

MATLAB Fundamentals

Using the MATLAB Desktop Creating Informative Scripts

Summary: Working with Live Scripts

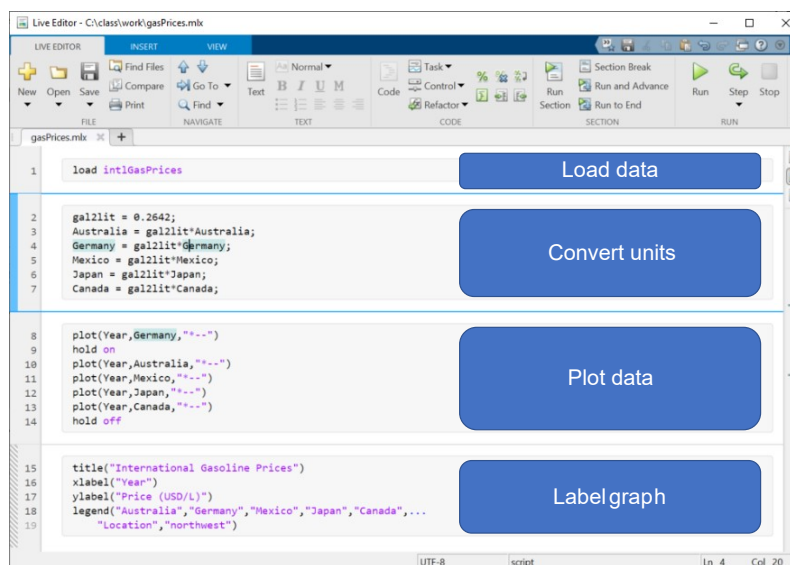
Create and Run a Script

Use the controls in the MATLAB toolstrip to create and run scripts.

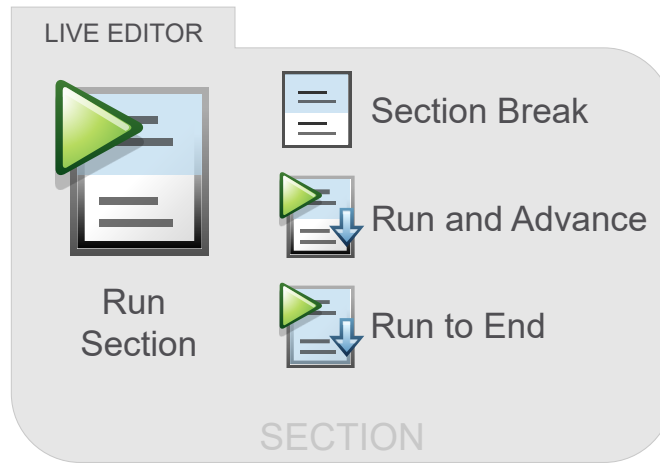


Code Sections

Code sections allow you to organize your code and run sections of code independently. On the **Live Editor** tab, in the **Section** section, click **Section Break** to create a new code section, or press **Ctrl+Alt+Enter**.



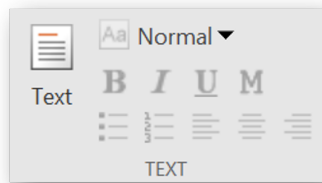
You can run and add code sections in the **Section** section of the **Live Editor** tab in the toolstrip.



Comments and Text

To insert a line of text, click the  **Text** button in the **Text** section of the **Live Editor** tab in the MATLAB Toolstrip.

Format the text using the formatting options provided in the **Text** section.



Comments

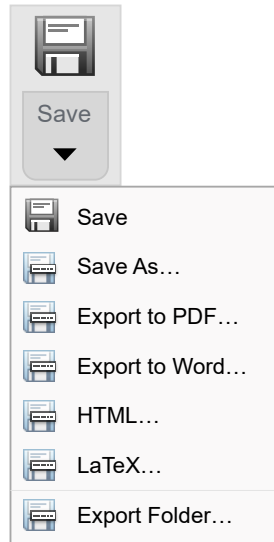
To create a comment, add *% comment* where you want to add more information.

```
load gCosts
% Converts from US$/gal to US$/L
gal2lit = 0.2642; % conversion factor
Germany = gal2lit*Germany;
Australia = gal2lit*Australia;
Mexico = gal2lit*Mexico;
```

Exporting Live Script Files

You can export your live script and results using the **Save** button in the **Live Editor** tab.

Available formats include PDF, Word, HTML, and LaTeX.



Creating and Manipulating Arrays

Summary of Creating and Manipulating Arrays

Summary: Creating and Manipulating Arrays

Manually Entering Arrays

Create a Row Vector

Use square brackets and separate the values using a comma or a space.

```
a = [10 15 20 25]
```

```
a =  
    10    15    20    25
```

Create a Column Vector

Use square brackets and separate the values using a semi-colon.

```
b = [2;3;5;7]
```

```
b =  
     2  
     3  
     5  
     7
```

Transpose a Vector

Use the transpose operator ' .

```
c = b'
```

```
c =  
     2     3     5     7
```

Create a Matrix

Use square brackets and enter values row-by-row. Separate values in a row using a comma or a space, and use a semicolon to start a new row.

```
A = [1 3 5; 2 4 6]
```

```
A =  
    1    3    5  
    2    4    6
```

Creating Evenly-Spaced Vectors

Given the Start Value, End Value, and Interval

Use the colon operator to separate the starting value, interval, and the ending value.

```
a = 3:2:7
```

```
a =  
    3    5    7
```

When Interval is 1

Use the colon operator to separate the starting and the ending value.

```
b = 3:7
```

```
b =  
    3    4    5    6    7
```

Given the Start Value, End Value, and Number of Elements

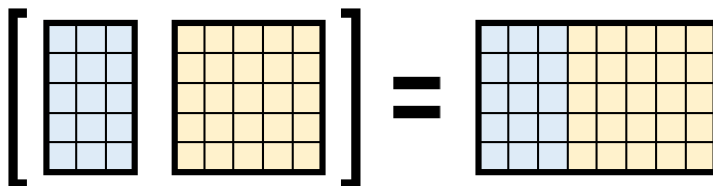
Use the function `linspace` when the number of elements in the vector are known.

```
c = linspace(3.2,8.1,5)
```

```
c =  
    3.2    4.42    5.65    6.87    8.1
```

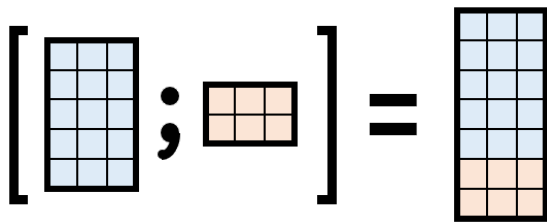
Concatenating Arrays

Horizontal Concatenation



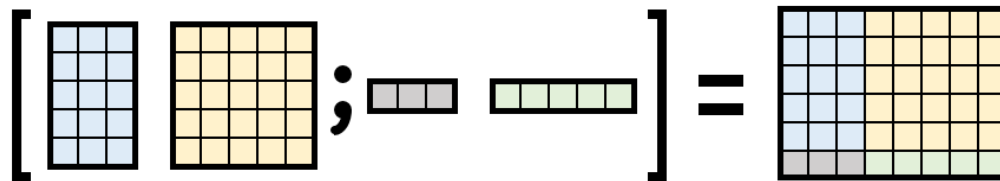
Separate elements using a **comma (,)** or **space ()**

Vertical Concatenation



Separate elements
using a **semicolon (;)**

Combined Concatenation



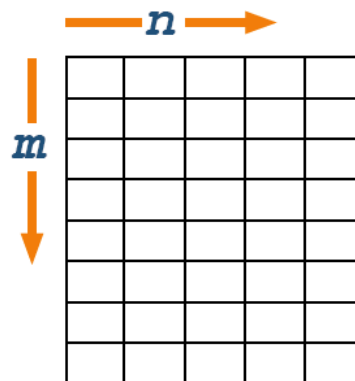
Create each row separating elements with a
comma (,) or **space ()**, then separate the rows
with a **semicolon (;)**

Array Creation Functions



Several functions exist that allow you to create arrays.

A = fun(m,n);

| | |
|------------------|------------------|
| companion | pascal |
| eye | rand |
| gallery | randn |
| hadamard | rosser |
| hankel | toeplitz |
| hilb | vander |
| invhilb | wilkinson |
| magic | zeros |
| ones | |

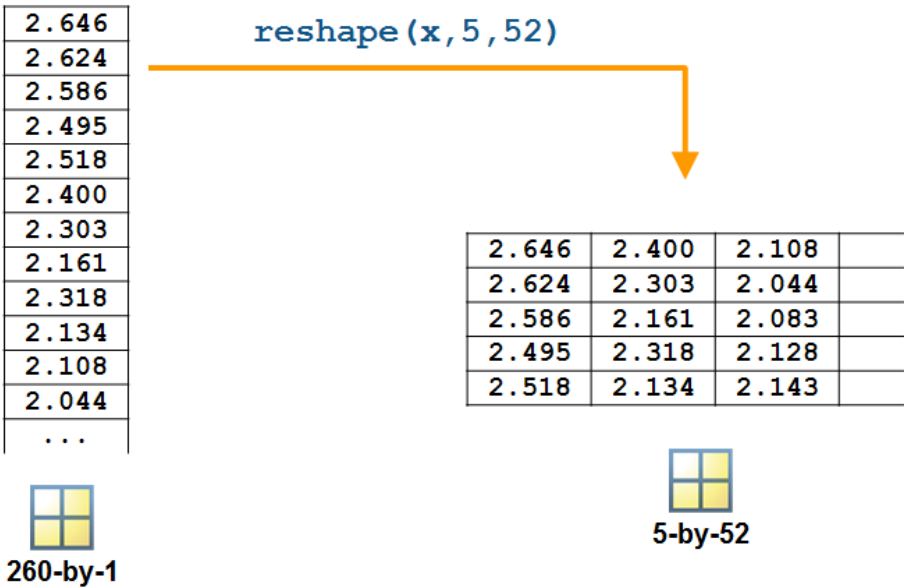


Most of these functions support the calling syntaxes shown below.

| Calling syntax | Output |
|-----------------------|--|
| <code>fun(m,n)</code> |  <i>m-by-n</i> |
| <code>fun(n)</code> |  <i>n-by-n</i> |

Reshaping Arrays

The following column of information is reshaped into a matrix.



