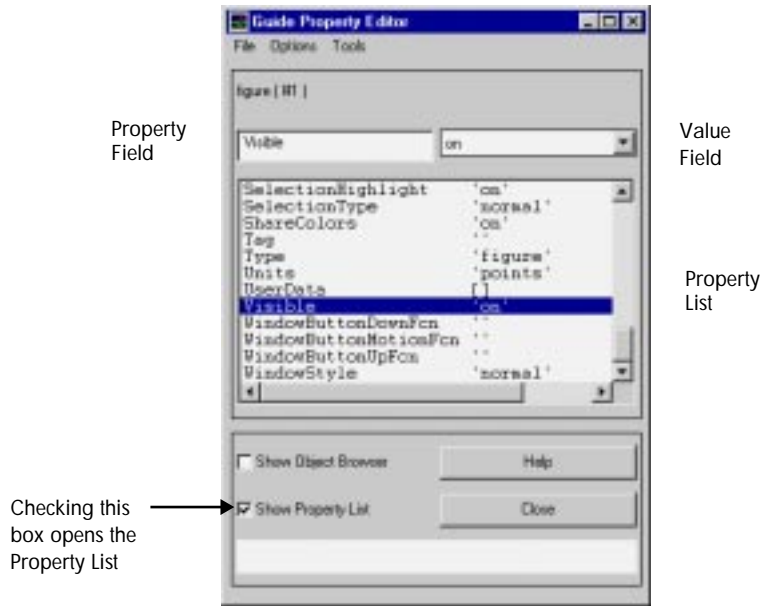


That's the quickest possible introduction to the most versatile of the Guide tools, the Property Editor. Remember, *anything* you can view or modify with the `set` and `get` commands, you can view or modify with the Property Editor. Notice that what you type into the Value Field is the same thing you would type on the command line. So could also enter `[1 0 0]` in the Value Field, and the result would be exactly the same.

Using the Property List

One drawback with the Property Field is that you have to know the name of the property you want to change. Suppose you want to label this figure, but you

don't know the property name that does the right thing. Is it Label or Name? Click on the checkbox marked **Show Property List** and the Property Editor changes shape so that it includes a listbox with all the properties of the current object displayed.



You can now scroll through all of the properties until you find the one that looks right: Name. Here we've entered the name 'Red Peaks', so now that name appears in the title bar of the figure window.

Some properties, such as the Visible property, can be set to one of only a few possibilities. In the case of Visible, since the only two options are yes and no, the Value Field is replaced with a pop-up menu. In other words, the Property Editor is anticipating the fact that you only get to choose between two things.

Using the Object Browser

Just as you typed `propedit(gcf)` to edit the figure, you can also type

```
>> propedit(gca)
```

to edit the current axes. There can only be one Property Editor open at a time, so if you already had it pointing at the current figure, then it just gets redirected at the current axes.

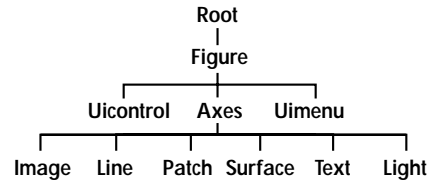
The Property Editor, Callback Editor, and Alignment Tool all include an Object Browser. The Object Browser shows you hierarchically the name and selection state of every object on screen. The Object Browser allows you to expand and collapse the hierarchy of objects starting with the root. A plus sign (+) preceding an object indicates that the object has children and that the hierarchy can be expanded. A minus sign (-) also indicates that an object has children and that the hierarchy can be collapsed. If an object has neither a plus nor minus sign, then the object has no children and so cannot be expanded or collapsed.

A sample Object Browser is shown below. The left column of the Object Browser lists the object type and the right column lists the tag.

Object Browser

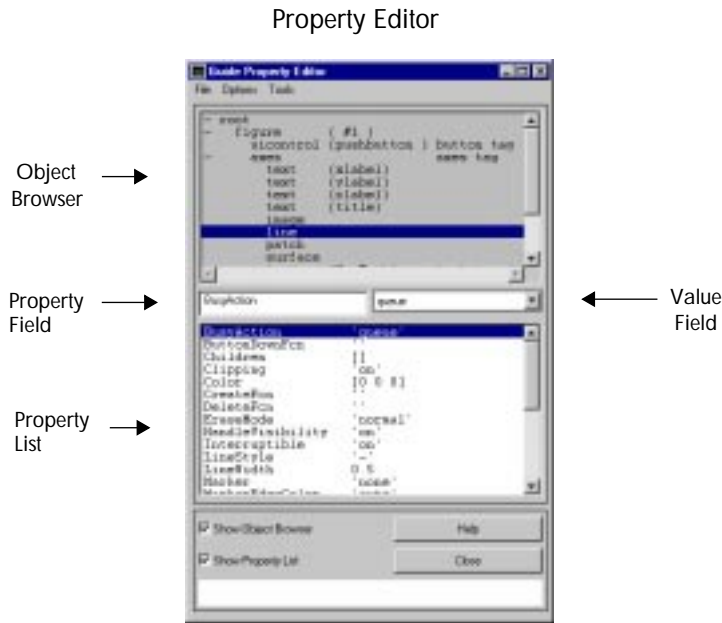


Checking this box opens the Object Browser



Notice the correspondence between the Handle Graphics hierarchy and the indentation level on the Object Browser

You can use the Property Editor with both the Object Browser and the Property List visible.



Secrets for the Power User

The Property Editor is the Swiss Army knife of MATLAB graphics. Here are some tips for maximum efficiency usage.

- **Multiple selection.** Multiple objects, even multiple objects of widely varying types, can be queried and modified together. For instance, if you use the Object Browser to select a line and a listbox, only the properties in common are displayed. You can set the `Visible` property to `off` and both will disappear.
- **Command completion.** If a figure has been selected, you can type the letters “col” in the property field (and press **Return**), and the Property Editor is smart enough to know that `Color` is the only possible completion. This saves huge amounts of typing.
- **Truncation.** The property field ignores anything after the first space. So if you type into the beginning of the text field, press space and return, the old

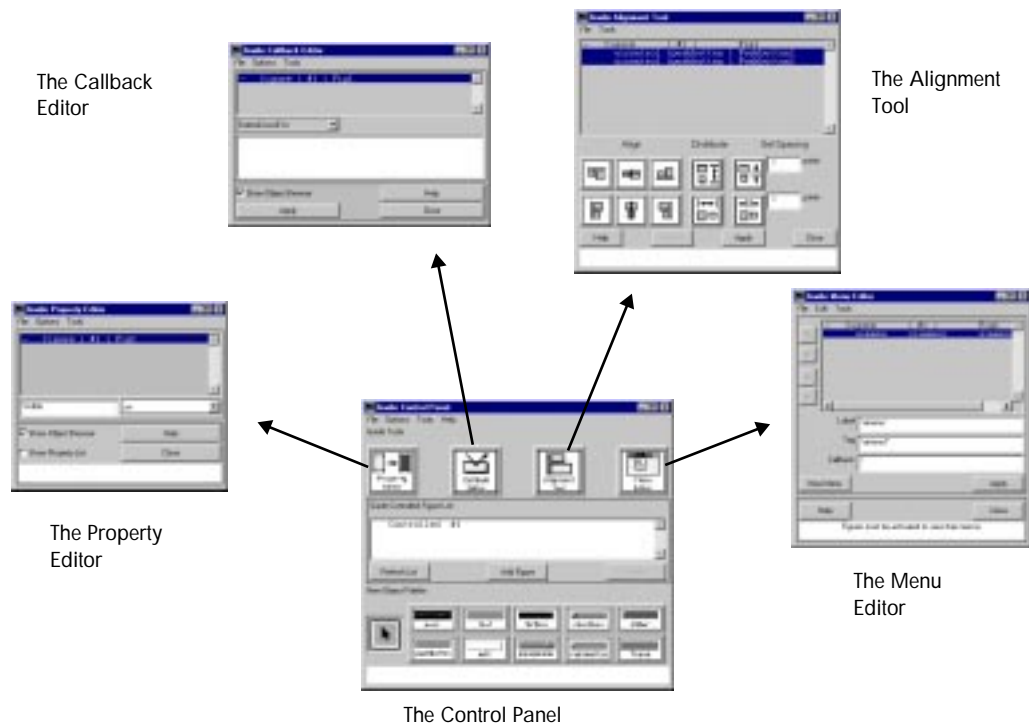
text is ignored. This helps with “hands on keyboard” editing, because it mean you don’t have to select the old text with the mouse and then erase before typing in the new value.

GUI Layout with Guide

Guide simplifies the creation and manipulation of Handle Graphics objects. This means that:

- It simplifies access to the properties of Handle Graphics objects.
- It lets you place and arrange GUI elements on a figure by clicking and dragging with the mouse.

Specifically, Guide consists of five MATLAB tools that offer a streamlined approach to working with MATLAB figure windows.



The Guide tools are the Control Panel, the Property Editor, the Callback Editor, the Alignment Tool, and the Menu Editor. Each tool performs a distinct

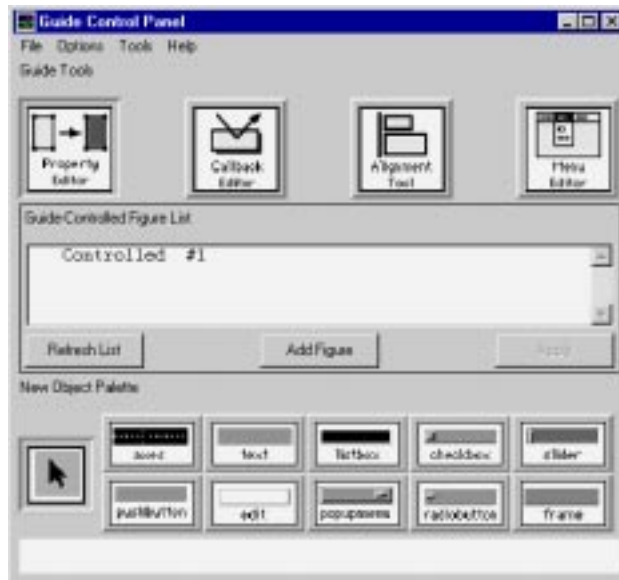
task and also is aware of and interacts with other tools. You can open each Guide tool by entering one of these commands at the MATLAB prompt.

