DIGITAL SIGNAL PROCESSING LAB MANUAL 8

Dr. Ahsan Latif, Ms. Anosh Fatima

WINTER SEMESTER 2022

UNIVERSITY OF AGRICULTURE, FAISALABAD (UAF)

Fundamentals of MATLAB

TASK

- Open New Script from Toolbar
- Write code (all examples and Exercises) in Editor Window
- Click Run from Tool Bar to see output in command window.
- Save all MATLAB Files in Separate folder called DSP Labs
- Write name of file: your group number and lab number. E.g., G2Lab3
- Click add to the path
- Check results (output)
- Submit MATLAB code to TA online. Solve each exercise in separate script.
- ALL Code must be properly commented for submission, explaining each each step/line of exercises.

Creating, Concatenating, and Expanding Matrices

The most basic MATLAB® data structure is the matrix. A matrix is a two-dimensional, rectangular array of data elements arranged in rows and columns. The elements can be numbers, logical values (`true` or `false`), dates and times, strings, or some other MATLAB data type.

Even a single number is stored as a matrix. For example, a variable containing the value 100 is stored as a 1-by-1 matrix of type `double`.

```
A = 100;
whos A
```

```
  Name      Size            Bytes  Class     Attributes

  A         1x1                 8  double
```

Constructing a Matrix of Data

If you have a specific set of data, you can arrange the elements in a matrix using square brackets. A single row of data has spaces or commas in between the elements, and a semicolon separates the rows. For example, create a single row of four numeric elements. The size of the resulting matrix is 1-by-4, since it has one row and four columns. A matrix of this shape is often referred to as a row vector.

```
A = [12 62 93 -8]
```

```
A = 1×4

    12    62    93    -8
```

```
sz = size(A)
```

```
sz = 1×2

     1     4
```

Now create a matrix with the same numbers, but arrange them in two rows. This matrix has two rows and two columns.

```
A = [12 62; 93 -8]
```

```
A = 2×2

    12    62
    93    -8
```

```
sz = size(A)
```

```
sz = 1×2

     2     2
```

Specialized Matrix Functions

MATLAB has many functions that help create matrices with certain values or a particular structure. For example, the `zeros` and `ones` functions create matrices of all zeros or all ones. The first and second arguments of these functions are the number of rows and number of columns of the matrix, respectively.

```
A = zeros(3,2)
```

```
A = 3×2

     0     0
     0     0
     0     0
```

```
B = ones(2,4)
```

```
B = 2×4

     1     1     1     1
     1     1     1     1
```

The `diag` function places the input elements on the diagonal of a matrix. For example, create a row vector `A` containing four elements. Then, create a 4-by-4 matrix whose diagonal elements are the elements of `A`.

```
A = [12 62 93 -8];
B = diag(A)
```

```
B = 4×4

    12     0     0     0
     0    62     0     0
     0     0    93     0
     0     0     0    -8
```

Concatenating Matrices

You can also use square brackets to join existing matrices together. This way of creating a matrix is called *concatenation*. For example, concatenate two row vectors to make an even longer row vector.

```
A = ones(1,4);
B = zeros(1,4);
C = [A B]
```

```
C = 1×8

     1     1     1     1     0     0     0     0
```

To arrange `A` and `B` as two rows of a matrix, use the semicolon.

```
D = [A;B]
```

```
D = 2×4

     1     1     1     1
     0     0     0     0
```

To concatenate two matrices, they must have compatible sizes. In other words, when you concatenate matrices horizontally, they must have the same number of rows. When you concatenate them vertically, they must have the same number of columns. For example, horizontally concatenate two matrices that both have two rows.

```
A = ones(2,3)
```

```
A = 2×3

     1     1     1
     1     1     1
```

```
B = zeros(2,2)
```

```
B = 2×2

     0     0
     0     0
```

```
C = [A B]
```

```
C = 2×5

     1     1     1     0     0
     1     1     1     0     0
```

An alternative way to concatenate matrices is to use concatenation functions such as `horzcat`, which horizontally concatenates two compatible input matrices.

```
D = horzcat(A,B)
```

```
D = 2×5

     1     1     1     0     0
     1     1     1     0     0
```

Generating a Numeric Sequence

The `colon` is a handy way to create matrices whose elements are sequential and evenly spaced. For example, create a row vector whose elements are the integers from 1 to 10.

```
A = 1:10
```

```
A = 1×10

     1     2     3     4     5     6     7     8     9    10
```

