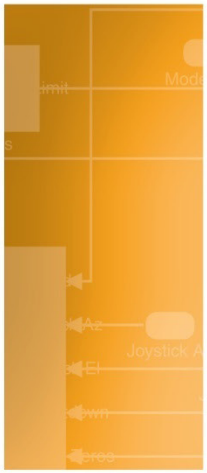
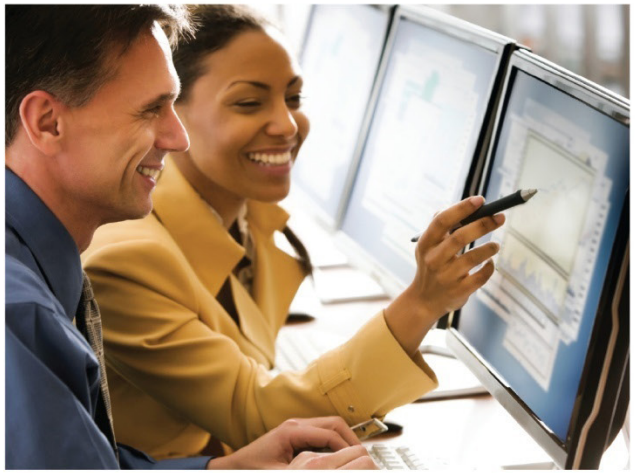
MathWorks | Training Services

Increasing Automation with Programming Constructs

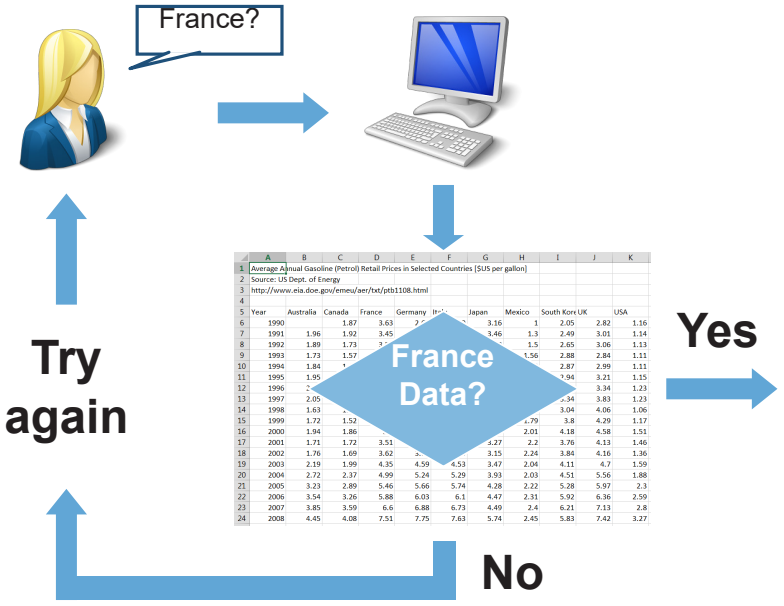
MATLAB® Fundamentals for Aerospace Applications



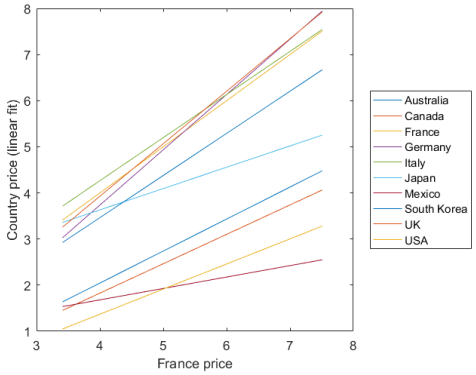


Outline

- Programming constructs
- User interaction
- Decision branching
- Loops




```
Australia/France slope: 0.69351
Canada/France slope: 0.63727
France/France slope: 1
Germany/France slope: 1.1991
Italy/France slope: 0.93439
Japan/France slope: 0.4621
Mexico/France slope: 0.24761
South Korea/France slope: 0.91372
UK/France slope: 1.1382
USA/France slope: 0.54434
```