- `guide` (for the Control Panel)
- `propedit` (for the Property Editor)
- `cbedit` (for the Callback Editor)
- `align` (for the Alignment Tool)
- `menuedit` (for the Menu Editor)

The Control Panel

The Control Panel is divided into three parts.

- The Guide Tool List, for launching individual tool
- The Guide-Controlled Figure List, for controlling and activating figures.
- The New Object Palette, for adding axes and uicontrols to a figure.



The Control Panel

Buttons for launching the Guide tools

Guide-Controlled Figure List for activating and controlling figures

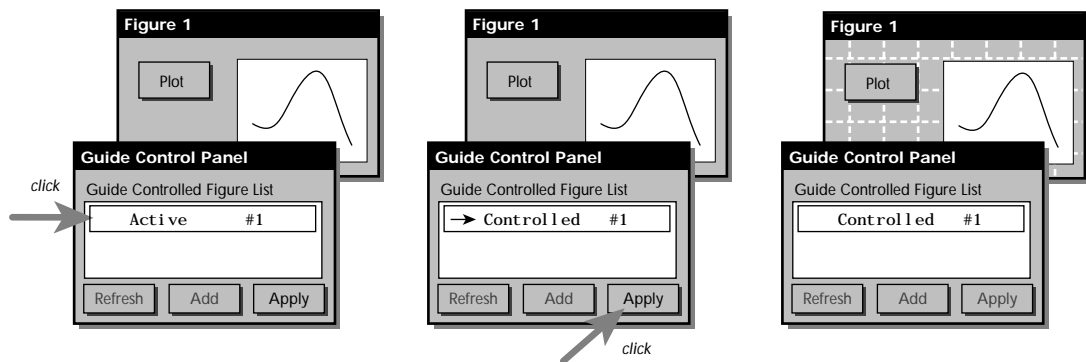
New Object Palette for adding axes and uicontrols

There are nine different kinds of uicontrols that can be added:

- Push buttons
- Radio buttons
- Checkboxes
- UI editable text
- UI static text
- Sliders
- Frames
- Listboxes
- Pop-up menus

Controlling and Activating the Figure

Within the context of Guide, all MATLAB figures are either controlled or activated. The “normal” state of a figure is the activated state. In other words, everything is working just as expected: push buttons push, and pop-up menus pop. But if you want to relocate your push button from one side of the figure to the other, you need to put the figure into an editable, or controlled state. A figure is automatically controlled by the Control Panel whenever you type `guide` from the command line.



To take control of a figure with Guide, click on the appropriate line of the Guide-Controlled Figure List...

... then click on the **Apply** button (alternatively, you can double-click on the listbox).

Now the figure is listed as controlled by Guide. The **Apply** button is disabled.

To take control of an existing figure, click on that figure's name in the Guide-Controlled Figure List, and then click on the **Apply** button. White grid lines provide a visual clue to remind you that a figure is currently controlled. A controlled figure means you can do the following.

- Select uicontrols and axes by clicking on them
- Move uicontrols and axes by clicking and dragging them
- Create and delete uicontrols and axes

You can even control more than one figure at a time, if desired. When you are finished making changes to the GUI, click once on the appropriate line of the Guide-Controlled Figure List in the Control Panel. The Figure List acknowledges that you are requesting that a figure be activated. All you need to do is click on the **Apply** button and the figure is activated.

All Guide tools are useful with both active and controlled figures. This means you can set the background color on a figure regardless of whether it's currently controlled by Guide.

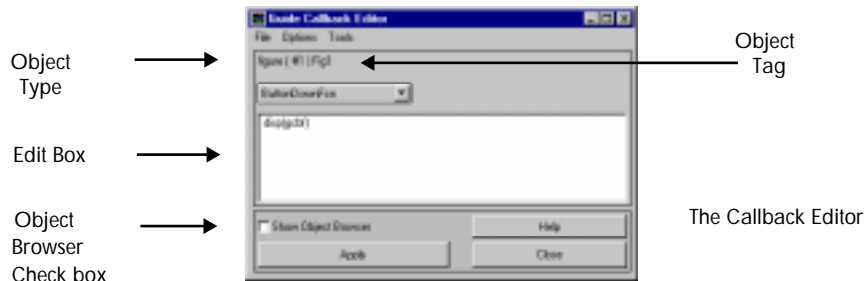
The Property Editor

For a full discussion of the Property Editor turn to the previous section "Handle Graphics and the Property Editor."

The Callback Editor

The Callback Editor lets you modify the callbacks of selected objects. You can change multiple lines of code at once using the Edit Box. The Edit Box also makes it easier to enter Handle Graphics code because it allows you to omit nested quotes and enter code on multiple lines. You can initialize the Callback

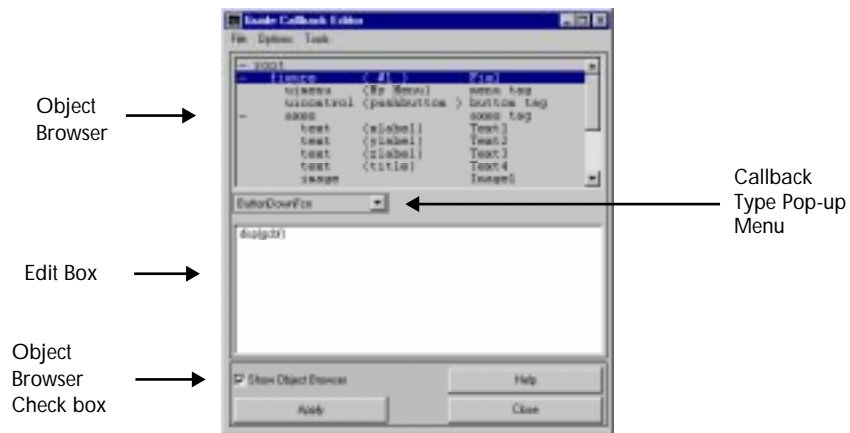
Editor by entering `cbedit t` from the MATLAB prompt, or by clicking on the appropriate button in the Control Panel.



You can use the Callback Editor to make changes to any figure. For example, you might want to change the `ButtonDownFcn` for a figure display the current figure handle.

Begin by selecting the `Show Object Browser` check box to display a list of objects. Select the property you want to modify, in this case the `ButtonDownFcn`.

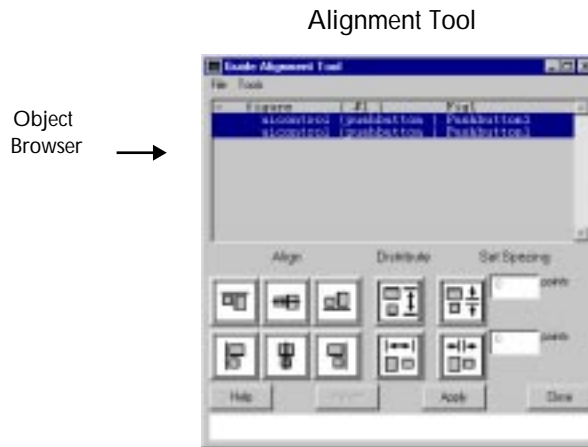
Callback Editor



Select the **Apply** push button when you have finished making changes. If you click in the figure now, it displays the current figure in the MATLAB workspace.

The Alignment Tool

The Alignment Tool allows you to align selected objects using a collection of vertical and horizontal alignment or distribution push buttons. In the past, you needed to know and specify the exact location of any object that you wanted to move using complicated Handle Graphics commands. Now you can move objects relative to each other or distribute the gaps between objects in a single step.



You can open the Alignment Tool by entering `align` from the MATLAB prompt or by clicking on the appropriate button on the Control Panel.

Alignment Tool Features

- The vertical alignment or distribution push buttons do the following: align by top, align by middle, align by bottom, distribute top-to-bottom, and set explicit spacing.
- The horizontal alignment or distribution push buttons do the following: align by left, align by middle, align by right, distribute left-to-right, and set explicit spacing.

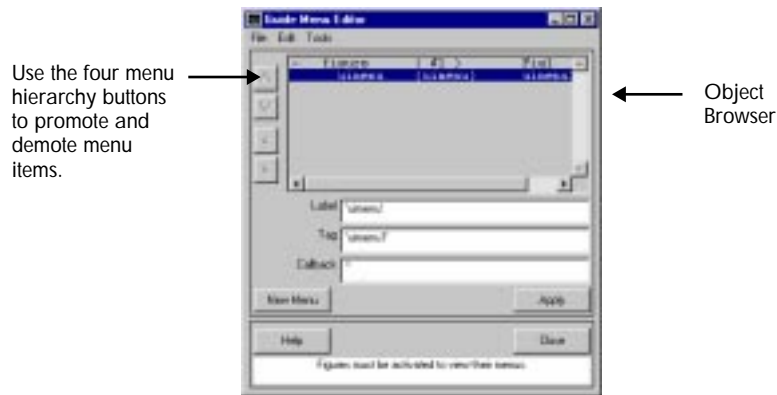
Alignment and distribution always leave the affected objects inside the imaginary box that originally bounded them.

The Menu Editor

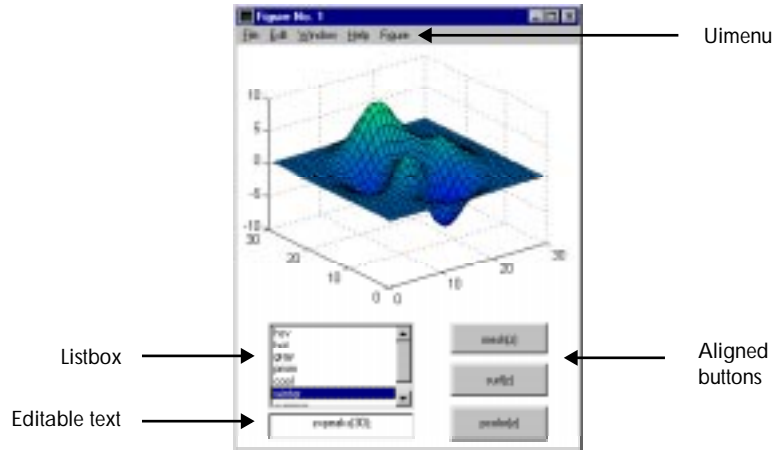
Use the Menu Editor to add and edit the user-created pull-down menus associated with a figure. The menus you create with the Menu Editor are in addition to the basic menus that normally belong to a figure (**File, Edit, Window, Help**). As far as Guide is concerned, those four system-level menus are invisible and unreachable. It is possible to hide the four system-level menus by setting the `MenuBar` property of the figure to none.