You can use the colon operator to create a sequence of numbers within any range, incremented by one.

```
A = -2.5:2.5
```

```
A = 1×6

   -2.5000   -1.5000   -0.5000    0.5000    1.5000    2.5000
```

To change the value of the sequence increment, specify the increment value in between the starting and ending range values, separated by colons.

```
A = 0:2:10
```

```
A = 1×6

     0     2     4     6     8    10
```

To decrement, use a negative number.

```
A = 6:-1:0
```

```
A = 1×7

     6     5     4     3     2     1     0
```

You can also increment by noninteger values. If an increment value does not evenly partition the specified range, MATLAB automatically ends the sequence at the last value it can reach before exceeding the range.

```
A = 1:0.2:2.1
```

```
A = 1×6

    1.0000    1.2000    1.4000    1.6000    1.8000    2.0000
```

Expanding a Matrix

You can add one or more elements to a matrix by placing them outside of the existing row and column index boundaries. MATLAB automatically pads the matrix with zeros to keep it rectangular. For example, create a 2-by-3 matrix and add an additional row and column to it by inserting an element in the (3,4) position.

```
A = [10  20  30; 60  70  80]
```

```
A = 2×3

    10    20    30
    60    70    80
```

```
A(3,4) = 1
```

```
A = 3×4

    10    20    30     0
    60    70    80     0
     0     0     0     1
```

You can also expand the size by inserting a new matrix outside of the existing index ranges.

```
A(4:5,5:6) = [2 3; 4 5]
```

```
A = 5×6

    10    20    30     0     0     0
    60    70    80     0     0     0
     0     0     0     1     0     0
     0     0     0     0     2     3
     0     0     0     0     4     5
```

To expand the size of a matrix repeatedly, such as within a `for` loop, it's usually best to preallocate space for the largest matrix you anticipate creating. Without preallocation, MATLAB has to allocate memory every time the size increases, slowing down operations. For example, preallocate a matrix that holds up to 10,000 rows and 10,000 columns by initializing its elements to zero.

```
A = zeros(10000,10000);
```

If you need to preallocate additional elements later, you can expand it by assigning outside of the matrix index ranges or concatenate another preallocated matrix to `A`.

Empty Arrays

An empty array in MATLAB is an array with at least one dimension length equal to zero. Empty arrays are useful for representing the concept of "nothing" programmatically. For example, suppose you want to find all elements of a vector that are less than 0, but there are none. The `find` function returns an empty vector of indices, indicating that it couldn't find any elements less than 0.

```
A = [1 2 3 4];
ind = find(A<0)
ind =

  1x0 empty double row vector
```

Many algorithms contain function calls that can return empty arrays. It is often useful to allow empty arrays to flow through these algorithms as function arguments instead of handling them as a special case. If you do need to customize empty array handling, you can check for them using the `isempty` function.

```
TF = isempty(ind)
```

```
TF = logical
   1
```

Removing Rows or Columns from a Matrix

The easiest way to remove a row or column of a matrix is setting that row or column equal to a pair of empty square brackets `[]`. For example, create a 4-by-4 matrix and remove the second row.

```
A = magic(4)
```

```
A = 4x4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
A(2,:) = []
```

```
A = 3x4

    16     2     3    13
     9     7     6    12
     4    14    15     1
```

Now remove the third column.

```
A(:,3) = []
```

```
A = 3x3

    16     2    13
     9     7    12
     4    14     1
```

You can extend this approach to any array. For example, create a random 3-by-3-by-3 array and remove all of the elements in the first matrix of the third dimension.

```
B = rand(3,3,3);
B(:,:,1) = [];
```

Reshaping and Rearranging Arrays

Many functions in MATLAB® can take the elements of an existing array and put them in a different shape or sequence. This can be helpful for preprocessing your data for subsequent computations or analyzing the data.

Reshaping

The `reshape` function changes the size and shape of an array. For example, reshape a 3-by-4 matrix to a 2-by-6 matrix.

```
A = [1 4 7 10; 2 5 8 11; 3 6 9 12]
```

```
A = 3×4

     1     4     7    10
     2     5     8    11
     3     6     9    12
```

```
B = reshape(A,2,6)
```

```
B = 2×6

     1     3     5     7     9    11
     2     4     6     8    10    12
```

As long as the number of elements in each shape are the same, you can reshape them into an array with any number of dimensions. Using the elements from `A`, create a 2-by-2-by-3 multidimensional array.

```
C = reshape(A,2,2,3)
```

```
C = 
C(:,:,1) =

     1     3
     2     4


C(:,:,2) =

     5     7
     6     8


C(:,:,3) =

     9    11
    10    12
```

Transposing and Flipping

A common task in linear algebra is to work with the transpose of a matrix, which turns the rows into columns and the columns into rows. To do this, use the `transpose` function or the `.'` operator.