Chapter Learning Outcomes

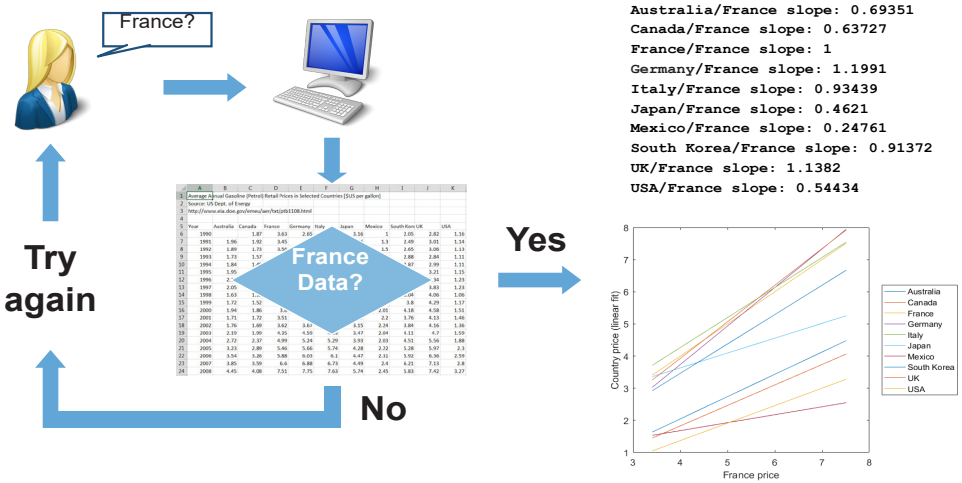
The attendee will be able to:

- Create script files that obtain user input and display output.
- Use loops and logical branching in code files for automation and decision making.

Course Example: Comparing Prices

So far, the MATLAB® scripts you have created have always executed the exact same sequence of commands. In this chapter, the gasoline price data will be used to illustrate how to create a program that can:

- Ask the user for a country.
- Check if the country exists in the data.
- If not, inform the user and continue asking for a country.
- Once a valid country is selected, analyze gasoline prices for all countries in the data set with respect to the selected country and plot the results.



Try

See the list of reserved keywords.

```
iskeyword
```

MATLAB programs can make use of *keywords* to control execution of the code. Keywords are the basic programming components of the MATLAB language. You can use them to build programming constructs that allow your programs to make decisions or repeat a procedure multiple times. Use these constructs to make your code more robust, more flexible, and more maintainable.

Keywords are reserved words with special meanings. The MATLAB Editor displays them in blue, and they are recognized as directions for control flow in your programs.

To see a list of MATLAB programming keywords, type

```
iskeyword
```

To read about the use of a particular keyword and see examples, use the syntax

```
doc keyword
```

MATLAB will not allow you to use a keyword as a variable:

```
for = pi;
```

```
The expression to the left of the equals sign
is not a valid target for an assignment.
```

User Interaction

MATLAB provides a number of functions that allow you to communicate with the user during execution, and give the user the opportunity to provide feedback.

You can provide common interactions in the Live Editor with the `inputdlg`, `disp`, and `fprintf` functions. The `inputdlg` function creates a dialog box to gather user input, then waits for the user to enter input and click **OK**:

```
ctry = inputdlg('Please enter a country:')
```

The `inputdlg` function returns the user input as a cell array. If the user clicks **Cancel**, an empty cell array is returned.

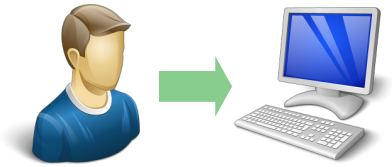
You can create your own error and warning messages in MATLAB with the `error` and `warning` functions, respectively. In their simplest use, they work very much like `disp`. However, unlike simple display statements, these functions create actual MATLAB errors or warnings. This means that errors will stop program execution, and provide a link to the error; warnings will be displayed in orange, and can be optionally disabled.