The Menu Editor does not display the new menu until the figure has been activated.

Menu Editor



A Complete Example

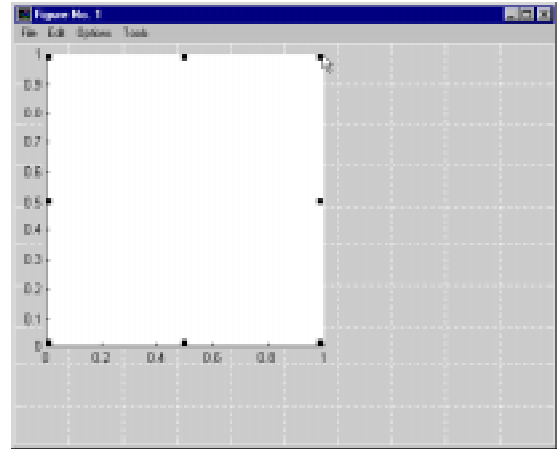


Here is a GUI that lets you modify the colormap and background color of a figure window and to change what it being plotted. Let's use Guide to build it from scratch. This example touches on all the major aspects of Guide.

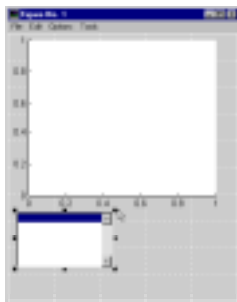
Type `gui de` at the MATLAB command line. A new figure and the Guide Control Panel will appear. Press the axes button on the Control Panel and select an area on the controlled figure.



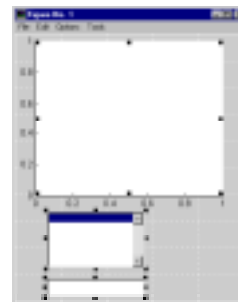
Click here to create a new axes.



After repositioning the axes and resizing the figure, add a listbox uicontrol by selecting from the Edit/Add Object/listbox menu on the Controlled Figure or by selecting it on the Control Panel and then dropping it on the figure. Add an editable text box below it as shown here.



First add a listbox...

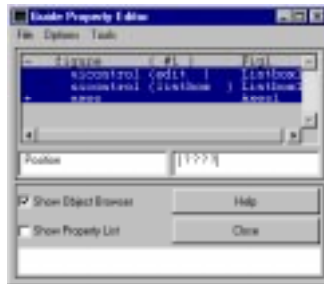


... then add an editable text box.

Next, left-align the listbox, the edit box, and the axes. You can do this with the Alignment Tool, but first open the Property Editor. You can open the Property Editor one of three ways:

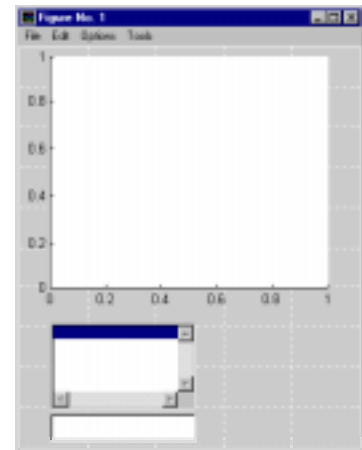
- Select all three objects, then double-click on one of the selected objects
- Click on the **Property Editor** button on the Control Panel
- Use the **Tools** menu on the controlled figure.

Once open, type the letters `pos` in the Property field on the left side of the Editor and the press the **Tab** key. Command completion puts the word `Position` in the Property Field and `[? ? ?]` in the Value Field.



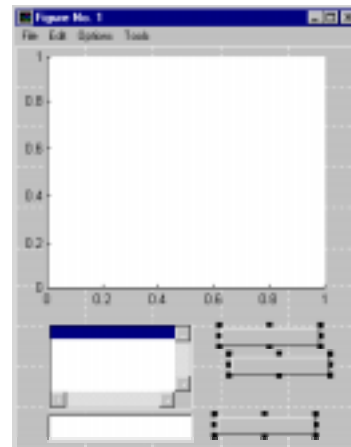
Question marks are used when multiple objects are selected and the given position values for the objects are not equal. You now have the option of doing one of two things to align the selected objects to the left. You can either replace the first question mark in the Property Editor Value Field with a number or you can use the Alignment Tool.

To use the Alignment Tool, open it from the Control Panel or from the **Tools** menu on the controlled figure. Once open, click on the **Align Left** button on the Alignment Tool. Press the **Apply** button and your GUI should now look like the figure on the right.

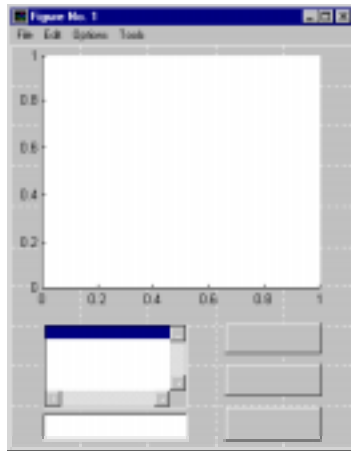


The three objects are aligned to the left side of the bounding box that encloses them.

Next, add the push buttons to the figure. Add a push button to the figure and copy it twice. Next, select all three buttons and look at their positions in the Property Editor. We want their widths and heights to be equal, so enter [? ? 75 25] in the value field for the Property Editor.



Next, we want to align the buttons to the right side of the axes and distribute the space between the buttons. When finished, the controlled figure should look like this.



All the GUI elements are in place, so you're ready to activate the figure.

We're now done laying out the figure, so activate it and then continue to use the Callback Editor and Menu Editor. To activate the figure, click on the Controlled Figure List listbox on the Control Panel to get figure #1 to indicate --> Active, then press **Apply**. Make sure to save the figure when prompted.

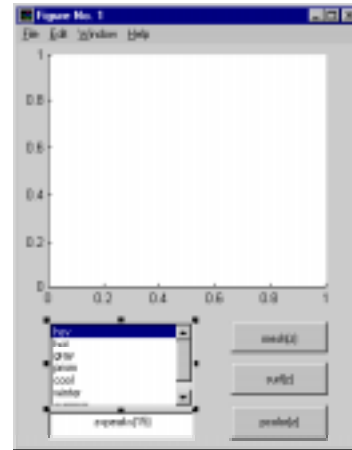
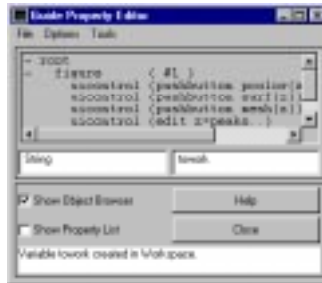
Go back to the Property Editor. The root is now listed in the Object Browser, along with any other figures that may exist. This is because the Guide tools are no longer looking at just the list of controlled figures; all graphics objects are visible to them now.

Our goal is to put labels on the three push buttons and the edit box and then to add the string to the listbox. Use the Property Editor to put the strings `mesh(z)`, `surf(z)`, and `pcolor(z)` on the buttons. To add a multiline string to the listbox, first create a cell array in the base workspace. You can do this by entering

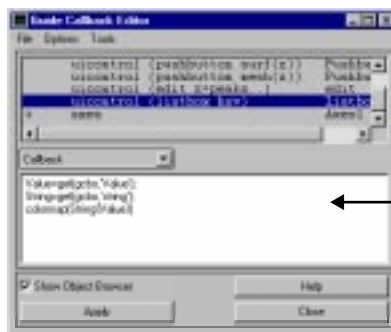
```
>> maps = {'hsv'; 'hot'; 'gray'; 'prism'; 'cool'; 'winter'; 'summer'};
```

at the MATLAB command line. This is a list of just a few of the colormaps that are available. Next we need to select the `String` property for the listbox in the Property Editor. For the value, enter the string `maps` *without quotes around it*. This retrieves the variable `maps` from the workspace. When you apply this value, the edit field shows the string `towork`, which is now a variable in the

workspace equivalent to the variable `maps.towork` is used whenever the value for a property has more than one line. The listbox now contains the correct string.



Now use the Callback Editor to assign the callbacks to the push buttons and listbox. The Property Editor would also work, but since our callbacks are more than one line, it's easier to use the Callback Editor. First, select the listbox in the Object Browser and then select the Callback property.



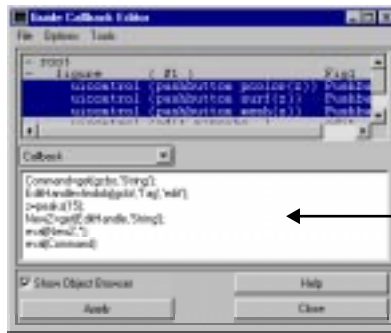
The callback:

```
Value = get(gcbo, 'Value');
String = get(gcbo, 'String');
colormap(String(Value))
```

Enter the callback as shown above. The first line of the callback gets the value for the current callback object, which will be the listbox. The second line of the callback gets the listbox's string. It should be noted that the string returned by

this get command is a cell array because it was entered into the listbox as a cell array. Finally, set the colormap of the figure to be the current selection in the listbox.

Now we want to set the callback for all three push buttons. As before, select the three push buttons, then select the Callback property and then enter the string as shown below.



The callback:

```
CommandString = get(gcbo, 'String');
EditHandle = findobj(gcf, 'Tag', 'edit');
ZString = get(EditHandle, 'String');
eval(ZString);
eval(CommandString);
```

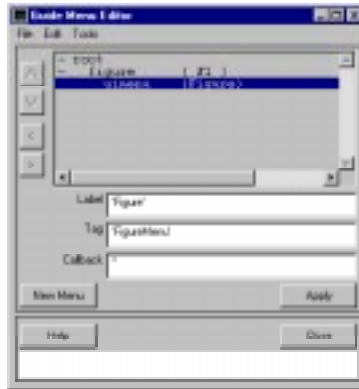
This callback does the following:

- 1 Get the plotting command to be executed from the string of the push button.
- 2 Find the handle to the edit box.
- 3 Get the string from the edit box.
- 4 Evaluate the string for the edit box.
- 5 Evaluate the plotting command.