Create a 3-by-3 matrix and compute its transpose.

```
A = magic(3)
```

```
A = 3×3

     8     1     6
     3     5     7
     4     9     2
```

```
B = A.'
```

```
B = 3×3

     8     3     4
     1     5     9
     6     7     2
```

A similar operator `'` computes the conjugate transpose for complex matrices. This operation computes the complex conjugate of each element and transposes it. Create a 2-by-2 complex matrix and compute its conjugate transpose.

```
A = [1+i 1-i; -i i]
```

```
A = 2×2 complex

   1.0000 + 1.0000i   1.0000 - 1.0000i
   0.0000 - 1.0000i   0.0000 + 1.0000i
```

```
B = A'
```

```
B = 2×2 complex

   1.0000 - 1.0000i   0.0000 + 1.0000i
   1.0000 + 1.0000i   0.0000 - 1.0000i
```

`flipud` flips the rows of a matrix in an up-to-down direction, and `fliplr` flips the columns in a left-to-right direction.

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

```
A = 2×2

     1     2
     3     4
```

```
B = flipud(A)
```

```
B = 2×2

     3     4
     1     2
```

```
C = fliplr(A)
```

```
C = 2×2

     2     1
     4     3
```

Shifting and Rotating

You can shift elements of an array by a certain number of positions using the `circshift` function. For example, create a 3-by-4 matrix and shift its columns to the right by 2. The second argument `[0 2]` tells `circshift` to shift the rows 0 places and shift the columns 2 places to the right.

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

```
A = 3×4

     1     2     3     4
     5     6     7     8
     9    10    11    12
```

```
B = circshift(A,[0 2])
```

```
B = 3×4

     3     4     1     2
     7     8     5     6
    11    12     9    10
```

To shift the rows of `A` up by 1 and keep the columns in place, specify the second argument as `[-1 0]`.

```
C = circshift(A,[-1 0])
```

```
C = 3×4

     5     6     7     8
     9    10    11    12
     1     2     3     4
```

The `rot90` function can rotate a matrix counterclockwise by 90 degrees.

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

```
A = 2×2

     1     2
     3     4
```

```
B = rot90(A)
```

```
B = 2×2

     2     4
     1     3
```

If you rotate 3 more times by using the second argument to specify the number of rotations, you end up with the original matrix A.

```
C = rot90(B,3)
```

```
C = 2×2

     1     2
     3     4
```

Sorting

Sorting the data in an array is also a valuable tool, and MATLAB offers a number of approaches. For example, the `sort` function sorts the elements of each row or column of a matrix separately in ascending or descending order. Create a matrix A and sort each column of A in ascending order.

```
A = magic(4)
```

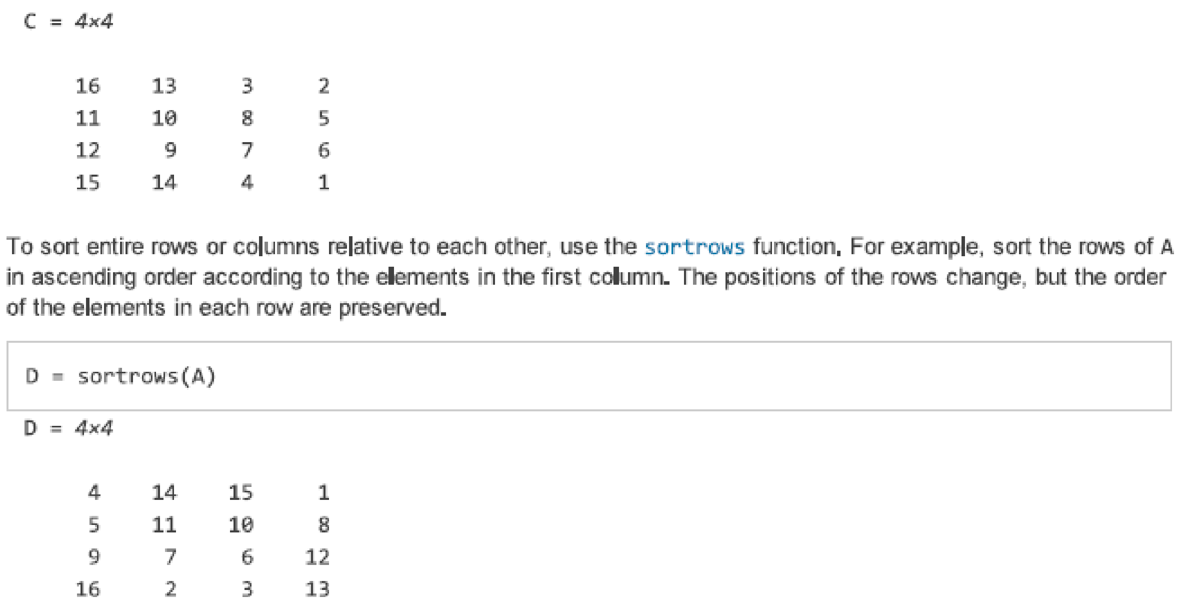
```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
B = sort(A)
```

```
B = 4×4

     4     2     3     1
     5     7     6     8
     9    11    10    12
    16    14    15    13
```