Create a vector of random numbers to reshape.

Specify the dimensions for the new array.

For convenience, you can also leave one of the dimensions blank when calling [reshape](#) and that dimension will be calculated automatically.

```
x = rand(260,1);  
  
y = reshape(x,5,52);  
  
y = reshape(x,5,[]);
```

Accessing Data in Arrays

Summary of Accessing Data in Arrays

Summary: Accessing Data in Arrays

Indexing

1

2.3

2

1.5

3

1.3

4

0.9

5

1.3

v

1

1.5

2

1.1

3

2.6

4

0.9

1.5

2.4

1.7

1.4

2.5

1.6

1.9

0.7

2.4

1.1

1.8

2.5

1.9

2.8

0.6

0.6

1

2

3

4

1

2

3

4

5

M

Extract one element from a vector

Extract the last element from a vector

Extract multiple elements from a vector

v(2)

1.5

v(end)

1.3

v([1 end-2:end])

2.3

1.3

0.9

1.3

When you are extracting elements of a matrix you need to provide two indices, the row and column numbers.

Extract one element from a matrix

Extract an entire column. Here, it is the last one.

M(2,3)

1.7

M(:,end)

Extract multiple elements from a matrix.

```
0.9
1.4
0.7
2.5
0.6
```

```
M([1 end],2)
```

```
1.1
2.8
```

Changing Elements in Arrays

Change one element from a vector

```
v(2) = 0
```

```
2.3
0
1.3
0.9
1.3
```

Change multiple element of a vector to the same value

```
v(1:3) = 0
```

```
0
0
0
0.9
1.3
```

Change multiple element of a vector to different values

```
v(1:3) = [3 5 7]
```

```
3
5
7
0.9
1.3
```

Assign a non-existent value

```
v(9) = 42
```

```
3
5
7
0.9
1.3
0
0
0
42
```


Remove elements from a vector

```
v(1:3) = []
```

0.9

1.3

0

0

0

42

Changing elements in matrices works the same way as with vectors, but you must specify both rows and columns.

Mathematical and Statistical Operations with Arrays

Summary of Operations with Arrays

Summary: Mathematical and Statistical Operations with Arrays

Performing Operations on Arrays

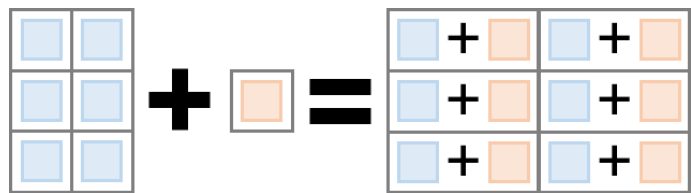
There are many operators that behave in element-wise manner, i.e., the operation is performed on each element of the array individually.

Mathematical Functions

$$\sin \left(\begin{bmatrix} \square & \square \\ \square & \square \\ \square & \square \end{bmatrix} \right) = \begin{bmatrix} \sin(\square) & \sin(\square) \\ \sin(\square) & \sin(\square) \\ \sin(\square) & \sin(\square) \end{bmatrix}$$

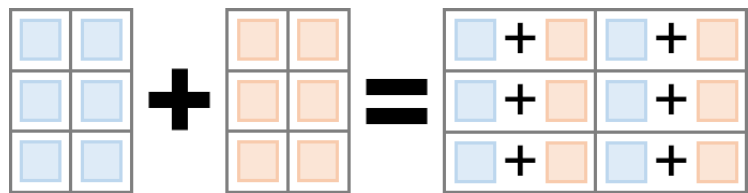
| Other Similar Functions | |
|-------------------------|--------------------|
| sin | Sine |
| cos | Cosine |
| log | Logarithm |
| round | Rounding Operation |
| sqrt | Square Root |
| mod | Modulus |
| Many more | |

Matrix Operations (Including Scalar Expansion)



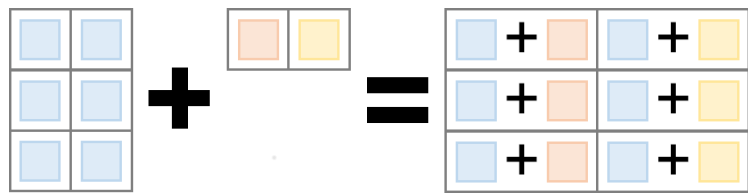
| Operators | |
|-----------|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| ^ | Exponentiation (Matrix exponentiation) |

Element-wise Operations



| Operators | |
|-----------|-----------------------------|
| + | Addition |
| - | Subtraction |
| .* | Element-wise Multiplication |
| ./ | Element-wise Division |
| .^ | Element-wise Exponentiation |

Implicit Expansion



| Operators | |
|-----------|-----------------------------|
| + | Addition |
| - | Subtraction |
| .* | Element-wise Multiplication |
| ./ | Element-wise Division |
| .^ | Element-wise Exponentiation |

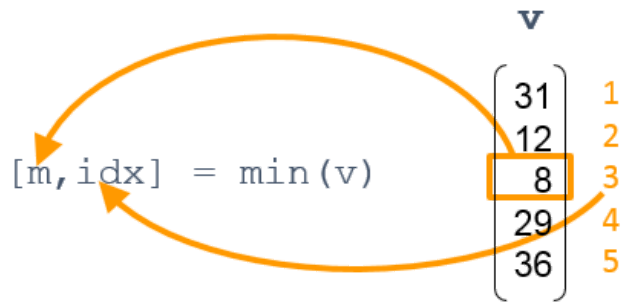
Array operations can be performed on operands of different compatible sizes. Two arrays have compatible sizes if the size of each dimension is either the same or one.

Calculating Statistics of Vectors

Common Statistical Functions

| Function | Description |
|----------|--|
| min | Returns the minimum element |
| max | Returns the maximum element |
| mean | Returns the average of the elements |
| median | Returns the median value of the elements |

Using min and max



Ignoring NaNs

When using statistical functions, you can ignore NaN values

```
avg = mean(v, "omitnan")
```

Statistical Operations on Matrices

Some common mathematical functions which calculate a value for each column in a matrix include:

| Function | Behavior |
|----------|-----------------------|
| max | Largest elements |
| min | Smallest elements |
| mean | Average or mean value |
| median | Median value |
| mode | Most frequent values |
| std | Standard deviation |
| var | Variance |
| sum | Sum of elements |
| prod | Product of elements |

A = [8 2 4 ; 3 2 6 ; 7 5 3 ; 7 10 8]

A =

8 2 4
3 2 6
7 5 3
7 10 8

Amax = max(A)
Amax =

8 10 8

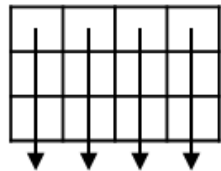
Astd = std(A)
Astd =

2.2174 3.7749 2.2174

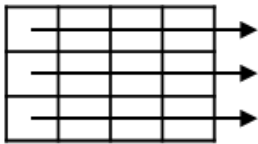
Asum = sum(A)
Asum =

25 19 21

Many statistical functions accept an optional dimensional argument that specifies whether the operation should be applied to columns independently (the default) or to rows.



Dimension = 1



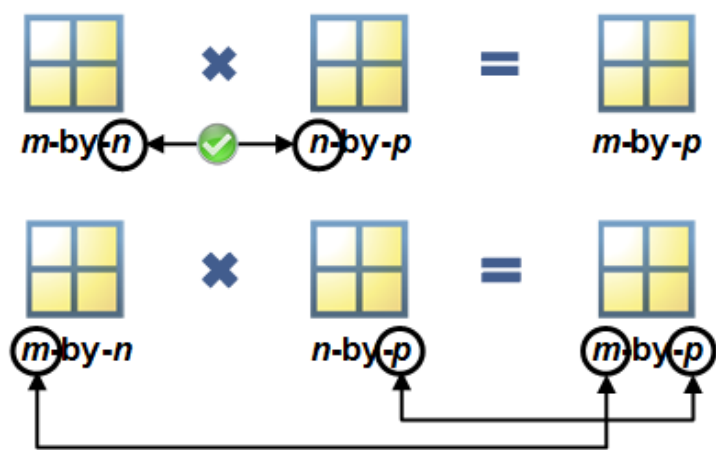
Dimension = 2

```
>> M = mean(A,dim)
```

| Outputs | | Inputs | |
|---------|---|--------|--|
| M | Vector of average values along dimension dim. | A | Matrix |
| | | dim | Dimension across which the mean is taken. 1: the mean of each column 2: the mean of each row |

Matrix Multiplication

Matrix multiplication requires that the inner dimensions agree. The resultant matrix has the outer dimensions.