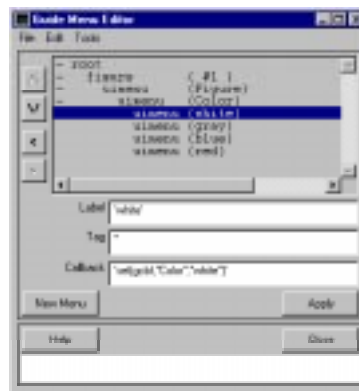
The GUI should now be functioning. Before we test it, go to the **File** menu on the figure and select **Save...** You can now test to see that the GUI works.

The last step is to add some menus to the figure to allow control over the color of the figure with the Menu Editor. Open the Menu Editor, select the figure in the Object Browser and click on **New Menu**. Enter 'Options' for the Label for

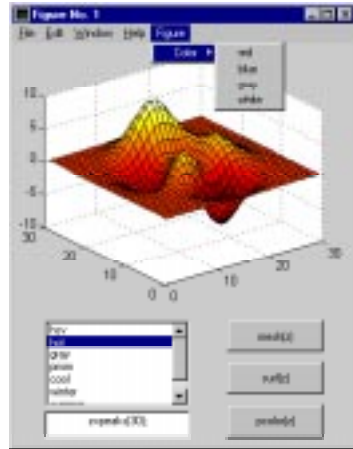
the menu and then click on **New Menu** again to place a sub-menu below the **Options** menu.



Label this menu **Col or** and then click on **New Menu** a third time. This new menu has a Label of 'whi te' and a callback of 'set(gcbf, '' Col or'' , '' whi te'')'.



You can now add additional menus as desired. Again save the figure and test out your GUI. The final result should be similar to the figure below.



Writing Callbacks

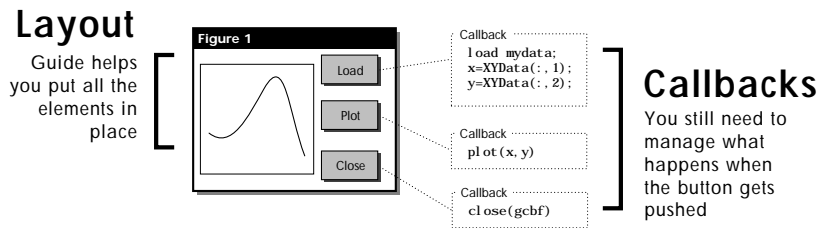
Layout vs. Callbacks

Just as building GUIs divides into design and implementation, so does the implementation part divide into screen layout (typically using Guide) and callback handling. This is the difference between how the screen looks versus what it does when you interact with it.

Happily, using Guide means that you'll never again have to write code that looks like this:

```
ui control('Style','pushbutton', ...
  'String','Plot', ...
  'Callback','plot(x,y)', ...
  'Units','point', ...
  'Position',[50 65 75 35])
```

In other words, it's possible that Guide is all you need to build your GUI, but chances are you need to write some more support code. Using Guide, you keep all your layout changes in the figure file. This is nice because it separates the graphic design part of building the GUI from the callback coding part.



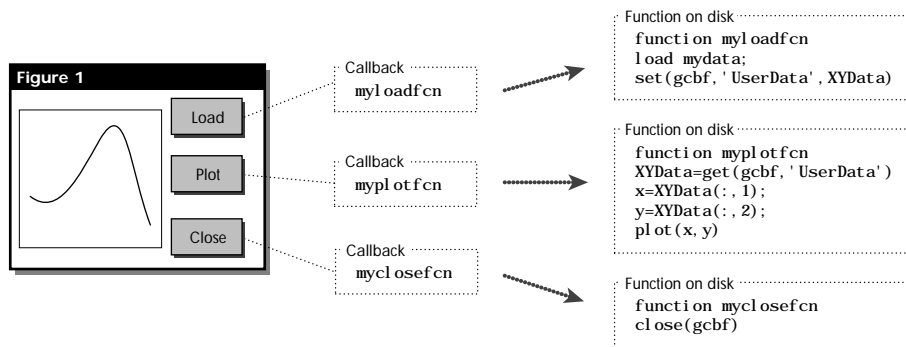
Callback Tips

As part of your GUI design, you typically produce a good description of how the GUI should behave. Even if you have experience coding in MATLAB, it can still be tricky to get the callbacks working robustly and efficiently. In this section, we walk through a collection of callback coding tips that you might use in your GUIs.

Function Callbacks Are Faster Callbacks that consist of long strings to evaluate execute more slowly than callbacks that are short function names.

The code that you put in the callback string for any uicontrol gets evaluated when that uicontrol is activated. You can put a long string of commands directly into the callback. For instance, suppose you have a mini-GUI that loads some data (`mydata.mat`) into the workspace, then plots it, as shown in the previous figure.

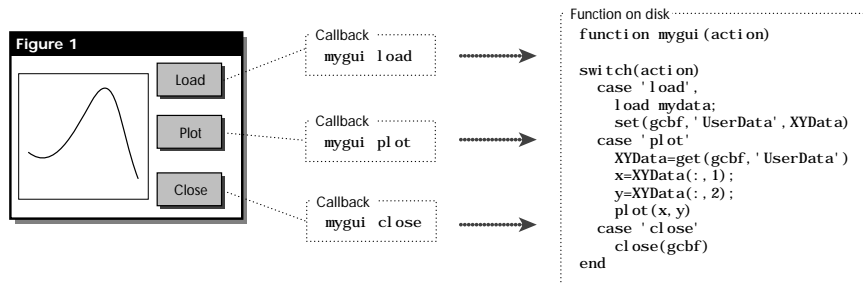
For speed, robustness, and ease of coding, you should make all of your callbacks functions. The code is *faster* because functions, once loaded and “compiled” by the interpreter, run much faster, whereas the evaluated strings get parsed anew every time. The code is *easier to read and write* because you don’t have to double up quotes (though the Callback Editor alleviates the need to double up quotes). The code is *more robust* because it executes in the function’s workspace, as opposed to the callback string, which gets executed directly in the base workspace where it can collide with other data. In this example, `XYData` gets deposited in the workspace every time you push the button. Here is the same application with functional callbacks.



Notice we have to modify the callbacks to handle the workspace issue. In the function’s workspace, we can’t simply say `plot(x,y)`, because the function doesn’t know what `x` and `y` are. One way to deal with this is to store the data in the `UserData` a property of the figure for later retrieval, as shown here.

Switchyard Programming Prevents Function Proliferation Send all the callbacks of your GUI to one single function with a `switch` statement so there won't be a separate function for every callback.

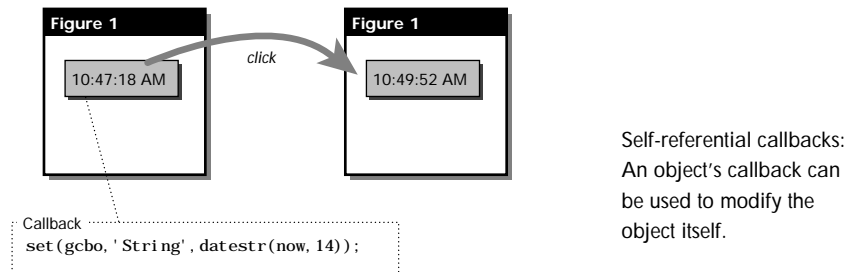
Separate functions for each callback can lead to a lot of files to support one GUI. For reasons of maintainability and ease-of-coding, we're better off putting all those into a single function as shown below. This approach, known as *switchyard programming*, keeps all the code associated with figure callbacks in one centralized place. If your GUI uses one figure file (and its associated MAT-file) and one callback function like the one below, you need to keep track of only three files per GUI.



Use `gcbo` and `gcbf` for Self-Referential Callbacks If your callback needs to know about or modify its own state or the state of its parent figure, the functions `gcbo` and `gcbf` are short and convenient.

There are several special handles that make writing callbacks much easier. They all involve grabbing the current handle of a certain type: `gcbf` (Get Callback Figure), `gca` (Get Current Axes), `gcbo` (Get Callback Object). These are all

simple functions that use the Handle Graphics hierarchy to grab the correct handle. Try this:



This button has a callback that makes its own string equal to the time of day. You can think of the `gcbo` function as being loosely translated to “me.” So in this example the button is saying: “when I get clicked, make my label equal to the time of day.” The `datestr` function, operating on the output of the `now` command, has many formatting options, and option 14 corresponds to HH:MM:SS PM.

Stateless Programming with `findobj` If you use the `findobj` function to locate an object whenever you need to modify it, you can often eliminate the need for redundant state variables.

The command `findobj` returns the handle of an object that matches the description (a list of property name/property value pairs) you give it. Since `findobj` allows you to find any object in the entire Handle Graphics hierarchy quickly and easily, you can find the handle associated with a line, for example, only when you need it, rather than having to remember it from the moment you created it. Here's the old way of dealing with handles:

```
lineHandle = plot(rand(10, 1))
% Remember that handle!
... [lots of other code here] ...
set(lineHandle, 'LineWidth', 4, 'Color', 'red')
```

If the handle gets lost, cleared, or goes out of scope between the first statement and the last, then you either cause an error or you must regenerate the plot from scratch. There are several strategies we might use to keep from losing

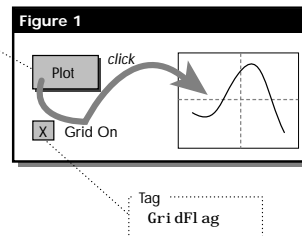
track of `lineHndl`, but they all have drawbacks. You can use a global variable (though this is discouraged) or you can store the handle in `UserData` somewhere in the figure. All of these mean keeping track of redundant state. In addition, the handle may change if the figure is stored and reloaded, and if you use saved figure files, then you don't have access to the object handles at creation time.