Sort each row in descending order. The second argument value 2 specifies that you want to sort row-wise.

```
C = sort(A,2,'descend')
```



```
C = 4x4

    16    13     3     2
    11    10     8     5
    12     9     7     6
    15    14     4     1
```

To sort entire rows or columns relative to each other, use the sortrows function. For example, sort the rows of A in ascending order according to the elements in the first column. The positions of the rows change, but the order of the elements in each row are preserved.

```
D = sortrows(A)
```

```
D = 4x4

     4    14    15     1
     5    11    10     8
     9     7     6    12
    16     2     3    13
```

Exercise 1

1. Create following Matrix A and display its size.

$$\begin{matrix} 1 & 5 & 10 & 4 \\ 14 & 2 & 3 & 9 \\ 6 & 7 & 8 & 1 \end{matrix}$$

2. Create a row vector B with 5 columns.
3. Create a column vector C with 5 rows.
4. Create a matrix D of size 5*5 whose diagonal elements are the elemnts of matrix B, using diag function.
5. Concatenate Matrix A with itself horizontaly.
6. Generate a numeric sequence from 10 to 20 using colon.
7. Generate a number sequence from -10 to 20 with gaps of 2.
8. Generate a number sequence from 100 to 80.
9. Generate a number sequence from 100 to 0 with gaps of 10.
10. Generate a number sequence from 10.5 to 20.5 with gaps of 1.5.
11. Add new number 11 in original matrix A in 5th row and 5th column.
12. Add these new rows in original matrix A, starting from 5th row and 5th column.

$$\begin{matrix} 2 & 0 & 3 \\ 1 & 6 & 4 \end{matrix}$$

13. Find element 3 in original matrix A using find function.

14. Remove 2nd row from original matrix A.
15. Remove 1st column from original matrix A.
16. Create Matrix Z of size 3*4 using magic function and reshape it into size 2*6.
17. Take transpose of original matrix A.
18. Flip rows of original matrix A up to down direction.
19. Flip columns of original matrix B in left to right direction.
20. Create following matrix M and shift its columns to right by 2.

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 8 & 9 & 10 & 11 \\ 5 & 6 & 7 & 12 \end{matrix}$$

21. Shift rows up by 1 and keep the columns in place using original matrix M.
22. Create a matrix of size 2*2 and rotate it on 90 degree.
23. Sort elements of original matrix M in ascending order.
24. Sort elements of original matrix M in desecending order.
25. Sort only rows of original matrix M in ascending order.
26. Sort only rows of original matrix M in descending order.

Array Indexing

In MATLAB®, there are three primary approaches to accessing array elements based on their location (index) in the array. These approaches are indexing by position, linear indexing, and logical indexing.

Indexing with Element Positions

The most common way is to explicitly specify the indices of the elements. For example, to access a single element of a matrix, specify the row number followed by the column number of the element.

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
```

```
A = 4×4

     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

```
e = A(3,2)
```

```
e = 10
```

`e` is the element in the 3,2 position (third row, second column) of `A`.