Solving Systems of Linear Equations

| Expression | Interpretation |
|----------------------|-----------------------------------|
| $x = B/A$ | Solves $x \cdot A = B$ (for x) |
| $x = A \backslash B$ | Solves $A \cdot x = B$ (for x) |

Visualizing Data in 2D and 3D

Summary of Visualizing Data in 2D and 3D

Summary: Visualizing Data in 2D and 3D

Identifying Available Plot Types

| Function | Description |
|-----------|---|
| scatter | Scatter plot, with variable marker size and color |
| bar | Bar graph (vertical and horizontal) |
| stem | Discrete sequence (signal) plot |
| stairs | Stairstep graph |
| area | Filled area plot |
| pie | Pie chart |
| histogram | Histogram |

```
>> scatter(x,y,n,c,filled)
```

Inputs

| | |
|--------|--|
| x | x-data |
| y | y-data |
| n | marker size |
| c | color |
| filled | If provided, markers will be filled in disks. Otherwise, they are circles. |

See the complete list of all available plots [here](#).

Customizing Annotations

Arrays of strings are useful for annotating visualizations. Use square brackets, `[]`, with spaces and semicolons, `;` to create a string array the same way you create a numeric matrix.

```
x = ["hello" "sweet"; "peaceful" "world"]
x =
    2x2 string array
    "hello"      "sweet"
    "peaceful"   "world"
```

```
ylabel("\pi r^2")
```

You can use markup in your labels.

[xticks](#) Sets tick locations along the x-axis.

[xticklabels](#) Labels the x-axis ticks.

[xtickangle](#) Rotates the x-axis tick labels.

Customizing Plot Properties

Specifying Property Values

```
plot(x,y,linespec,Property1,Value1,Property2,Value2,Property3,Value3,...)
```

See the complete list of line specifications here:

https://www.mathworks.com/help/matlab/ref/plot.html#btzitol_sep_mw_3a76f056-2882-44d7-8e73-c695c0c54ca8.

Common line properties to modify:

- "LineWidth" (width of the line and marker edges)
- "MarkerSize" (size of the marker symbols)
- "MarkerEdgeColor" (color of the edge of the marker symbols)
- "MarkerFaceColor" (color of the interior of the marker symbols)
- "Color" (color of the line, particularly when given as RGB values)

[MATLAB Line Properties reference](#)

Specifying Colors

| | | | |
|---------------|--------------|------------|-------------|
| red ("r") | green ("g") | blue ("b") | black ("k") |
| magenta ("m") | yellow ("y") | cyan ("c") | white ("w") |

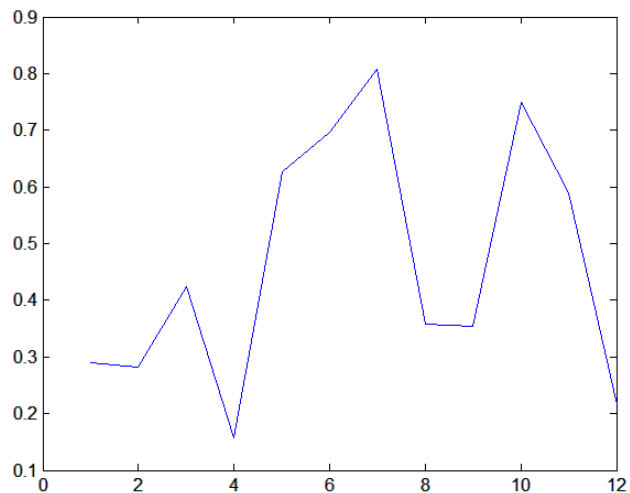
Or as a vector `[R G B]` where each value is from 0 to 1.

Axis Control

Get Axes Limits

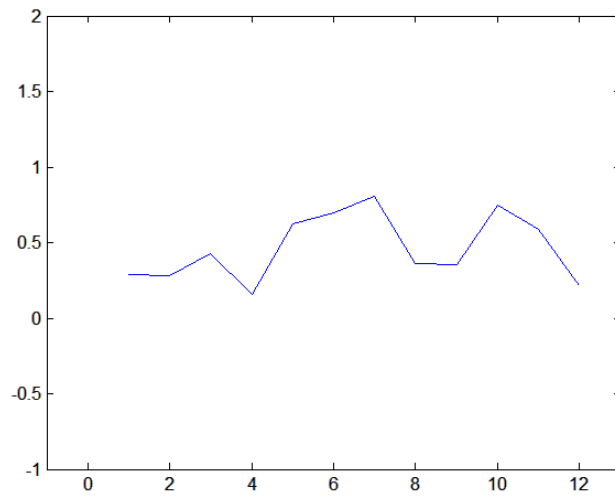
```
v = axis
```

```
v =  
    0    12    0.1    0.9
```



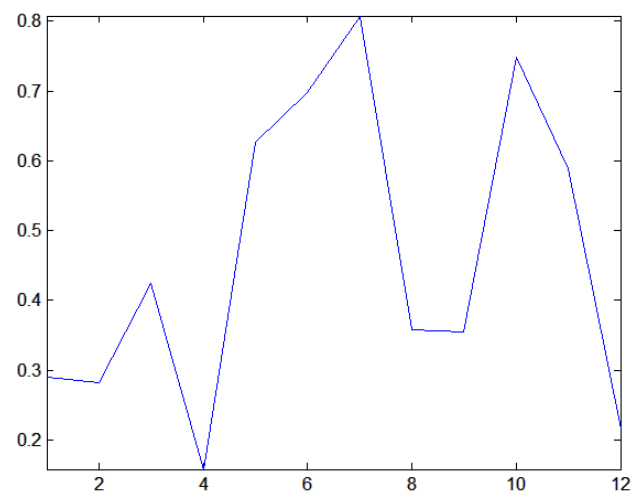
Custom Axis Limits

```
xlim([-1 13])  
ylim([-1 2])
```



Axis Limits = Data Range

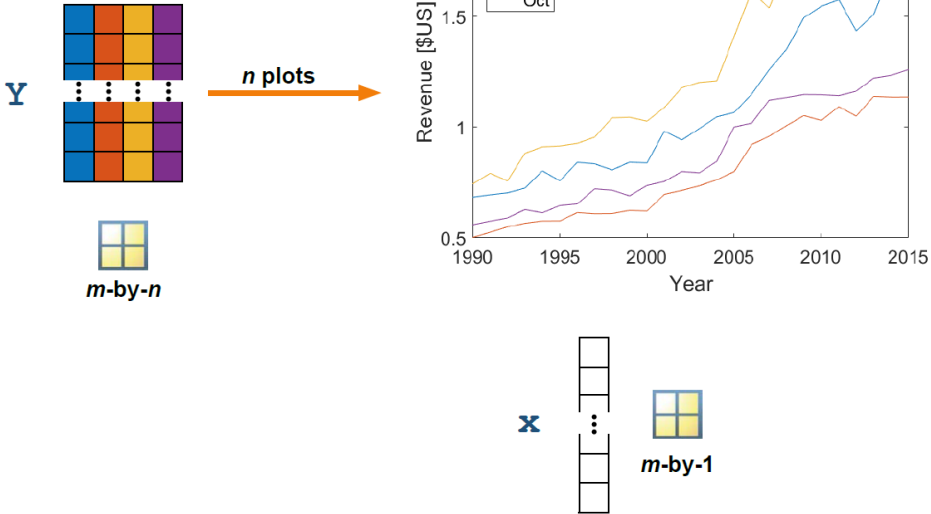
```
axis tight
```



Plotting Multiple Columns

You can use the `plot` function on a matrix to plot each column as a separate line in your plot.

`plot(x,Y)`



Visualizing Matrices

You can use visualization functions to plot your three-dimensional data.

`z` is a 5-by-5 matrix

The `surf` function plots `z(j,k)` over the point `x=k` and `y=j`

To specify `x` and `y` coordinates, you can pass them in as vectors. Here,

- The number of elements of `x` must match the number of columns of `z`
- The number of elements of `y` must match the number of rows of `z`

`z`

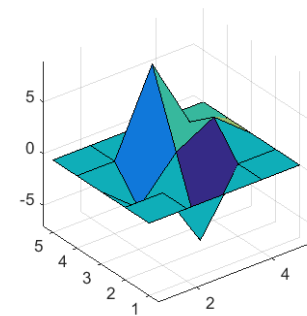
`z =`

```

0     0     0     0     0
0     0    -6     0     0
0    -3     1     3     0
0     0     8     1     0
0     0     0     0     0

```

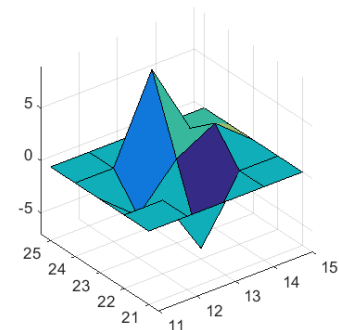
`surf(z)`



`x = 11:15;`

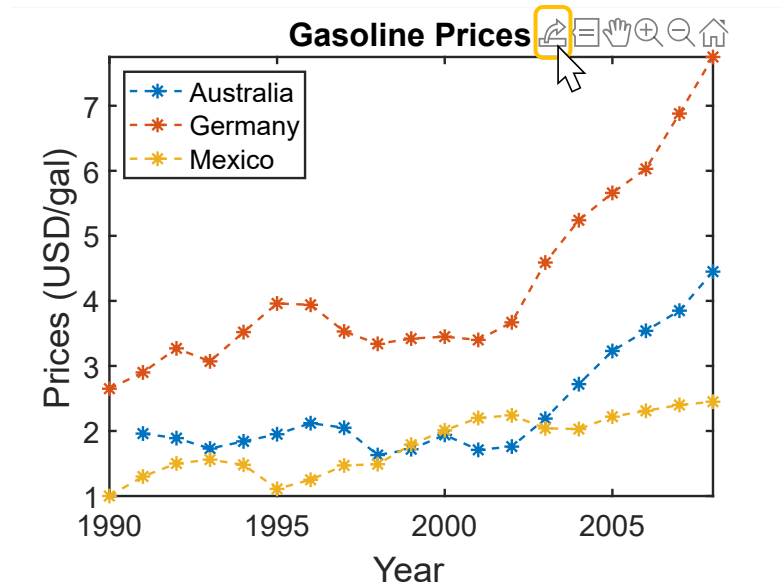
`y = 21:25;`

`surf(x,y,z)`



Exporting a Figure

You can either copy and paste output or export a figure as an image file.