Here's how the same example works with `findobj`.

```
plot(rand(10, 1), 'Tag', 'My Line')
% Don't even bother with the handle yet; just give it a Tag
... [lots of other code here] ...
lineHandle = findobj('Tag', 'My Line');
set(lineHandle, 'LineWidth', 4, 'Color', 'red')
```

You can use `findobj` to search for any property name/property value combination, but the `Tag` property is generally the easiest one to use. You can think of the `Tag` field as an invariant handle that you get to designate. In the example below, the button's callback needs to determine the state of the checkbox before it can finish its work. This is a good example of when `findobj` is useful.

```
Callback
gridHndl=findobj(gcf, ...
    'Tag', 'GridFlag');
plot(x,y);
if get(gridHndl, 'Value'),
    grid on;
else
    grid off;
end
```

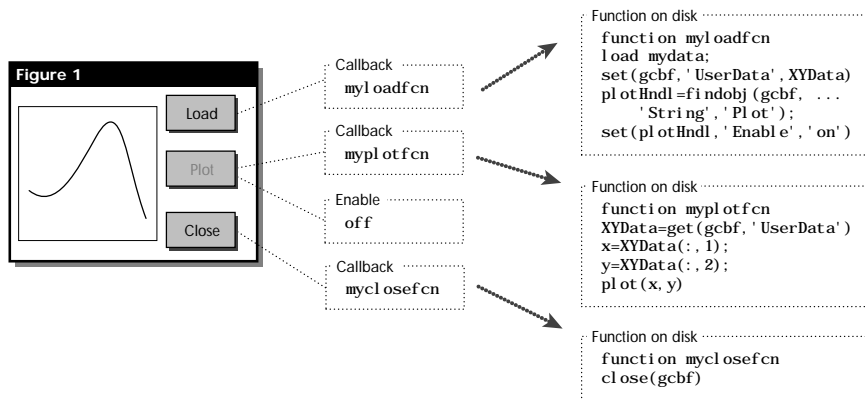


Here the button is modifying its action based on a setting somewhere else in the figure (the grid check box)

One word of warning concerning `findobj`: since `findobj` searches through the object hierarchy every time it's called, you may find it too slow for some purposes. In cases where speed is essential (such as `WindowButtonMotion` loops, which are discussed later in this section), you may want to store handles in the figure's `UserData` or in global variables.

Avoid Potential Errors by Disabling Uicontrols If a callback is likely to cause an error if invoked at a certain time, set the `Enable` property on the calling object to `off`.

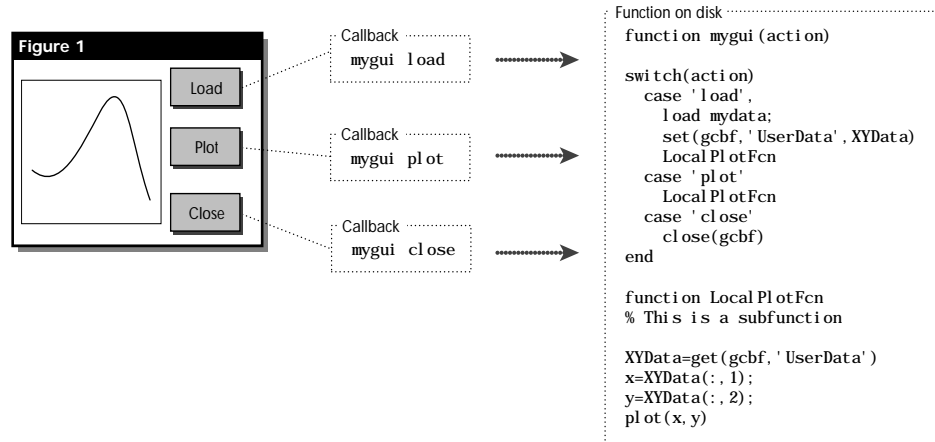
In the example below, the top button loads the data, and the middle button plots it. If the middle button is clicked on first, an error results, because the data will not yet be in the `UserData`. Avoid this problem by disabling the **Plot** button initially (set its `Enable` property to `off`), and then using the **Load** button's callback to re-enable it.



Use Subfunctions Inside the Switchyard Callback Another good way to keep the callback function clean is to route common functionality into a subfunction in the same file.

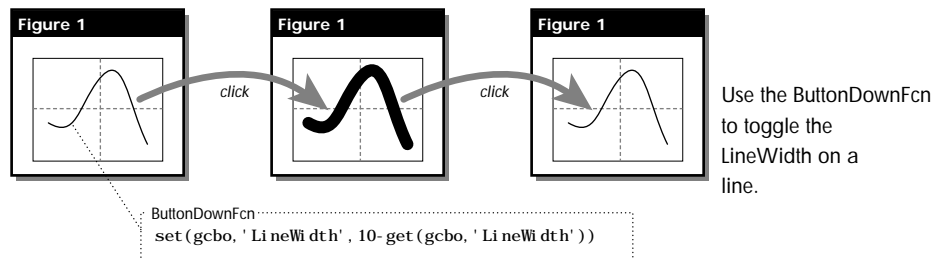
In this example, we've decided we want to plot at two different times, both when the **Plot** button is pushed, and when the data is loaded by the **Load** button. Rather than duplicate the code, we just create a new function in the same file called `Local PlotFcn`. This function is then executed in each button's

callback. As the complexity of your callback function rises, this kind of thing becomes very valuable.



Don't Forget About the ButtonDownFcn Everything from a line to a figure has a `ButtonDownFcn`. Use these and other special callback functions to give your GUI more leverage.

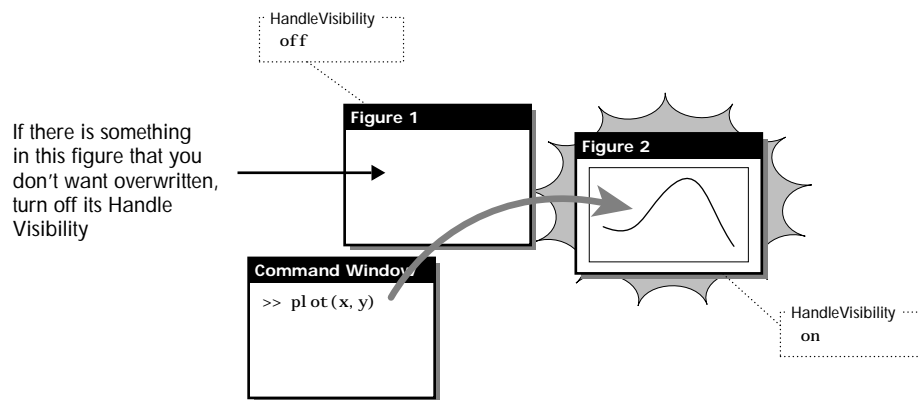
Callbacks from `uicontrols` are not the only way to run code from a GUI. Making appropriate `ButtonDownFcns` for lines, text, and axes in your GUI is one of the most powerful ways to make it interactive. It also alleviates the need for lots and lots of `uicontrols`.



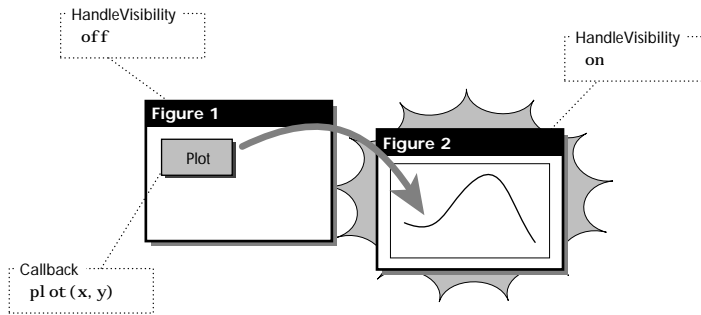
The figure has a number of event functions specific to it, including the `ResizeFcn` and the `KeyPressFcn`. Use the Callback Editor to browse around and see what callback functions are associated with what objects.

Prevent Overplotting with Handle Visibility The `HandleVisibility` property lets you hide your GUI from the MATLAB command line so it can't be tampered with.

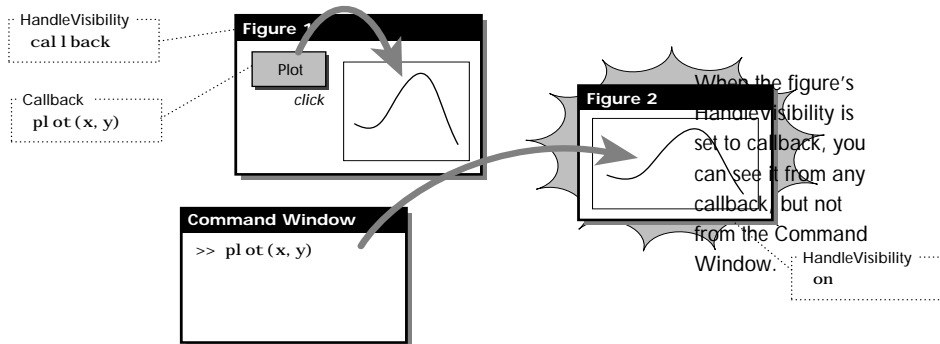
The `HandleVisibility` property (with value settings `on`, `off`, and `callback`) has been added so that regular plotting commands act as though an object doesn't exist.



Plotting commands issued from callbacks aren't aware of their own figures if the figure's `HandleVisibility` is off.