You can also reference multiple elements at a time by specifying their indices in a vector. For example, access the first and third elements of the second row of `A`.

```
r = A(2,[1 3])
```

```
r = 1×2

     5     7
```

To access elements in a range of rows or columns, use the `colon`. For example, access the elements in the first through third row and the second through fourth column of `A`.

```
r = A(1:3,2:4)
```

```
r = 3×3

     2     3     4
     6     7     8
    10    11    12
```

An alternative way to compute `r` is to use the keyword `end` to specify the second column through the last column. This approach lets you specify the last column without knowing exactly how many columns are in `A`.

```
r = A(1:3,2:end)
```

```
r = 3×3

     2     3     4
     6     7     8
    10    11    12
```

If you want to access all of the rows or columns, use the colon operator by itself. For example, return the entire third column of A.

```
r = A(:,3)
```

```
r = 4×1

     3
     7
    11
    15
```

In general, you can use indexing to access elements of any array in MATLAB regardless of its data type or dimensions. For example, directly access a column of a `datetime` array.

```
t = [datetime(2018,1:5,1); datetime(2019,1:5,1)]
```

```
t = 2x5 datetime array
   01-Jan-2018   01-Feb-2018   01-Mar-2018   01-Apr-2018   01-May-2018
   01-Jan-2019   01-Feb-2019   01-Mar-2019   01-Apr-2019   01-May-2019
```

```
march1 = t(:,3)
```

```
march1 = 2x1 datetime array
   01-Mar-2018
   01-Mar-2019
```

For higher-dimensional arrays, expand the syntax to match the array dimensions. Consider a random 3-by-3-by-3 numeric array. Access the element in the second row, third column, and first sheet of the array.

```
A = rand(3,3,3);
e = A(2,3,1)
```

```
e = 0.5469
```

For more information on working with multidimensional arrays, see Multidimensional Arrays.

Indexing with a Single Index

Another method for accessing elements of an array is to use only a single index, regardless of the size or dimensions of the array. This method is known as *linear indexing*. While MATLAB displays arrays according to their defined sizes and shapes, they are actually stored in memory as a single column of elements. A good way to visualize this concept is with a matrix. While the following array is displayed as a 3-by-3 matrix, MATLAB stores it as a single column made up of the columns of A appended one after the other. The stored vector contains the sequence of elements 12, 45, 33, 36, 29, 25, 91, 48, 11, and can be displayed using a single colon.

```
A = [12 36 91; 45 29 48; 33 25 11]
```

```
A = 3×3

    12    36    91
    45    29    48
    33    25    11
```

```
Alinear = A(:)
```

```
Alinear = 9×1

    12
    45
    33
    36
    29
    25
    91
    48
    11
```

For example, the 3,2 element of `A` is `25`, and you can access it using the syntax `A(3,2)`. You can also access this element using the syntax `A(6)`, since `25` is sixth element of the stored vector sequence.

```
e = A(3,2)
```

```
e = 25
```

```
elinear = A(6)
```

```
elinear = 25
```

While linear indexing can be less intuitive visually, it can be powerful for performing certain computations that are not dependent on the size or shape of the array. For example, you can easily sum all of the elements of `A` without having to provide a second argument to the `sum` function.

```
s = sum(A(:))
```

```
s = 330
```

The `sub2ind` and `ind2sub` functions help to convert between original array indices and their linear version. For example, compute the linear index of the 3,2 element of `A`.

```
linearidx = sub2ind(size(A),3,2)
```

```
linearidx = 6
```

Convert from the linear index back to its row and column form.

```
[row,col] = ind2sub(size(A),6)
```

```
row = 3
col = 2
```

Indexing with Logical Values

Using true and false logical indicators is another useful way to index into arrays, particularly when working with conditional statements. For example, say you want to know if the elements of a matrix `A` are less than the corresponding elements of another matrix B. The less-than operator returns a logical array whose elements are 1 when an element in A is smaller than the corresponding element in B.

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

```
A = 2×3

     1     2     6
     4     3     6
```

```
B = [0 3 7; 3 7 5]
```

```
B = 2×3

     0     3     7
     3     7     5
```

```
ind = A<B
```

```
ind = 2x3 logical array

   0   1   1
   0   1   0
```

Now that you know the locations of the elements meeting the condition, you can inspect the individual values using ind as the index array. MATLAB matches the locations of the value 1 in ind to the corresponding elements of A and B, and lists their values in a column vector.

```
Avals = A(ind)
```

```
Avals = 3×1

     2
     3
     6
```

```
Bvals = B(ind)
```

```
Bvals = 3×1

     3
     7
     7
```

MATLAB "is" functions also return logical arrays that indicate which elements of the input meet a certain condition. For example, check which elements of a string vector are missing using the ismissing function.

```
str = ["A" "B" missing "D" "E" missing];
ind = ismissing(str)
ind = 1x6 logical array

   0   0   1   0   0   1
```

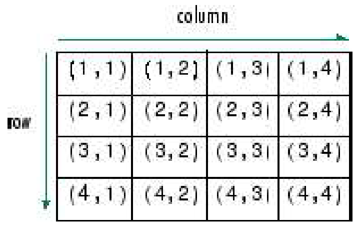
Suppose you want to find the values of the elements that are *not* missing. Use the ~ operator with the index vector `ind` to do this.

```
strvals = str(~ind)
```