Conditional Data Selection

Summary of Conditional Data Selection

Summary: Conditional Data Selection

Logical Operations and Variables

Relational Operators

| | |
|--------------------|-----------------------|
| <code>==</code> | Equal |
| <code>></code> | Greater than |
| <code><</code> | Less than |
| <code>>=</code> | Greater than or equal |
| <code><=</code> | Less than or equal |
| <code>~=</code> | Not equal |

```
v = [6 7 8 9];
w = [2 4 8 16];
NE = v ~= w
NE =
    1    1    0    1
```

Logical Operators

| | |
|--------------------|-----|
| <code>&</code> | AND |
| <code> </code> | OR |
| <code>~</code> | NOT |

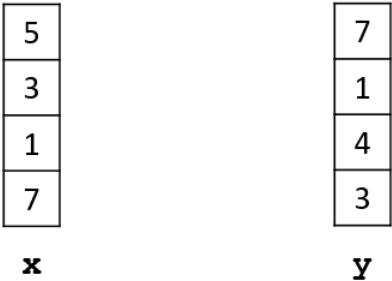
```
v = [6 7 8 9];
w = [2 4 8 16];
x = 5;
A = (v > x) & (w > x)
A =
    0    0    1    1
```

Counting Elements

| Purpose | Function | Output |
|---|----------------------|------------|
| Are any of the elements true? | any | true/false |
| Are all the elements true? | all | true/false |
| How many elements are true? | nnz | double |
| What are the indices of the elements that are true? | find | double |

Logical Indexing

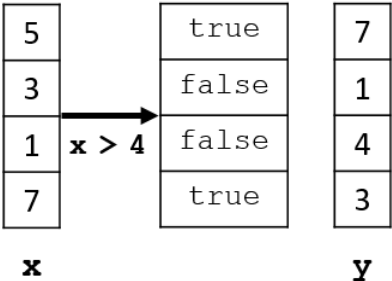
Purpose: Select the elements of an array based on certain criteria.



Step 1: Create a logical vector by evaluating the given condition.

Example:

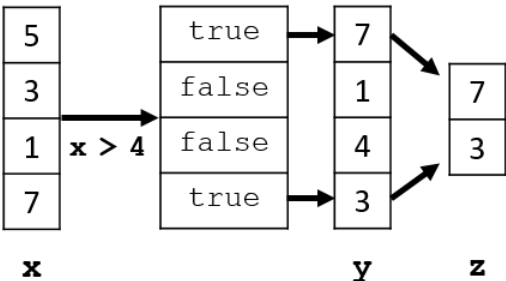
```
idx = x > 4
```



Step 2: Use the logical vector as an index into another array to extract the elements corresponding to the true values.

Example:

```
idx = x > 4      or      z = y(idx)  
z = y(idx)
```



Tables of Data

Summary of Tables of Data

Summary: Tables of Data

Storing Data in a Table

The [readtable](#) function creates a table in MATLAB from a data file.

The [table](#) function can create a table from workspace variables.

The [array2table](#) function can convert a numeric array to a table. The [VariableNames](#) property can be specified as a string array of names to include as variable names in the table.

```
EPL = readtable("EPLresults.xlsx","TextType","string");
```

```
teamWinsTable = table(team,wins)
```

```
teamWins =
```

| Team | Wins |
|---------------------|------|
| "Arsenal" | 20 |
| "Chelsea" | 12 |
| "Leicester City" | 23 |
| "Manchester United" | 19 |

```
stats = array2table(wdl, ...
    "VariableNames",["Wins" "Draws" "Losses"])
```

```
stats =
```

| Wins | Draws | Losses |
|------|-------|--------|
| 20 | 11 | 7 |
| 12 | 14 | 12 |
| 23 | 12 | 3 |
| 19 | 9 | 10 |

Sorting Table Data

The [sortrows](#) function sorts the data in ascending order, by default.

Use the optional `"descend"` parameter to sort the list in descending order.

You can also sort on multiple variables, in order, by specifying a string array of variable names.

You can also show [summary](#) statistics for variables in a table.

```
EPL = sortrows(EPL,"HomeWins");
```

```
EPL = sortrows(EPL,"HomeWins","descend");
```

```
EPL = sortrows(EPL,["HomeWins" "AwayWins"],"descend");
```

```
summary(EPL)
```

Extracting Portions of a Table

Display the original table.

Inside parenthesis, specify the row numbers of the observations and column numbers of the table variables you would like to select.

You may also use the name of the variable for indexing.

If you want to reference more than one variable, use a string array containing the variable names.

EPL

EPL =

| Team | HW | HD | HL | AW | AD | AL |
|---------------------|----|----|----|----|----|----|
| "Leicester City" | 12 | 6 | 1 | 11 | 6 | 2 |
| "Arsenal" | 12 | 4 | 3 | 8 | 7 | 4 |
| "Manchester City" | 12 | 2 | 5 | 7 | 7 | 5 |
| "Manchester United" | 12 | 5 | 2 | 7 | 4 | 8 |
| "Chelsea" | 5 | 9 | 5 | 7 | 5 | 7 |
| "Bournemouth" | 5 | 5 | 9 | 6 | 4 | 9 |
| "Aston Villa" | 2 | 5 | 12 | 1 | 3 | 15 |

EPL(2:4,[1 2 5])

ans =

| Team | HW | AW |
|---------------------|----|----|
| "Arsenal" | 12 | 8 |
| "Manchester City" | 12 | 7 |
| "Manchester United" | 12 | 7 |

EPL(2:4,["Team" "HW" "AW"])

ans =

| Team | HW | AW |
|---------------------|----|----|
| "Arsenal" | 12 | 8 |
| "Manchester City" | 12 | 7 |
| "Manchester United" | 12 | 7 |

Extracting Data from a Table

Display the original table.

You can use dot notation to extract data for use in calculations or plotting.

EPL

EPL =

| Team | HW | HD | HL | AW | AD | AL |
|---------------------|----|----|----|----|----|----|
| "Leicester City" | 12 | 6 | 1 | 11 | 6 | 2 |
| "Arsenal" | 12 | 4 | 3 | 8 | 7 | 4 |
| "Manchester City" | 12 | 2 | 5 | 7 | 7 | 5 |
| "Manchester United" | 12 | 5 | 2 | 7 | 4 | 8 |

tw = EPL.HW + EPL.AW

You can also use dot notation to create new variables in a table.

If you want to extract multiple variables, you can do this using curly braces.

Specify row indices to extract specific rows.

```
tw =
    23
    20
    19
    19

EPL.TW = EPL.HW + EPL.AW

EPL =
```

| Team | HW | HD | HL | AW | AD | AL | TW |
|---------------------|----|----|----|----|----|----|----|
| "Leicester City" | 12 | 6 | 1 | 11 | 6 | 2 | 23 |
| "Arsenal" | 12 | 4 | 3 | 8 | 7 | 4 | 20 |
| "Manchester City" | 12 | 2 | 5 | 7 | 7 | 5 | 19 |
| "Manchester United" | 12 | 5 | 2 | 7 | 4 | 8 | 19 |

```
draws = EPL{:, ["HD" "AD"]}

draws =
     6     6
     4     7
     2     7
     5     4

draws13 = EPL{[1 3], ["HD" "AD"]}

draws =
     6     6
     2     7
```

Exporting Tables

You can use the [writetable](#) function to create a file from a table.

```
writetable(tableName, "myFile.txt", "Delimiter", "\t")
```

The file format is based on the file extension, such as .txt, .csv, or .xlsx, but you can also specify a delimiter.

[writetable](#) Write a table to a file.