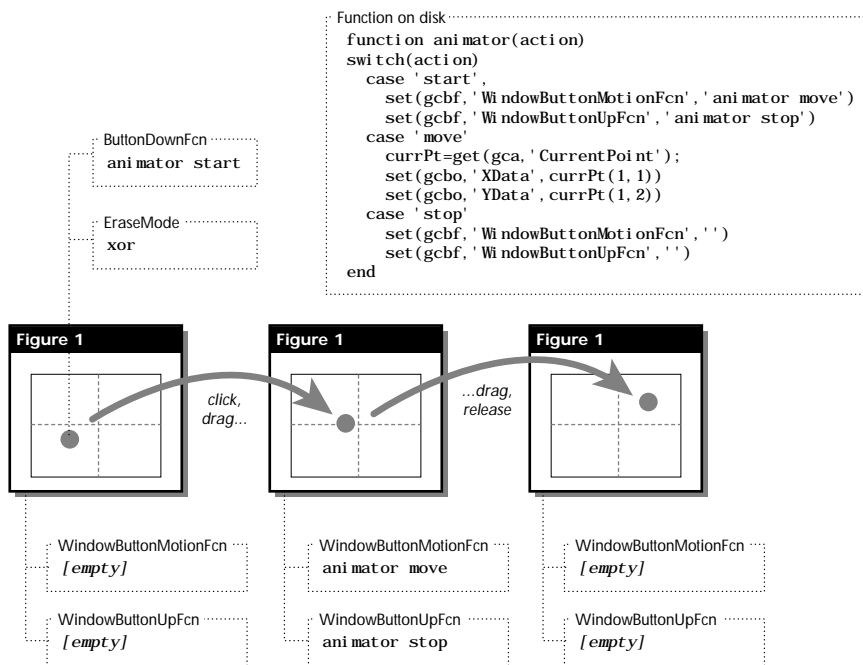
This can be a problem when you're working on a GUI, since the GUI needs to see itself, so a third value, `callback`, exists. Plotting commands issued from the Command Window, for example, won't overwrite the figure when `HandleVisibility` is `callback`. But any commands initiated by the callbacks of the figure itself see the figure.



If you set the `HandleVisibility` of a figure to `off` or `callback`, you can no longer see it from the command line. This means you can't close it with the `close` command or plot into it with the `plot` command. How do you close a hidden figure? There is an option built into the `close` function that lets you close everything, whether hidden or not: `close hidden`.

Animate with WindowButtonMotionFcn To animate a response *as the mouse is being dragged*, use a combination of the callback functions `WindowButtonDownFcn`, `WindowButtonMotionFcn`, and `WindowButtonUpFcn`.

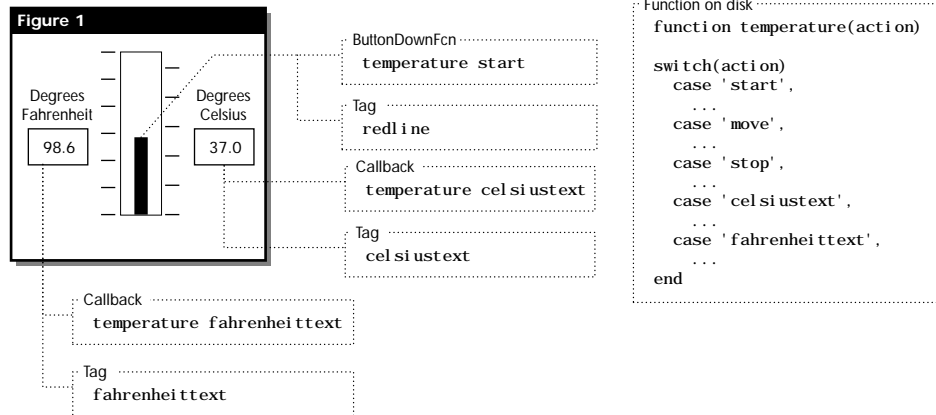
One of the niftier things to do with a MATLAB GUI is animate a calculation on the fly. To do this, you need a function that gets called while the mouse is moving across the screen. This gets a little tricky, but the diagram below explains the situation. Let's say we want to click, drag, and release a dot on a figure window. When we click down on the dot (a line with a single point, to be specific), we need to pick up the coordinates of the mouse and set the position of the dot accordingly.



We can do exactly this with the figure property `WindowButtonMotionFcn`. But the motion function can only be active *when the mouse button is down*.

A Complete Example: Temperature Conversion

Let's consider a GUI that converts the temperature from Celsius to Fahrenheit, and vice versa. We assume that we've already gone through the careful design process from the last chapter, and now we want to build the callbacks into the GUI. This is what a drawing of our design looks like:

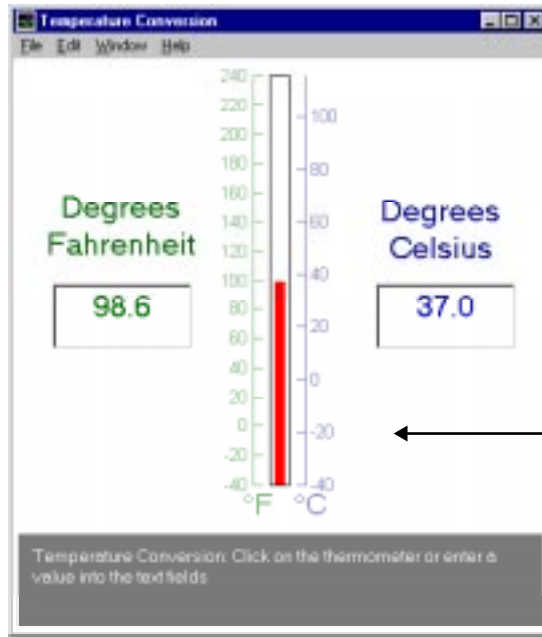


Here's how it works. You can enter a temperature in degrees Fahrenheit in the editable text field on the left (Callback = `celsius fahrenheit text`), or a temperature in degrees Celsius in the editable text field on the right (Callback = `celsius celsius text`). In either case, the callback function takes the temperature in one set of units, converts it to the other using the equation

$$T_{\text{celsius}} = (5/9) (T_{\text{fahrenheit}} - 32)$$

then sets the length of the red line and the other text field. In addition, if you click on the red line in the middle axes, you animate the red line in the thermometer and update the text fields.

After working with Guide, we end up with the following screen and save it to the figure file `temperaturefig.m`.



This center region is actually three different axes—the one in the middle has the animation `ButtonDownFcn`.

And here's what the code looks like that makes the GUI work.

```

function temperature(action)
% This is the callback function for the temperature conversion function

switch action

case 'start'
    set(gcf,'WindowButtonMotionFcn','temperature move')
    set(gcf,'WindowButtonUpFcn','temperature stop')
    temperature move

case 'move'
    currentPoint = get(gca,'CurrentPoint');
    newY = currentPoint(3);
    % Limit the endpoints of the motion and set the pointer
    fah = min(240,max(-40,newY));
    cels = (fah-32)*(5/9);

    % Set the text strings properly
    LocalSetDisplay(fah,cels)

case 'stop'
    set(gcf,'WindowButtonMotionFcn','')
    set(gcf,'WindowButtonUpFcn','')

case 'fahrenheit'
    fah = eval(get(gcbo,'String'));
    % Limit the endpoints of the motion and set the pointer
    fah = min(240,max(-40,fah));
    cels = (fah-32)*(5/9);
    LocalSetDisplay(fah,cels)

case 'celsius'
    cels = eval(get(gcbo,'String'));
    % Limit the endpoints of the motion and set the pointer
    cels = min(120,max(-40,cels));
    fah = cels*(9/5)+32;
    LocalSetDisplay(fah,cels)

end

function LocalSetDisplay(fah,cels)
% Set the text strings properly

pointerHandle = findobj(gcf,'Tag','redline');
set(pointerHandle,'YData',[-40 fah])
fahInndl = findobj(gcf,'Tag','fahrenheittext');
set(fahInndl,'String',sprintf('%3.1f',fah))
celsInndl = findobj(gcf,'Tag','celsius text');
set(celsInndl,'String',sprintf('%3.1f',cels))

```

The start, move, and stop calls are for animating the red line

Here's one of the calls to the subfunction

This is the subfunction for filling the Celsius and Fahrenheit text fields

Reference

align

Purpose Align uicontrols and axes.

Syntax

```
align
align(HandleList)
align(HandleList, HorizontalAlignment, VerticalAlignment)
Positions =
    align(HandleList, HorizontalAlignment, VerticalAlignment)
Positions =
    align(CurPositions, HorizontalAlignment, VerticalAlignment)
```

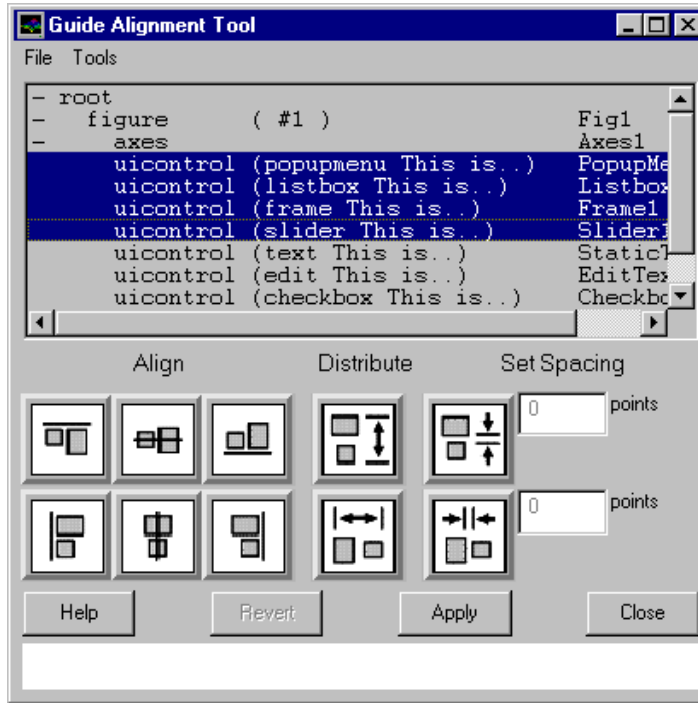
Description `align` brings up an alignment tool to interactively align uicontrols and axes. `align(HandleList)` aligns the handles specified in `HandleList`. Calling the tool as `align(HandleList, HorizontalAlignment, VerticalAlignment)` automatically aligns the objects in the handle list without opening the alignment tool. Adding a left hand argument to this calling syntax causes the updated positions of the objects to be returned while the position of the objects on the figure does not change. Finally, calling the alignment tool with `Positions=align(CurPositions, HorizontalAlignment, VerticalAlignment)` returns the updated position matrix from the initial position matrix without opening the alignment tool.

Possible values for *HorizontalAlignment* are: `None`, `Left`, `Center`, `Right`, `Distribute`, and `Fixed`.

Possible values for *VerticalAlignment* are: `None`, `Top`, `Middle`, `Bottom`, `Distribute`, and `Fixed`. All alignment options will align the objects within the bounding box that encloses the objects.