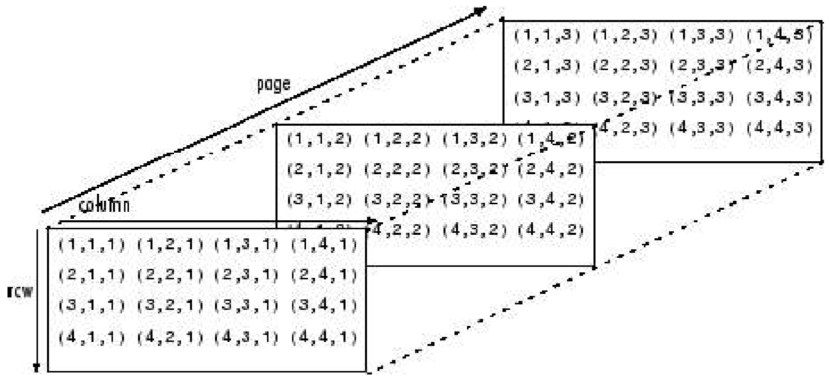
```
strvals = 1x4 string array
    "A"    "B"    "D"    "E"
```

Multidimensional Arrays

A multidimensional array in MATLAB® is an array with more than two dimensions. In a matrix, the two dimensions are represented by rows and columns.



Each element is defined by two subscripts, the row index and the column index. Multidimensional arrays are an extension of 2-D matrices and use additional subscripts for indexing. A 3-D array, for example, uses three subscripts. The first two are just like a matrix, but the third dimension represents *pages* or *sheets* of elements.



Creating Multidimensional Arrays

You can create a multidimensional array by creating a 2-D matrix first, and then extending it. For example, first define a 3-by-3 matrix as the first page in a 3-D array.

```
A = [1 2 3; 4 5 6; 7 8 9]
```

```
A = 3×3

     1     2     3
     4     5     6
     7     8     9
```

Now add a second page. To do this, assign another 3-by-3 matrix to the index value 2 in the third dimension. The syntax `A(:,:,2)` uses a colon in the first and second dimensions to include all rows and all columns from the right-hand side of the assignment.

```
A(:,:,2) = [10 11 12; 13 14 15; 16 17 18]
```

```
A =
A(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


A(:,:,2) =

    10    11    12
    13    14    15
    16    17    18
```

The `cat` function can be a useful tool for building multidimensional arrays. For example, create a new 3-D array `B` by concatenating `A` with a third page. The first argument indicates which dimension to concatenate along.

```
B = cat(3,A,[3 2 1; 0 9 8; 5 3 7])
```

```
B =
B(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


B(:,:,2) =

    10    11    12
    13    14    15
    16    17    18


B(:,:,3) =

     3     2     1
     0     9     8
     5     3     7
```

Another way to quickly expand a multidimensional array is by assigning a single element to an entire page. For example, add a fourth page to `B` that contains all zeros.

```
B(:,:,4) = 0
```

```
B =
B(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


B(:,:,2) =

    10    11    12
    13    14    15
    16    17    18


B(:,:,3) =

     3     2     1
     0     9     8
     5     3     7


B(:,:,4) =

     0     0     0
     0     0     0
     0     0     0
```

Accessing Elements

To access elements in a multidimensional array, use integer subscripts just as you would for vectors and matrices. For example, find the 1,2,2 element of A, which is in the first row, second column, and second page of A.

```
A
```

```
A =
A(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


A(:,:,2) =

    10    11    12
    13    14    15
    16    17    18
```

```
elA = A(1,2,2)
elA = 11
```

Use the index vector `[1 3]` in the second dimension to access only the first and last columns of each page of `A`.

```
C = A(:,[1 3],:)
```

```
C = 
C(:,:,1) =

     1     3
     4     6
     7     9


C(:,:,2) =

    10    12
    13    15
    16    18
```

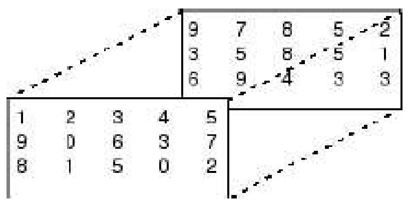
To find the second and third rows of each page, use the colon operator to create your index vector.

```
D = A(2:3,:,:)
```

```
D = 
D(:,:,1) =

     4     5     6
     7     8     9


D(:,:,2) =

    13    14    15
    16    17    18
```

Manipulating Arrays

Elements of multidimensional arrays can be moved around in many ways, similar to vectors and matrices. `reshape`, `permute`, and `squeeze` are useful functions for rearranging elements. Consider a 3-D array with two pages.