Organizing Tabular Data

Summary of Organizing Tabular Data


Summary: Organizing Tabular Data

Combining Tables


If the tables are already aligned so that the rows correspond to the same observation, you can concatenate them with square brackets.

```
[teamInfo games]
```

| | Team | Payroll_M_ | Manager |
|---|-----------------|------------|------------------|
| 1 | "Leicester ..." | 48.2000 | "Claudio Ran..." |
| 2 | "Arsenal" | 192 | "Arsène Wen..." |
| 3 | "Tottenham ..." | 110.5000 | "Mauricio Po..." |
| 4 | "Mancheste..." | 193.8000 | "Manuel Pell..." |
| 5 | "Mancheste..." | 203 | "Louis van G..." |

 **teamInfo**

| | Wins | Draws | Losses |
|---|------|-------|--------|
| 1 | 23 | 12 | 3 |
| 2 | 20 | 11 | 7 |
| 3 | 19 | 13 | 6 |
| 4 | 19 | 9 | 10 |
| 5 | 19 | 9 | 10 |

 **games**

Leicester
Arsenal
Tottenham
Manchester City
Manchester United

If the tables are *not* already aligned so that the rows correspond to the same observation, you can still combine the data by merging them with a join.

| | Player | Goals |
|---|-----------------|-------|
| 1 | "Alex Morgan" | 6 |
| 2 | "Megan Rapinoe" | 6 |
| 3 | "Rose Lavelle" | 3 |

 **uswntTop3**

| | Player | Position |
|---|-----------------|--------------|
| 1 | "Rose Lavelle" | "midfielder" |
| 2 | "Alex Morgan" | "forward" |
| 3 | "Megan Rapinoe" | "forward" |

 **posTop3**

The `join` function can combine tables with a common variable.

```
top3 = join(uswntTop3,posTop3)
```

```
top3 =
```

| Player | Goals | Position |
|-----------------|-------|--------------|
| "Alex Morgan" | 6 | "forward" |
| "Megan Rapinoe" | 6 | "forward" |
| "Rose Lavelle" | 3 | "midfielder" |

Table Properties

Display the table properties.

EPL.Properties

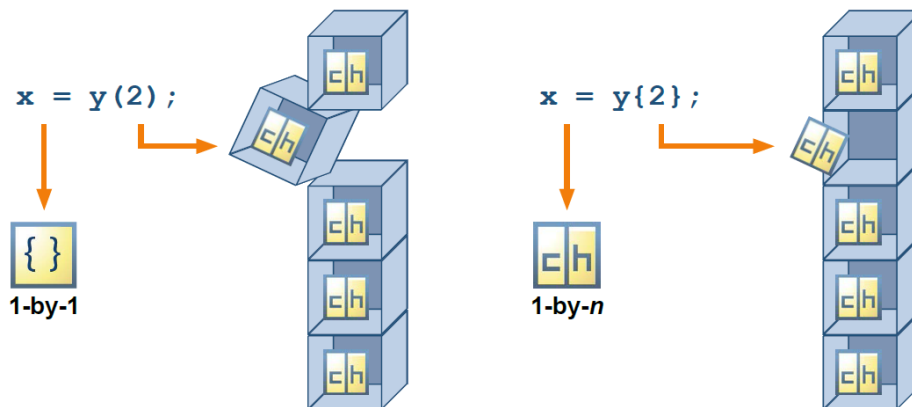
You can access an individual property of Properties using dot notation.

```
ans =
    Table Properties with properties:

        Description: ''
        UserData: []
        DimensionNames: {'Row' 'Variable'}
        VariableNames: {1x11 cell}
        VariableDescriptions: {1x11 cell}
        VariableUnits: {}
        VariableContinuity: []
        RowNames: {}
        CustomProperties: No custom properties are set.

EPL.Properties.VariableNames
ans =
    1x11 cell array
    Columns 1 through 4
        {'Team'}    {'HomeWins'}    {'HomeDraws'}    {'HomeLosses'}
    Columns 5 through 8
        {'HomeGF'}    {'HomeGA'}    {'AwayWins'}    {'AwayDraws'}
    Columns 9 through 11
        {'AwayLosses'}    {'AwayGF'}    {'AwayGA'}
```

Indexing into Cell Arrays



The variable `varNames` is a cell array that contains character arrays of different lengths in each cell.

Using parentheses to index produces a cell array, not the character array inside the cell.

In order to extract the contents inside the cell, you

```
varNames = teamInfo.Properties.VariableNames
```

```
'Team'    'Payroll_M__'    'Manager'    'ManagerHireDate'
```

```
varName(2)
```

```
'Payroll_M__'
```

```
varName{2}
```

should index using curly braces, `{ }`.

Using curly braces allows you to rename the variable.

```
'Payroll_M__'
```

```
varName{2} = 'Payroll'
```

```
'Team'
```

```
'Payroll'
```

```
'Manager'
```

```
'ManagerHireDate'
```

Specialized Data Types

Summary of Specialized Data Types

Summary: Specialized Data Types

Working with Dates and Times

Dates are often automatically detected and brought in as `datetime` arrays.

Many functions operate on `datetime` arrays directly, such as [sortrows](#).

You can create a [datetime](#) array using numeric inputs. The first input is year, then month, then day.

To create a vector, you can specify an array as input to the `datetime` function.

```
teamInfo
```

```
ans =
```

| Manager | ManagerHireDate |
|-------------------|-----------------|
| "Rafael Benítez" | 3/11/2016 |
| "Claudio Ranieri" | 7/13/2015 |
| "Ronald Koeman" | 6/16/2014 |
| "David Unsworth" | 5/12/2016 |
| "Slaven Bilić" | 6/9/2015 |

```
sortrows(teamInfo,"ManagerHireDate")
```

```
ans =
```

| Manager | ManagerHireDate |
|-------------------|-----------------|
| "Ronald Koeman" | 6/16/2014 |
| "Slaven Bilić" | 6/9/2015 |
| "Claudio Ranieri" | 7/13/2015 |
| "Rafael Benítez" | 3/11/2016 |
| "David Unsworth" | 5/12/2016 |

```
t = datetime(1977,12,13)
```

```
t =
```

```
13-Dec-1977
```

```
ts = datetime([1903;1969],[12;7],[17;20])
```

```
ts =
    17-Dec-1903
    20-Jul-1969
```

Operating on Dates and Times

Create `datetime` variables to work with.

Use subtraction to produce a duration variable.

Functions such as [years](#) and [days](#) can help make better sense of the output.

They can also create durations from a numeric value.

Use the [between](#) function to produce a context-dependent `calendarDuration` variable.

Create a calendar duration from a numeric input with functions such as [calmonths](#) and [calyears](#).

```
seasonStart = datetime(2015,8,8)
seasonStart =
    08-Aug-2015

seasonEnd = datetime(2016,5,17)
seasonEnd =
    17-May-2016

seasonLength = seasonEnd - seasonStart
seasonLength =
    6792:00:00

seasonLength = days(seasonLength)
seasonLength =
    283

seconds(5)
ans =
    5 seconds

seasonLength = between(seasonStart,seasonEnd)
seasonLength =
    9mo 9d

calmonths(2)
ans =
    2mo
```

You can learn more about `datetime` and duration functions in the documentation.

[Create Date and Time Arrays](#)

Representing Discrete Categories

`x` is a string array.

```
x = ["C" "B" "C" "A" "B" "A" "C"];
```

You can convert `x` into a categorical array, `y`, using the [categorical](#) function.

You can use `==` to create a logical array, and count elements using `nnz`.

You can view category statistics using the [summary](#) function.

You can view combine categories using the [mergcats](#) function.

```
x =
    "C"    "B"    "C"    "A"    "B"    "A"    "C"

y = categorical(x);

y =
    C     B     C     A     B     A     C

nnz(x == "C")

ans =
     3

summary(y)

    A     B     C
    2     2     3

y = mergcats(y, ["B" "C"], "D")

y =
    D     D     D     A     D     A     D
```

Preprocessing Data

Summary of Preprocessing Data

Summary: Preprocessing Data

Normalizing Data

[normalize](#) Normalize data using a specified normalization method.

Normalize the columns of a matrix using z-scores.

Center the mean of the columns in a matrix on zero.

Scale the columns of a matrix by the first element of each column.

Stretch or compress the data in each column of a matrix into a specified interval.

```
xNorm = normalize(x)
```

```
xNorm = normalize(x, "center", "mean")
```

```
xNorm = normalize(x, "scale", "first")
```

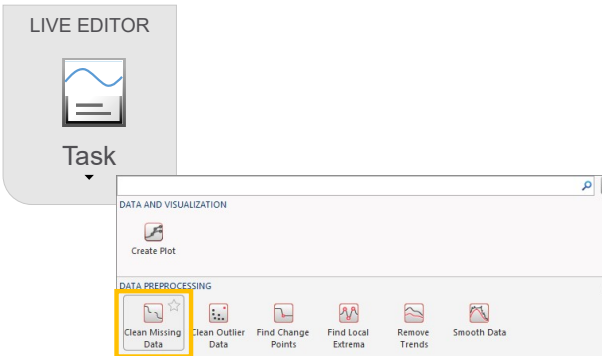
```
xNorm = normalize(x, "range", [a b])
```

Working with Missing Data

Any calculation involving NaN results in NaN. There are three ways to work around this, each with advantages and disadvantages:

| | |
|--|--|
| Ignore missing data when performing calculations. | Maintains the integrity of the data but can be difficult to implement for involved calculations. |
| Remove missing data. | Simple but, to keep observations aligned, must remove entire rows of the matrix where any data is missing, resulting in a loss of valid data. |
| Replace missing data. | Keeps the data aligned and makes further computation straightforward, but modifies the data to include values that were not actually measured or observed. |

The **Clean Missing Data** task can be used to remove or interpolate missing data. You can add one to a script by selecting it from the **Live Editor** tab in the toolstrip.



Data contains missing values, in the form of both -999 and NaN.

The `ismissing` function identifies only the NaN elements by default.

Specifying the set of missing values ensures that `ismissing` identifies all the missing elements.