There are several functions that create predefined dialog boxes without requiring any graphics programming. See **MATLAB → App Building → GUIDE or Programmatic Workflow → Dialog Boxes**.

Try

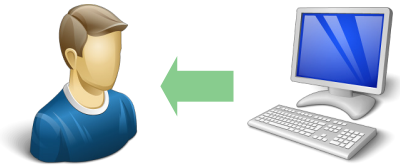
Use different input methods to obtain a country name from the user: `gpinput.mlx`.

Obtain a country name from the user and display information on that country's gasoline prices: `gpinout.mlx`.



	Input	Output	
Text	`input`	`disp` `fprintf`	`warning` `error`

Graphical	`inputdlg`
	`listdlg`
	`uigetfile`
	
	`ginput`
		`msgbox`
		`waitbar`
			`warndlg`
			`errordlg`



Decision Branching

Programming constructs allow you to introduce conditional branching into your programs, so that execution follows one path or another, depending on certain checks that are made along the way.

Mutually exclusive branching based on the values of arbitrary logical conditions is achieved using the `if-elseif-else` construction:

```
if (logical test 1)
    statements 1
elseif (logical test 2)
    statements 2
else
    statements 3
end
```

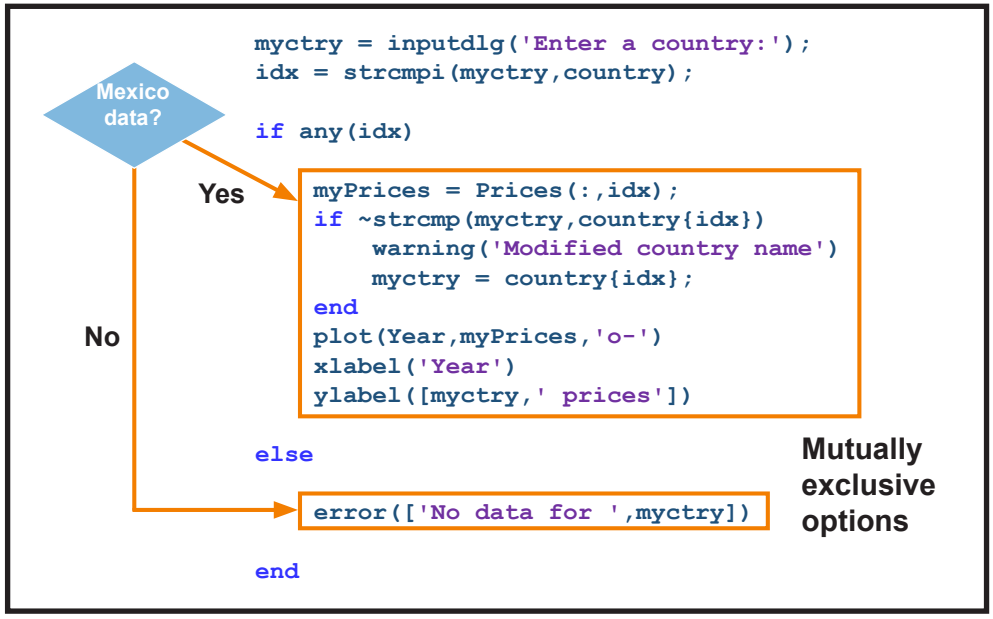
Only one block of code (*`statements n`*) is executed. The logical tests are performed in order and only as necessary. Hence, if *`logical test 1`* evaluates as `true`, *`statements 1`* is executed and *`logical test 2`* is never performed. There can be any number of `elseif` statements, including none. There can be at most one `else` statement (and it must be the last condition).