Reshaping a multidimensional array can be useful for performing certain operations or visualizing the data. Use the `reshape` function to rearrange the elements of the 3-D array into a 6-by-5 matrix.

```
A = [1 2 3 4 5; 9 0 6 3 7; 8 1 5 0 2];
A(:,:,2) = [9 7 8 5 2; 3 5 8 5 1; 6 9 4 3 3];
B = reshape(A,[6 5])
```

```
B = 6×5

     1     3     5     7     5
     9     6     7     5     5
     8     5     2     9     3
     2     4     9     8     2
     0     3     3     8     1
     1     0     6     4     3
```

`reshape` operates columnwise, creating the new matrix by taking consecutive elements down each column of `A`, starting with the first page then moving to the second page.

Permutations are used to rearrange the order of the dimensions of an array. Consider a 3-D array `M`.

```
M(:,:,1) = [1 2 3; 4 5 6; 7 8 9];
M(:,:,2) = [0 5 4; 2 7 6; 9 3 1]
```

```
M = 
M(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


M(:,:,2) =

     0     5     4
     2     7     6
     9     3     1
```

Use the `permute` function to interchange row and column subscripts on each page by specifying the order of dimensions in the second argument. The original rows of `M` are now columns, and the columns are now rows.

```
P1 = permute(M,[2 1 3])
```

```
P1 =
P1(:,:,1) =

     1     4     7
     2     5     8
     3     6     9


P1(:,:,2) =

     0     2     9
     5     7     3
     4     6     1
```

Similarly, interchange row and page subscripts of `M`.

```
P2 = permute(M,[3 2 1])
```

```
P2 =
P2(:,:,1) =

     1     2     3
     0     5     4


P2(:,:,2) =

     4     5     6
     2     7     6


P2(:,:,3) =

     7     8     9
     9     3     1
```

When working with multidimensional arrays, you might encounter one that has an unnecessary dimension of length 1. The `squeeze` function performs another type of manipulation that eliminates dimensions of length 1. For example, use the `repmat` function to create a 2-by-3-by-1-by-4 array whose elements are each 5, and whose third dimension has length 1.

```
A = repmat(5,[2 3 1 4])
```

```
A = 
A(:,:,1,1) =

     5     5     5
     5     5     5


A(:,:,1,2) =

     5     5     5
     5     5     5


A(:,:,1,3) =

     5     5     5
     5     5     5


A(:,:,1,4) =

     5     5     5
     5     5     5
```

```
szA = size(A)
```