Use the `standardizeMissing` function to convert all missing values to NaN.

```
x = [2 NaN 5 3 -999 4 NaN];

ismissing(x)

ans =
1x7 logical array
0    1    0    0    0    0    1

ismissing(x,[-999,NaN])

ans =
1x7 logical array
0    1    0    0    1    0    1

xNaN = standardizeMissing(x,-999)

xNaN =
2    NaN    5    3    NaN    4    NaN
```

Use the [rmmissing](#) function to remove missing values.

```
cleanX = rmmissing(xNaN)

cleanX =
     2     5     3     4
```

| Ignores NaNs by default (default flag is "omitnan") | Includes NaNs by default (default flag is "includenan") |
|--|--|
| max min | cov mean median std var |

| Data Type | Meaning of "Missing" |
|--------------------------------|--------------------------|
| double single | NaN |
| string array | Empty string (<missing>) |
| datetime | NaT |
| duration calendarDuration | NaN |
| categorical | <undefined> |

Interpolating Missing Data

[fillmissing](#) *Fills missing values of an array or table.*

Interpolation assuming equal spacing of observations.

Interpolation with given observation locations.

```
z = fillmissing(y,"method")
```

```
z = fillmissing(y,"method","SamplePoints",x)
```

| Method | Meaning |
|------------|---|
| "next" | The missing value is the same as the next nonmissing value in the data. |
| "previous" | The missing value is the same as the previous nonmissing value in the data. |
| "nearest" | The missing value is the same as the nearest (next or previous) nonmissing value in the data. |

| Method | Meaning |
|----------|---|
| "linear" | The missing value is the linear interpolation (average) of the previous and next nonmissing values. |
| "spline" | Cubic spline interpolation matches the derivatives of the individual interpolants at the data points. This results in an interpolant that is smooth across the whole data set. However, this can also introduce spurious oscillations in the interpolant between data points. |
| "pchip" | The cubic Hermite interpolating polynomial method forces the interpolant to maintain the same monotonicity as the data. This prevents oscillation between data points. |

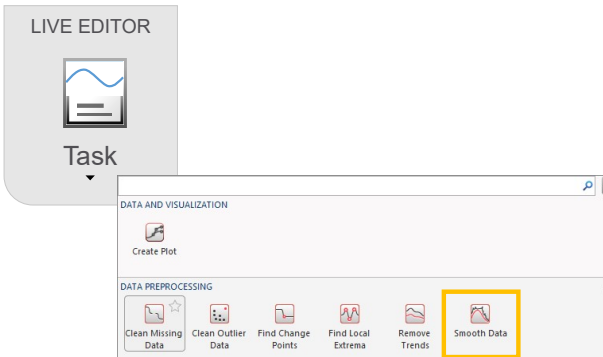
Common Data Analysis Techniques

Summary of Common Data Analysis Techniques

Summary: Common Data Analysis Techniques

Moving Window Operations

The **Smooth Data** task can be used to smooth variation or noise in data. You can add one to a script by selecting it from the **Live Editor** tab in the toolstrip.



The **Smooth Data** task uses the `smoothdata` function.

Mean calculated with a centered moving k -point window.

Mean calculated with a moving window with kb points backward and kf points forward from the current point.

Median calculated with a centered moving k -point window.

```
z = smoothdata(y,"movmean",k)

z = smoothdata(y,"movmean",[kb kf])

z = smoothdata(y,"movmedian",k)

z = smoothdata(y,"movmedian",k,"SamplePoints",x)
```

Median calculated with a centered moving k -point window using sample points defined in x .

Linear Correlation

You can investigate relationships between variables visually and computationally:

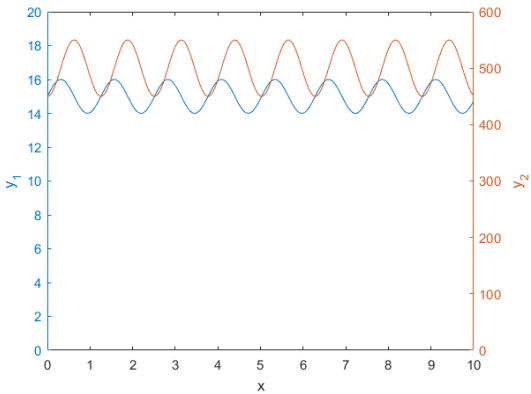
- Plot multiple series together. Use `yyaxis` to add another vertical axis to allow for different scales.
- Plot variables against each other. Use `plotmatrix` to create an array of scatter plots.
- Calculate linear correlation coefficients. Use `corrcoef` to calculate pairwise correlations.

Plot multiple series together.

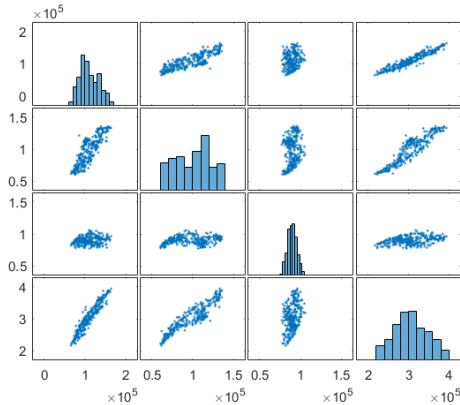
Plot variables against each other.

Calculate linear correlation coefficients.

`yyaxis left`
`plot(...)`
`yyaxis right`
`plot(...)`



`plotmatrix(data)`



`corrcoef(data)`

`ans =`

| | | | |
|--------|--------|--------|--------|
| 1.0000 | 0.8243 | 0.1300 | 0.9519 |
| 0.8243 | 1.0000 | 0.1590 | 0.9268 |
| 0.1300 | 0.1590 | 1.0000 | 0.2938 |
| 0.9519 | 0.9268 | 0.2938 | 1.0000 |

Polynomial Fitting

[polyfit](#) *Fits a polynomial to data.*

[polyval](#) *Evaluates a polynomial at specified locations.*

Simple fitting

Fit polynomial to data.

Evaluate fitted polynomial.

```
c = polyfit(x,y,n);
```

```
yfit = polyval(c,xfit);
```

Fitting with centering and scaling

Fit polynomial to data.

Evaluate fitted polynomial.

```
[c,~,scl] = polyfit(x,y,n);
```

```
yfit = polyval(c,xfit,[],scl);
```

Programming Constructs

Summary of Programming Constructs

Summary: Programming Constructs

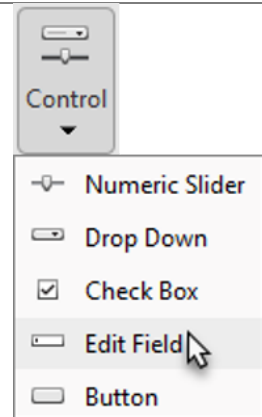
User Interaction

You can add a live control to get input from the user.

You can use [disp](#) to show output on the command window.

You can use [warning](#) and [error](#) as well.

The [msgbox](#), [errordlg](#), and [warndlg](#) functions can display messages to the user.



```
disp("Message")
```

Message

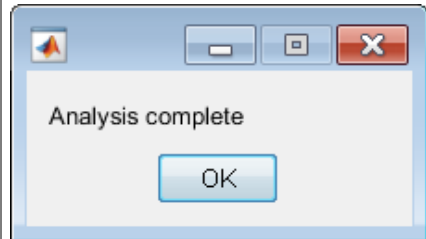
```
warning("Missing data")
```

Warning: Missing data

```
error("Missing data")
```

Missing data

```
msgbox("Analysis complete")
```



Decision Branching

The *condition_1* is evaluated as true or false.

If *condition_1* is true, then the *code_1* code block is executed.

Otherwise, the next case is tested. There can be any number of cases.

```
if condition_1
```

```
    code_1
```

```
elseif condition_2
```

```
    code_2
```

```
elseif condition_3
```

```
    code_3
```

If none of the cases are a match, then the code, `code_e`, in **else** is executed.

Always end the expression with the keyword **end**

```
else
    code_e

end
```

Evaluate `expression` to return a value.

If `expression` equals `value_1`, then `code_1` is executed. Otherwise, the next case is tested. There can be any number of cases.

If none of the cases are a match, then the code, `code_3`, in **otherwise** is executed. The **otherwise** block is optional.

Always end the expression with the keyword **end**

```
switch expression

    case value_1

        code_1

    case value_2

        code_2

    otherwise

        code_3

end
```

Determining Size



m-by-n



n-by-1



1-by-n

| <code>size(x)</code> | <code>[m n]</code> | <code>[n 1]</code> | <code>[1 n]</code> |
|------------------------|--------------------|--------------------|--------------------|
| <code>size(x,1)</code> | <i>m</i> | <i>n</i> | <i>1</i> |
| <code>size(x,2)</code> | <i>n</i> | <i>1</i> | <i>n</i> |
| <code>length(x)</code> | <i>max(m,n)</i> | <i>n</i> | <i>n</i> |
| <code>numel(x)</code> | <i>m*n</i> | <i>n</i> | <i>n</i> |

Use [size](#) to find the dimensions of a matrix.

```
s = size(prices)

s =
    19    10

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

Use [length](#) when working with vectors where one of the dimensions returned by `size` is 1.

Use [numel](#) to find the total number of elements in an array of any dimension.

```
m =  
    19  
n =  
    10  
  
m = size(prices,1)  
m =  
    19  
  
n = size(prices,2)  
n =  
    10  
  
m = length(Year)  
m =  
    19  
  
N = numel(prices)  
N =  
    190
```

For Loops

The index is defined as a vector. Note the use of the colon syntax to define the values that the index will take.

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

While Loops

The condition is a variable or expression that evaluates to true or false. While *condition* is true, code executes. Once *condition* becomes false, the loop ceases execution.