`Distribute` and `Fixed` will align objects to the bottom left of the bounding box. `Distribute` evenly distributes the objects while `Fixed` distributes the objects with a fixed distance (in points) between them. If `Fixed` is used for *HorizontalAlignment* or *VerticalAlignment*, then the distance must be passed in as an extra argument: `align(HandleList,`

Horizontal Alignment, Distance, Vertical Alignment) or `align`
(*HandleList, Horizontal Alignment, Vertical Alignment, Distance*).



The top of the Alignment Tool contains an Object Browser that lists all uicontrols and axes that can be aligned. Underneath the Object Browser are the alignment controls. Objects can be aligned to each other in both horizontal and vertical directions. The alignment takes place in the bounding box containing all of the objects. Objects can also be distributed with respect to each other such that the spacing between each object is equal. Finally, objects can have a set spacing between them. The spacing option starts with the object in the lower left corner and works its way up and right. This is the only type of alignment that does not pay attention to the bounding box that contains the objects. There must be more than one object selected in order for the alignment tool to work.

See Also

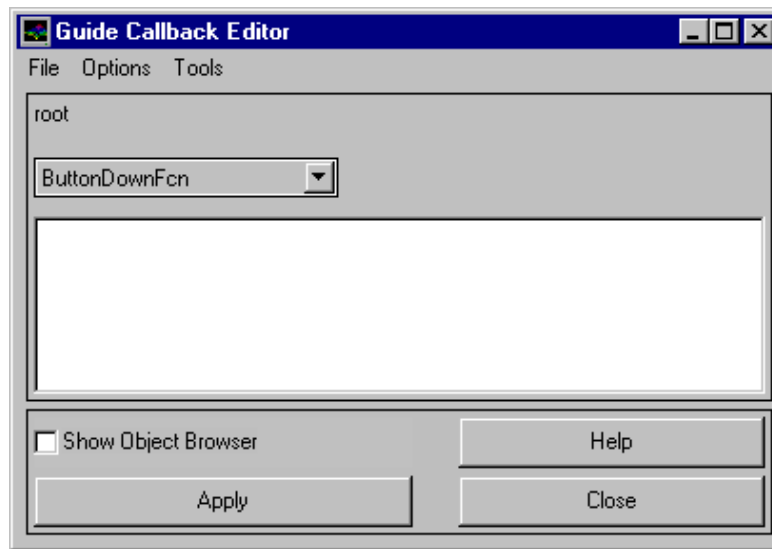
`guide`, `cbedit`, `menuedit`, `propedit`, Object Browser, Controlled Figure

cbedit

Purpose Edit callback.

Syntax `cbedit`
`cbedit (HandleList)`

Description `cbedit` interactively edits callback strings for a selected object. `cbedit (HandleList)` edits the properties for the object(s) in `HandleList`. If `HandleList` is a vector, only similar object properties are listed.

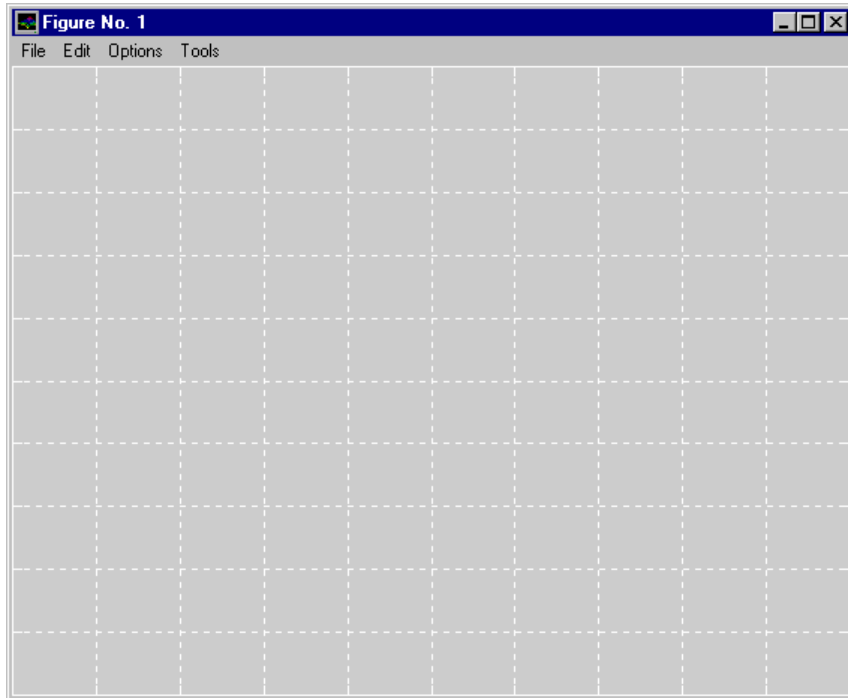


The Callback Editor allows callback type properties to be modified by entering them just as they would be entered from the command line. Similar to the Property Editor, the top of the Callback Editor contains either the currently selected object(s)' Type and Tag or it contains an Object Browser. Below this lies the list of properties that can be modified. The property is the current value of that property.

See Also `guide`, `align`, `menuedit`, `propedit`, Object Browser, Controlled Figure

Description

A Controlled figure is a figure that has been put into an editable state by the Guide Control Panel. When a figure is controlled, objects may be selected, moved, resized, or edited interactively. The figure's menus are hidden from view and a set of menus related to Guide are inserted in their place. To view the stored menus, activate the figure. When a figure is controlled, a white grid that covers the entire figure is present.



See Also

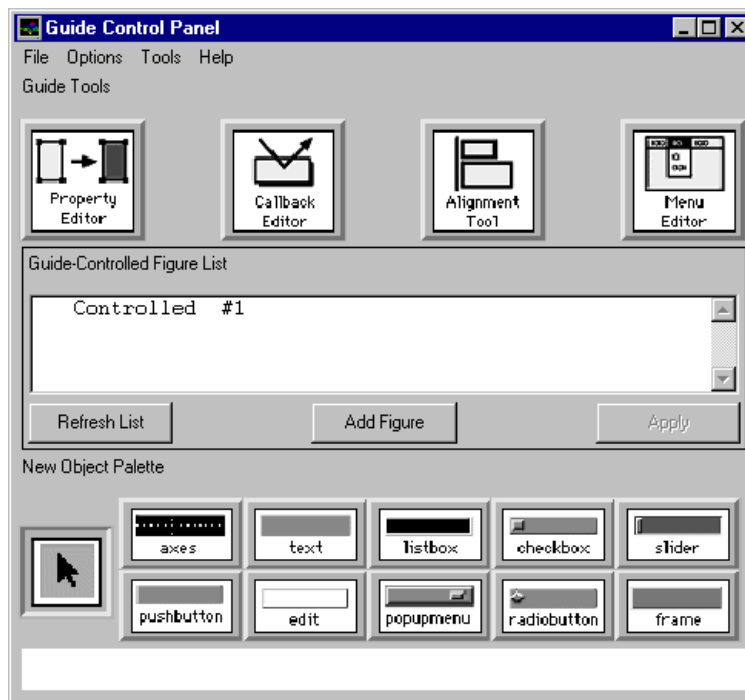
guide, align, cbedit, ctrlpanel, menuedit, propedit, Object Browser

ctlpanel

Purpose Guide control panel.

Syntax `ctl panel`
`ctl panel (HandleList)`

Description `ctl panel` is the initialization function for the Guide Control Panel. `ctl panel (HandleList)` initializes all tools to the figure(s) containing all handles in the `HandleList`.



See Also `guide`, `align`, `cbedit`, `menuedit`, `propedit`, `Object Browser`, `Controlled Figure`

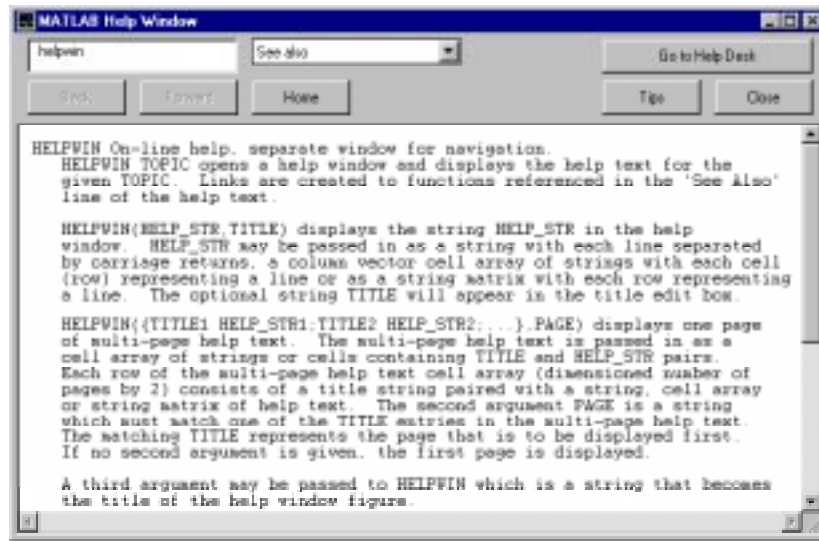
Purpose	Guide Control Panel.
Syntax	<code>gui de</code> <code>gui de(HandleList)</code>
Description	<code>gui de</code> initiates the GUI Design Environment tools that allow Handle Graphics objects to be manipulated interactively. Calling <code>gui de</code> by itself will open the Control Panel if it is not open already and update all other Guide tools that may already be open. All tools are updated to use the current figure. <code>gui de(HandleList)</code> initializes all tools to the figure(s) containing all handles in the <code>HandleList</code> .
See Also	<code>align</code> , <code>cbedit</code> , <code>ctlpanel</code> , <code>menuedit</code> , <code>propedit</code> , Object Browser, Controlled Figure

helpwin

Purpose On-line help

Syntax `hel pwi n t o p i c`

Description `hel pwi n t o p i c` opens a help window and displays the help text for the given *t o p i c*. Links are created to functions referenced in the “See Also” line of the help text.



`hel pwi n(hel p_str, t i t l e)` displays the string `HELP_STR` in the help window. `hel p_str` can be passed in as a string with each line separated by carriage returns, a column vector cell array of strings with each cell (row) representing a line, or a string matrix with each row representing a line. The optional string `t i t l e` appears in the title edit box.