If there is a finite set of discrete possible values for a variable (e.g., a set of menu options or the number of dimensions of an array), you can use the `switch-case` construction in place of an `if-elseif-else`:

Try

Use logic to test for problems before plotting gasoline prices: `gpif.mlx`.

```
switch expression
    case value 1        % executes if
        statements 1    % expression == value 1
    case value 2
        statements 2
    otherwise           % "else"
        statements 3
end
```



For-Loops

Loops execute a block of code repeatedly within a larger program. A `for`-loop executes code a fixed number of times. Each time through the loop, a parameter known as the *loop index* takes a specified value. The *loop index* is a regular MATLAB variable and can therefore be used in statements within the loop.

```
for index = first:increment:last
    statements
end
```

Note the use of the colon syntax to define the values that the index will take. Although this notation creates a vector, *`index`* will be a scalar, taking each value of the vector in turn. A row vector variable in memory can be used in place of the range declaration:

```
v = [1,2,3,5,8,13];
for k = v
    statements
end
```

As a MATLAB variable, the index can be used in calculations within the loop; it also persists in memory (with the value of *`last`*) after the loop terminates.

As a vectorized language, many MATLAB commands contain implicit loops. Judicious use of vectorized MATLAB functions can enable you to write clean code that uses a single line of code where other languages would require a `for`-loop.

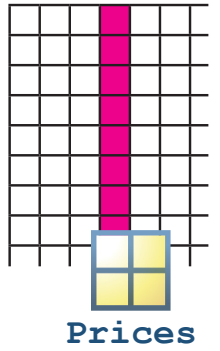
Try

Compare gasoline prices from one country to all other countries: `gpfor.mlx`.

Use other MATLAB constructs to avoid loops: `loops_noloops.mlx`.

```
for k = 1:length(country)
    c = polyfit(myPrices,Prices(:,k),1);
    disp([country{k},'/',myctry,': ',num2str(c(1))])
end
```

```
Australia/France slope: 0.69351
Canada/France slope: 0.63727
France/France slope: 1
Germany/France slope: 1.1991
Italy/France slope: 0.93439
Japan/France slope: 0.4621
Mexico/France slope: 0.24761
South Korea/France slope: 0.91372
UK/France slope: 1.1382
USA/France slope: 0.54434
```



Determining Size

When writing MATLAB code for automation, it is important to be able to determine the dimensions of arrays programmatically, rather than interactively (using the workspace).

The `size` function returns the number of rows and columns as either a vector or two separate scalars:

```
s = size(Prices)
```

```
[19 10]
   s
```

```
[m,n] = size(Prices)
```

```
19   10
m    n
```

19-by-10

You can also use a dimension argument to return the size of just the desired dimension:

```
m = size(Prices,1)
```

```
19
m
```

```
n = size(Prices,2)
```

```
10
n
```

19-by-10

Try

How many countries are in the gasoline price data set?

```
length(country)
```

The `length` function returns the largest value returned by `size`:

```
length(Prices)
```

This is generally intended for use with vectors (where one of the dimensions returned by `size` is 1):

```
length(Year)
```

The `numel` function returns the total number of elements in an array of any dimensions:

```
N = numel(Prices)
```

19-by-10

```
190
N
```

m-by-*n*

n-by-1

1-by-*n*

`size(x)`	[*m n*]	[*n* 1]	[1 *n*]
`size(x,1)`	*m*	*n*	1
`size(x,2)`	*n*	1	*n*
`length(x)`	*max(m,n)*	*n*	*n*
`numel(x)`	*m*n*	*n*	*n*

While-Loops

To use a `for`-loop, it is necessary to know in advance how many iterations of the loop are required. In many cases, however, you may want to execute a block of code repeatedly until a result is achieved or a condition is violated. For this, you can use a `while`-loop:

```
while   condition
    statements
end
```

The code contained in *`statements`* will be evaluated as long as the logical *`condition`* evaluates as `true`. Note that *`condition`* is evaluated before the *`statements`* block runs (i.e., at the beginning of the loop), which may require some initialization prior to the `while` statement. Furthermore, once the *`statements`* block starts to execute, the entire block will be executed even if *`condition`* becomes false; the loop will terminate only when *`condition`* is reevaluated at the beginning of the loop. (Once *`condition`* evaluates as `false` at the beginning of the `while`-loop, the *`statements`* block will not execute and execution will jump to the next statement after `end`.)