```
while condition  
    code  
end
```

Increasing Automation with Functions

Summary of Functions

Summary: Increasing Automation with Functions

Creating and Calling Functions

| Define a function | Call a function |
|---|---|
| <div>keyword<div>function</div></div> <div>function name<div>myFunc</div></div> <div><div>[o1,o2]</div><div>output</div></div> <div>=</div> <div><div>i1,i2</div><div>input</div></div> | <div>function name<div>myFunc</div></div> <div><div>[o1,o2]</div><div>output</div></div> <div>=</div> <div><div>i1,i2</div><div>input</div></div> |

Function Files

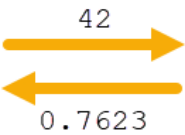
| Function Type | Function Visibility |
|--|--|
| Local functions: Functions that are defined within a script. | Visible only within the file where they are defined. |
| Functions: Functions that are defined in separate files. | Visible to other script and function files. |

Workspaces

A function maintains its own workspace to store variables created in the function body.

a = 42;

b = foo(a);



foo.mlx

```
1. function y = foo(x)
2.     a = sin(x);
3.     x = x + 1;
4.     b = sin(x);
5.     y = a*b;
6. end
```

| Base Workspace | |
|----------------|--------|
| a | 42 |
| b | 0.7623 |

| Function Workspace | |
|--------------------|---------|
| a | -0.9165 |
| b | -0.8318 |
| x | 43 |
| y | 0.7623 |

MATLAB Path and Calling Precedence

In MATLAB, there are rules for interpreting any named item. These rules are referred to as the function precedence order. Most of the common reference conflicts can be resolved using the following order:

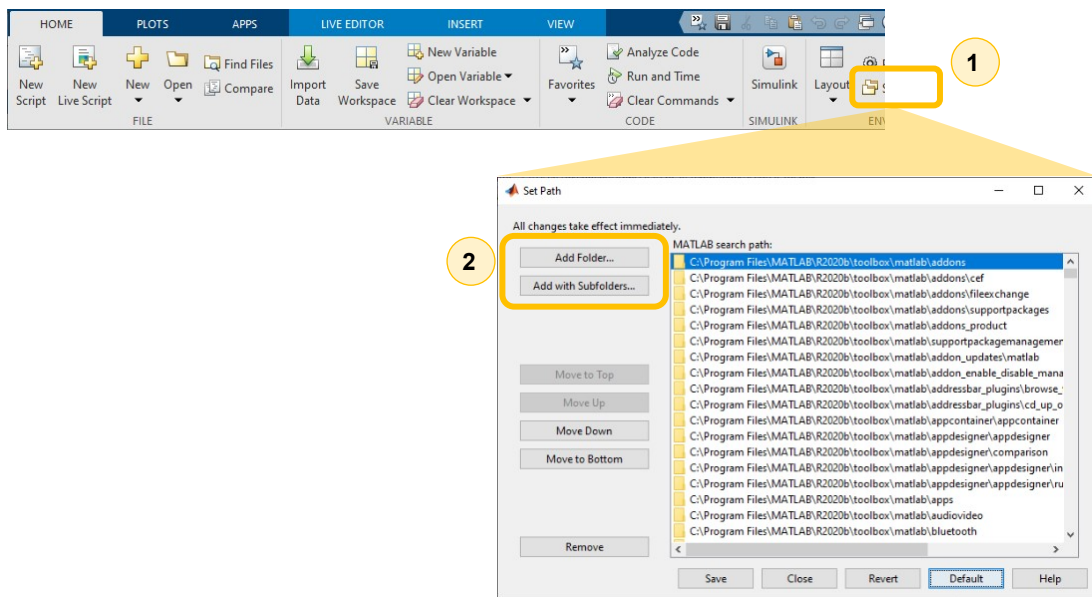
1. Variables
2. Functions defined in the current script
3. Files in the current folder
4. Files on MATLAB search path

A more comprehensive list can be found [here](#).

The search path, or *path* is a subset of all the folders in the file system. MATLAB can access all files in the folders on the search path.

To add folders to the search path:

1. On the **Home** tab, in the **Environment** section, click **Set Path**.
2. Add a single folder or a set of folders using the buttons highlighted below.



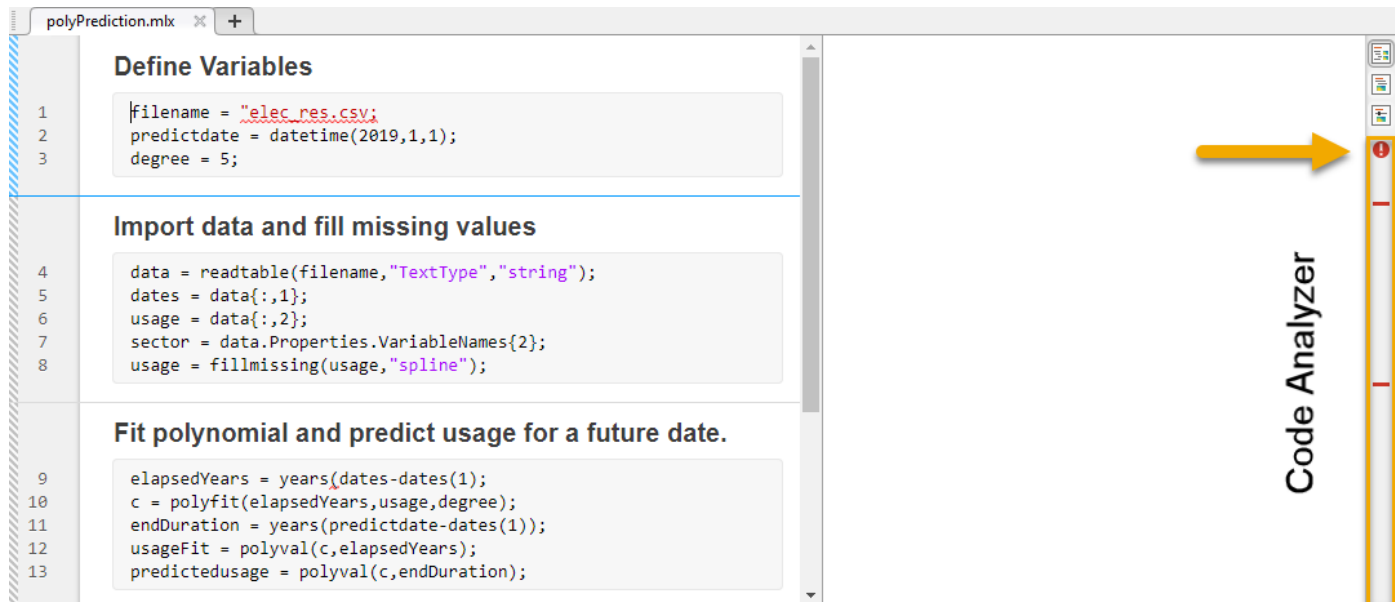
Troubleshooting Code

Summary of Troubleshooting Code

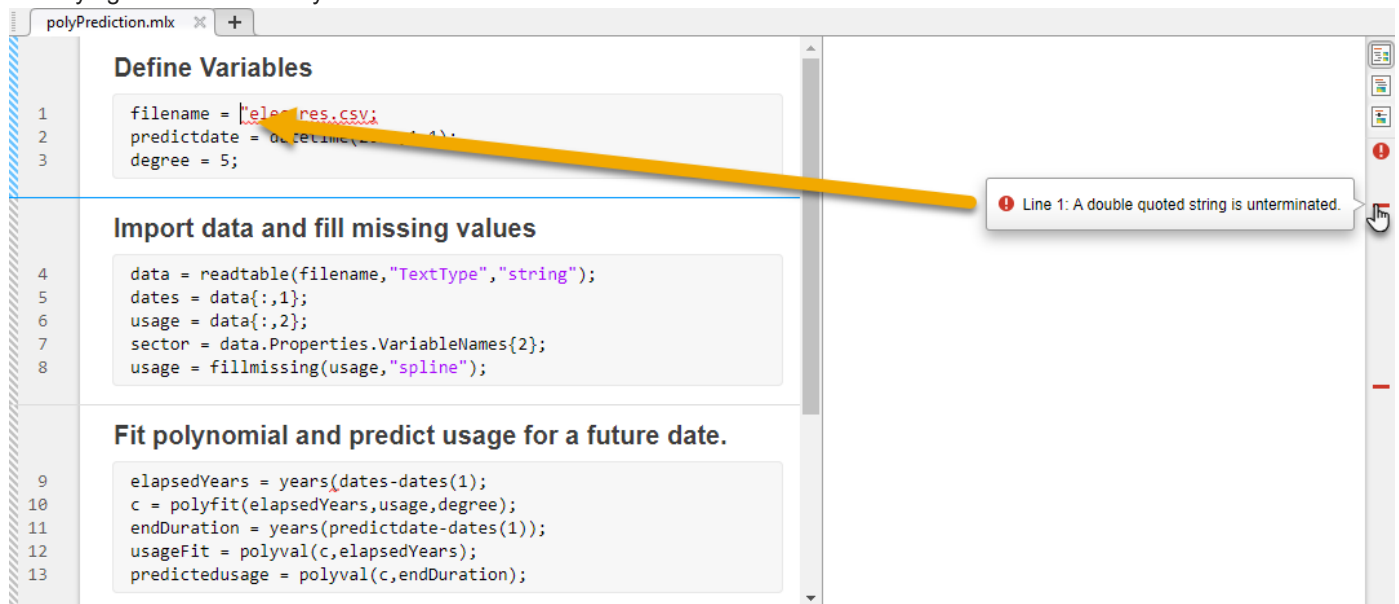
Summary: Troubleshooting Code

Code Analyzer

Use the MATLAB **Code Analyzer**. messages shown in the Editor to identify and fix syntax errors.



The small red icon at the top of the **Code Analyzer**. indicates there are errors in the script. Click on it to show red lines identifying the locations of syntax errors.



Red indicator lines in the **Code Analyzer**. identify specific syntax errors. You can mouse over one to see a description of that issue. The first indicator line describes the mistake you saw in the error message: the closing quotation mark is missing.

Clicking an indicator line puts your cursor where the error was found so you can fix it. Notice that the broken portion of the code is red and underlined.