`hel pwi n({t i t l e1 hel p_str1; t i t l e2 hel p_str2; ... }, page)` displays one page of multipage help text. The multipage help text is passed in as a cell array of strings or cells containing `t i t l e` and `hel p_str` pairs. Each row of the multipage help text cell array (dimensioned number of pages by 2) consists of a title string paired with a string, cell array or string matrix of help text. The second argument `page` is a string that must match one of the `t i t l e` entries in the multipage help text. The matching `t i t l e` represents the page that is to be displayed first. If no second argument is given, the first page is displayed.

A third argument can be passed to `helpwin`, which is a string that becomes the title of the help window figure.

Additional arguments, after the window title, are interpreted as Handle Graphics parameter-value pairs to be applied to the text displayed in the help window.

Examples

```
helpwin plot
helpwin('Help String','title')
helpwin(['Help text for' sprintf('\n') 'my m-file.'], 'title')
helpwin({'Help String for'; 'my m-file'}, 'title')

str = { 'Topic 1' 'Help string for Topic 1';
        'Topic 2' 'Help string for Topic 2';
        'Topic 3' 'Help string for Topic 3' }
helpwin(str, 'Topic 2', 'My Title')
```

See Also

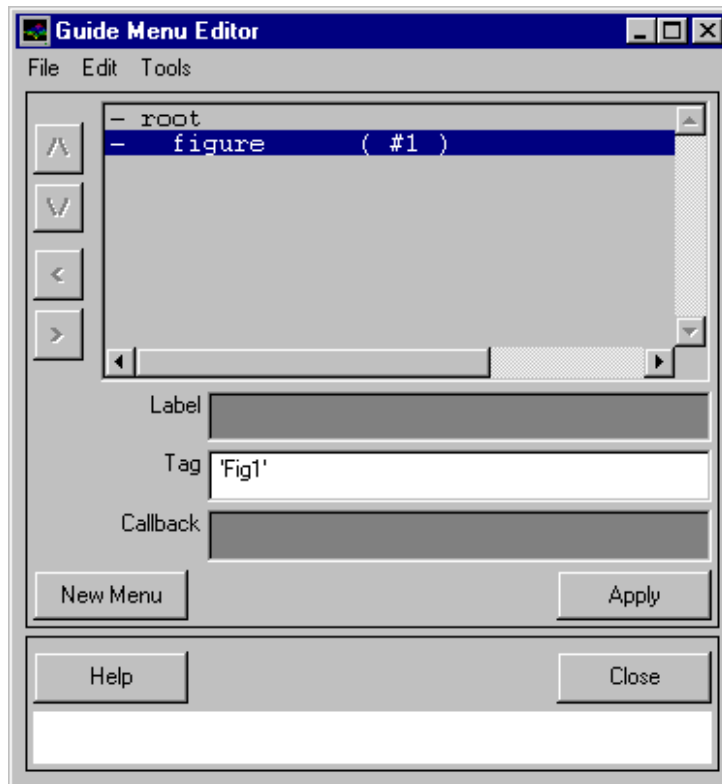
`doc`, `docopt`, `help`, `web`

menuedit

Purpose Edit menu.

Syntax `menuedit`
`menuedit (HandleList)`

Description `menuedit` allows the menus on figures to be edited interactively.
`menuedit (HandleList)` selects all objects in `HandleList`.



The Menu Editor allows the menus on figures to be interactively modified. Menus can be moved up and down in the hierarchy through the use of the arrow buttons on the left hand side of the tool. The Label, Tag, and Callback properties can be modified directly on the tool. All other properties should be

modified with the Property Editor or the Callback Editor. To view menus that are on a controlled figure, the figure must be activated.

See Also

guide, align, cbedit, ctlpanel, propedit, Object Browser, Controlled Figure

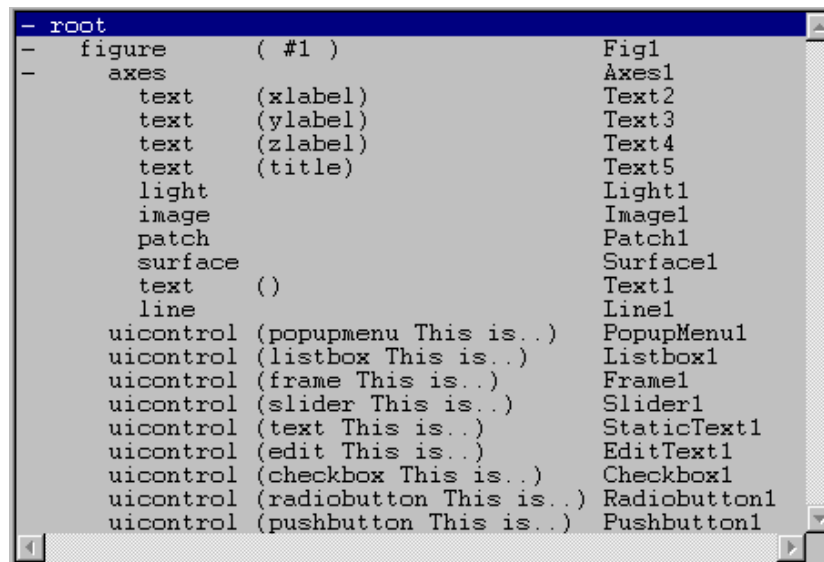
Object Browser

Description

The Object Browser on the Guide tools allows objects to be selected interactively regardless of whether or not a controlled figure exists. If any controlled figures exist, only controlled figures and their children appear in the Object Browser. If no figures are controlled, then all objects from Root down appear. There are three different Object Browsers. The Property Editor and the Callback Editor show all objects. The Menu Editor only shows figures and uimenu. The Alignment Tool only shows figures, uicontrols and axes. Selecting an object on any one of the tools also selects that object on all other tools. There are four columns in the Object Browser. The first column contains "+", "-", or nothing at all. A "+" indicates that the object has children and can be expanded. A "-" indicates the object has children and has already been expanded and a blank indicates the object has no children. The second column contains the object type. The third column contains supplemental information and should help in determining what object is selected. Finally the last column contains the tag of the selected object.

Example

The following is an example of an Object Browser when no figures are being controlled.



```
- root
- figure ( #1 ) Fig1
- axes Axes1
  text (xlabel) Text2
  text (ylabel) Text3
  text (zlabel) Text4
  text (title) Text5
  light Light1
  image Image1
  patch Patch1
  surface Surface1
  text ( ) Text1
  line Line1
  uicontrol (popupmenu This is..) PopupMenu1
  uicontrol (listbox This is..) Listbox1
  uicontrol (frame This is..) Frame1
  uicontrol (slider This is..) Slider1
  uicontrol (text This is..) StaticText1
  uicontrol (edit This is..) EditText1
  uicontrol (checkbox This is..) Checkbox1
  uicontrol (radiobutton This is..) Radiobutton1
  uicontrol (pushbutton This is..) Pushbutton1
```

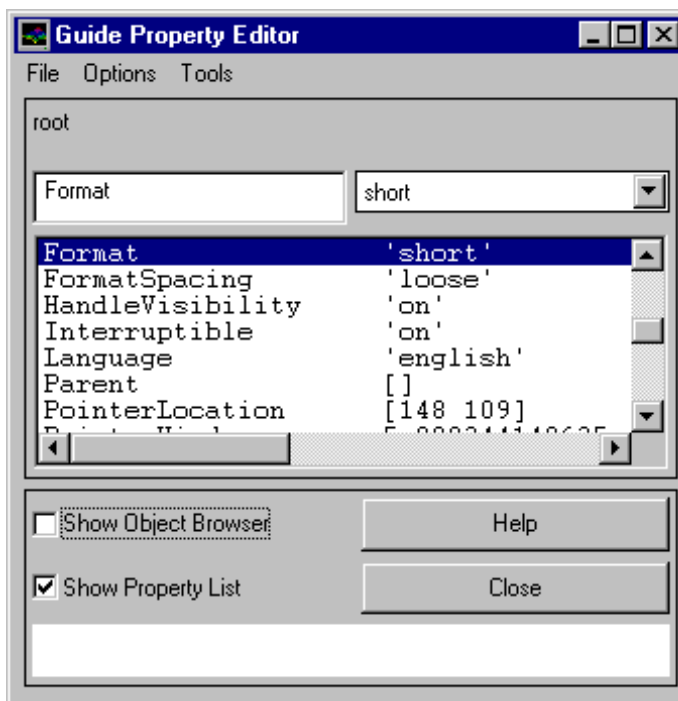
See Also

guide, align, cbedit, ctlpanel, menuedit, propedit, Controlled Figure

Purpose Edit property.

Syntax propedit
propedit (HandleList)

Description propedit edits all properties of any selected object through the use of a graphical interface. propedit (HandleList) edits the properties for the object(s) in HandleList. If HandleList is a vector, only similar object properties are listed.



The Property Editor allows object properties to be modified interactively. The top of the Property Editor shows either the currently selected object's type and that object's tag or it shows the Object Browser. Below this lies the current property and the value of that property. When changing properties, it is not necessary to enter the entire property name but rather just enough to uniquely identify the property. Below the current property, if open, is the list of all avail-

propedit

able properties for the given object(s) and the corresponding value of each of those properties.

See Also

gui de, al i gn, cbedi t, ctl panel , menuedi t, Object Browser, Controlled Figure

A

activating a figure 3-11
align 3-10
Alignment tool 3-14

B

ButtonDownFcn 3-31

C

Callback Editor 3-12
callbacks 2-14
cbedit 3-10, 3-13
close hidden command 3-33
colormaps 3-20
Control Panel 3-10
controlling a figure 3-11

D

design principles 2-3
design process 2-12

E

Enable property 3-30

F

findobj 3-28

G

gca 3-27
gcbf 3-27
gcbo 3-27
guide 3-10

Guide tool list 3-10

Guide-controlled figure list 3-10

H

Handle Graphics 3-2, 3-3

M

Menu Editor 3-15
MenuBar property 3-15
menuedit 3-10

N

new object palette 3-10

O

Object Browser 3-5

P

paper prototyping 2-13
precipitation files example 2-15
propedit 3-10
Property Editor 1-4, 3-3
Property list 3-4

S

subfunctions 3-30
switch statement 3-27
switchyard programming 3-27

T

towork string 3-20

V

Visible property 3-5

W

weather data 2-15