A common problem when writing code with `while`-loops is the creation of infinite loops. This is certain to happen if the variables involved in *`condition`* are not updated in the *`statements`* block. However, even if they are updated, it is possible that *`condition`* will never evaluate as `false`:

```
x = 3;
while (x>2)
    x = 2*x;
end
```

Try

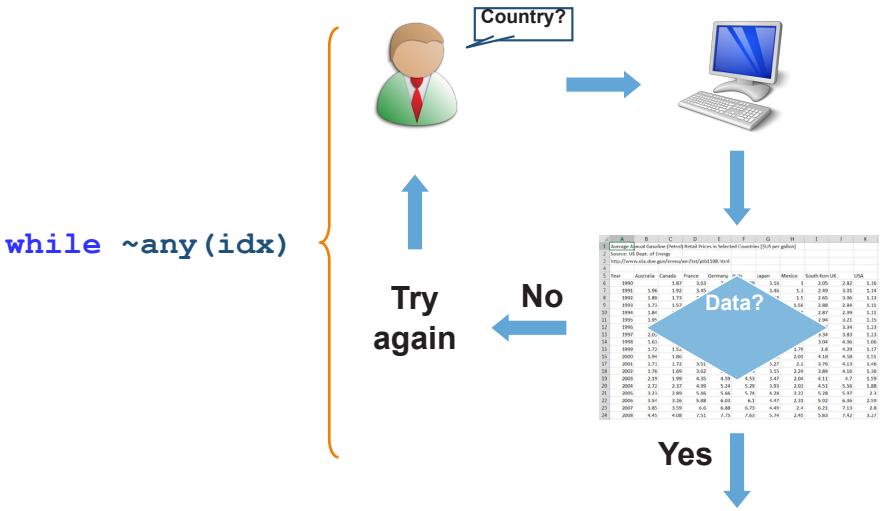
Keep asking the user for a country until a valid country is entered: `gpwhile.mlx`.

Simulate random walks: `randwalk.mlx`.

Is x+y always different to x? Investigate machine precision: `findeps.mlx`.

If this happens, execution can be interrupted by pressing **Ctrl+C**. A simple way to prevent infinite loops is to include a counter variable and a corresponding test condition:

```
x = 3;
n = 1;
while ((x>2) && (n<1000))
    x = 2*x;
    n = n+1;
end
```



Summary

- Programming constructs
- User interaction
- Decision branching
- Loops

Function/Keyword	Use

`if`
`elseif`
`else`
`end`
	Branch code based on conditions

`switch`
`case`
`otherwise`
`end`
	Branch code based on discrete possibilities

`for` `end`	Loop over code a predefined number of times
`while` `end`	Loop over code indefinitely until a condition is violated

`size`
`length`
`numel`
	Determine dimensions of an array

User Interaction Functions			
	Input	Output	
Text	`input`	`disp` `fprintf`	`warning` `error`

Graphical	`inputdlg`
	`menu`
	`uigetfile`
	
	`ginput`
		`msgbox`
		`waitbar`
			`warndlg`
			`errordlg`

Test Your Knowledge

Name:_____

1. If `x = -4` what will be the result of running the following code?

```
if (x < 0) || (sqrt(x) > pi)
    y = 7;
else
    y = 2;
end
```

   A. `y = 7;`
   B. `y = 2;`
   C. `y` will be left undefined
   D. An error message due to taking the square root of a negative number
   E. An error message due to comparing an imaginary number to a real number

2. What construction should you use to loop over a block of code an indefinite number of times?

   A. `if`
   B. `for`
   C. `switch`
   D. `while`
   E. Logical indexing