```

1 filename = "elec_res.csv";
2 predictdate = datetime(2019,1,1);
3 degree = 5;

4 data = readtable(filename,"TextType","string");
5 dates = data{:,1};
6 usage = data{:,2};
7 sector = data.Properties.VariableNames{2};
8 usage = fillmissing(usage,"spline");

9
10 elapsedYears = years(dates-dates(1));
11 c = polyfit(elapsedYears,usage,degree);
12 endDuration = years(predictdate-dates(1));
13 usageFit = polyval(c,elapsedYears);
14 predictedusage = polyval(c,endDuration);

```

After you fix the error, the code in line 1 is no longer red and underlined. The corresponding indicator line goes away. There is still one syntax error left, though. You should fix all syntax errors flagged by the **Code Analyzer** before running your script or function.

| Icon | Meaning |
|------|---|
| | There is a potential for unexpected results or poor code performance. |
| | There are syntax errors that must be addressed. |

The **Code Analyzer** identifies both syntax errors and warnings.

Inspecting Variables

Runtime errors are bugs that aren't syntax errors.

```

1 load elecPrices
2 avgYrPrices = priceYr.*dollar2019;
3 avgElecPrice = mean(avgYrPrices);
4
5 plot(avgYrPrices)
6 yline(avgElecPrice)

```

Run time errors can produce an execution-stopping error or just be something you didn't mean to do. An effective way to troubleshoot them is to inspect variables.


```

1 load elecPrices
2 avgYrPrices = priceYr.*dollar2019;
3 avgElecPrice = mean(avgYrPrices)

```

```

avgElecPrice = 1x30
    19.7075    18.9116 ...

```

Remove semicolons to inspect the output.

```

1 load elecPrices
2 avgYrPrices = priceYr.*dollar2019;
3 avgElecPrice = mean(avgYrPrices)

```

```

avgElecPrice = 1x30
    19.7075    18.9116 ...

```

```

avgYrPrices = 30x30
    15.3220    14.7033    14.2736    13.8587    13.5127 ...
    15.7262    15.0911    14.6501    14.2243    13.8692
    16.0778    15.4285    14.9777    14.5423    14.1793
    16.0845    15.6318    15.1675    14.7322    14.3500

```

Mouse over a variable to see its size and a preview.

```

1 load elecPrices
2 avgYrPrices = priceYr.*dollar2019;
3 avgElecPrice = mean(avgYrPrices)

```

```

avgElecPrice = 1x30
    19.7075    18.9116 ...

```

Line 2: avgYrPrices = priceYr.*dollar2019;

Click on a variable to view each place where the variable is used, created, or modified. Click the gray indicator lines in the **Code Analyzer** to go directly to the line where a variable is used.

| Workspace | | |
|--------------|--------|-------|
| Name | Class | Size |
| avgElecPrice | double | 1x30 |
| avgYrPrices | double | 30x30 |
| dollar2019 | double | 1x30 |
| priceYr | double | 30x1 |

| Variables - priceYr | | | | |
|---------------------|-------------|------------|-------------|---|
| | priceYr | dollar2019 | avgYrPrices | |
| | 30x1 double | | | |
| | 1 | 2 | 3 | 4 |
| 1 | 7.8331 | | | |
| 2 | 8.0397 | | | |
| 3 | 8.2195 | | | |
| 4 | 8.3236 | | | |
| 5 | 8.3868 | | | |
| 6 | 8.4056 | | | |

Look at the variables in the **Workspace** for a preview. Double click them to inspect elements in the **Variable Editor**.

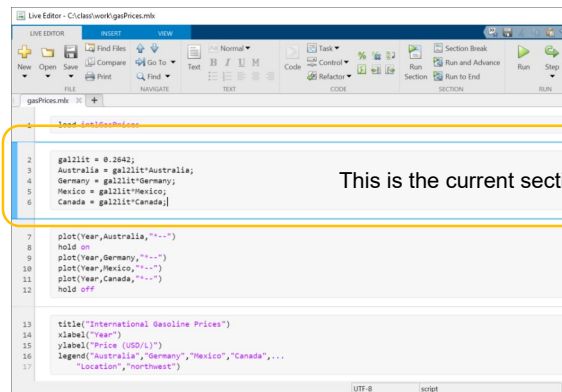
Stepping Through Code

When variables change throughout a script, you can step through your code to inspect intermediate values. You can run

section by section or set breakpoints.

Run and Advance

You can run scripts section by section. In the **Section** section of the **Live Editor** tab in the Toolstrip, you can break up your code into sections to run one at a time.



Section Break: Add a section break to create a code section.

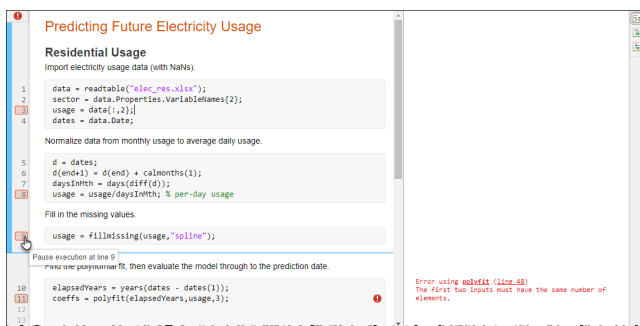


Run and Advance: Run code in the current section, then move to the next section.

Setting Breakpoints

You can also set breakpoints in scripts and functions to stop code execution before specific lines. This works particularly well with functions, where you otherwise don't have access to the workspace. Breakpoints give you access to the same tools you have in scripts for inspecting variables.

Add breakpoints by clicking line numbers.



Continue: Run code until the next breakpoint (or the end of the script).



Step: Run only the next line of code.

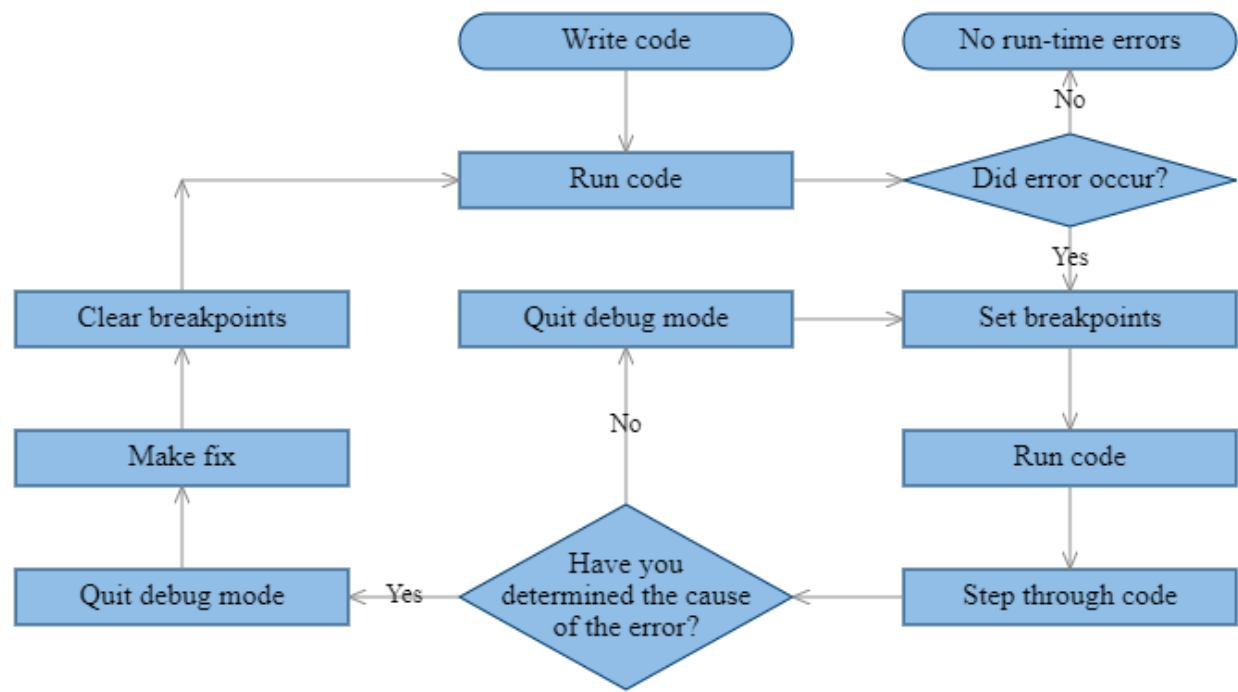


Stop: Stop code execution and exit debug mode.

Don't forget to clear your breakpoints and save your work!

A Debugging Workflow

When debugging MATLAB code, a common workflow is as follows.



Note that after you've identified and fixed any bugs, you should stop your debugging session, save your changes, and clear all breakpoints before running your code again.