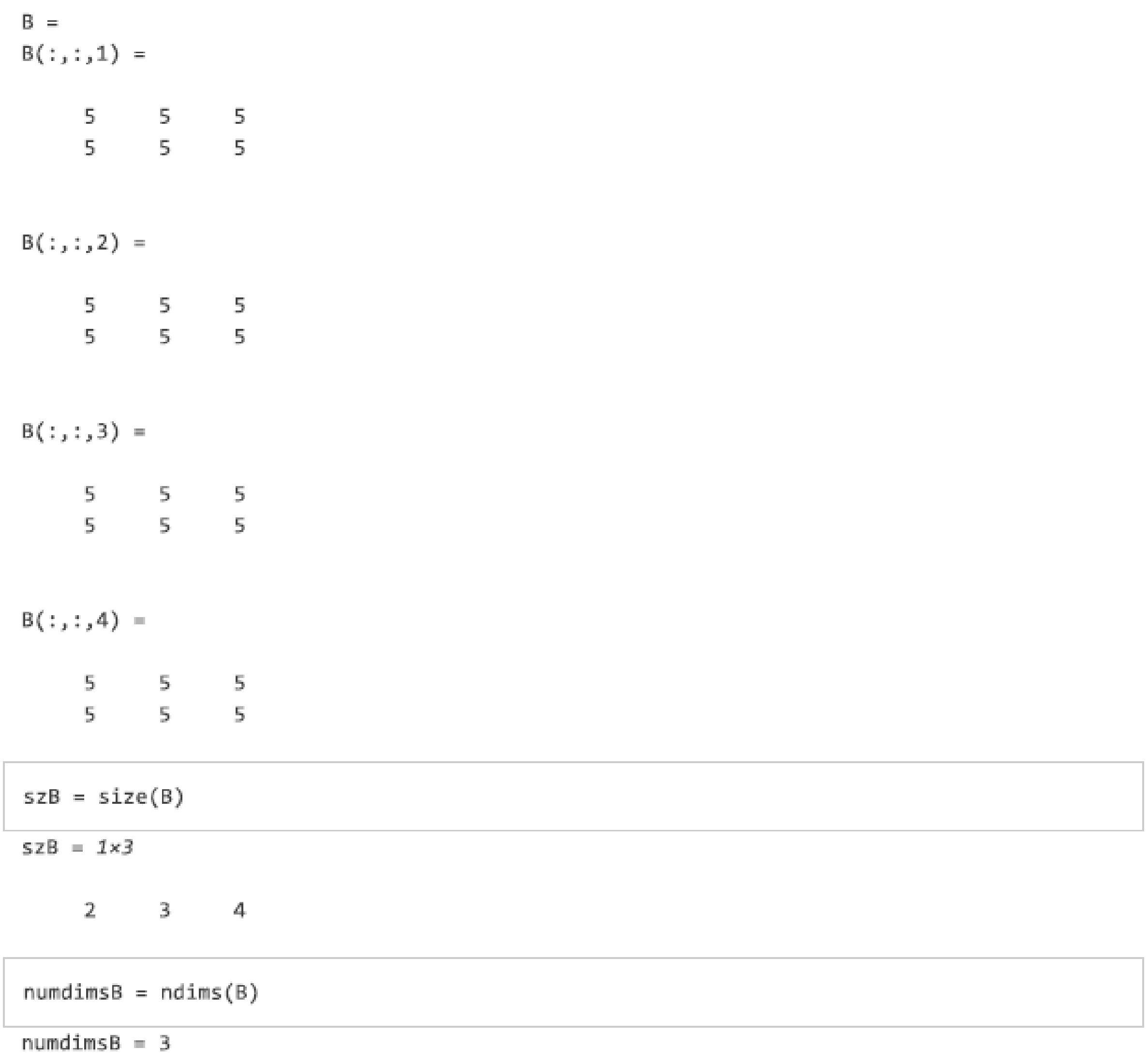
```
szA = 1×4

     2     3     1     4
```

```
numdimsA = ndims(A)
```

```
numdimsA = 4
```

Use the `squeeze` function to remove the third dimension, resulting in a 3-D array.

```
B = squeeze(A)
```



```
B =
B(:,:,1) =

     5     5     5
     5     5     5


B(:,:,2) =

     5     5     5
     5     5     5


B(:,:,3) =

     5     5     5
     5     5     5


B(:,:,4) =

     5     5     5
     5     5     5
```

```
szB = size(B)
```

```
szB = 1×3

     2     3     4
```

```
numdimsB = ndims(B)
```

```
numdimsB = 3
```

Exercise 2

Write MATLAB code for following questions:

1. Create following Matrix A, display element on 3rd row and 2nd column.

1	2	3	4
16	15	14	13
5	6	7	8
9	10	11	12

2. From original matrix A, display elements from 2nd row and 1st and 3rd column only. Out put will be 16 and 14 only.

3. Create date and time array, elements starting from (1st day of Jan to June 2020) to (1st day of July to Dec 2022).
4. Display sum of all elements of original matrix A using linear indexing.
5. Create matrix B

   2 4 1

   6 3 2

   And matrix C

   8 3 2

   3 1 5

   Output a logical array for A<B
6. Create multi dimensional array Z of size 3*3

   Page 1 elements

   1 2 3

   4 5 6

   9 8 7

   Page 2 elements

   10 11 12

   13 14 15

   18 17 16

   Page 3 elements

   19 20 21

   22 23 24

   25 26 27
7. Add page 4 in 3D array A having all elements 0.
8. Display all rows and all columns of page 2.
9. Diplay element on 2nd row, 2nd column and 3rd page.
10. Display element on 3rd row, 1st column and 2nd page.
11. Display elements of 2nd and 3rd row, all columns of 1st page.
12. Display elements of 1st and 2nd row, 2nd and 3rd columns of 2nd page.
13. Display elements of all rows and 3rd column of page 3rd.
14. Display elements of 2nd row and all columns of page 3rd.

Entering Commands

Build and run MATLAB® statements

Functions

`ans`	Most recent answer
`clc`	Clear Command Window
`diary`	Log Command Window text to file
`format`	Set Command Window output display format
`home`	Send cursor home
`iskeyword`	Determine whether input is MATLAB keyword
`more`	Control paged output in Command Window
Command Window	Select the Command Window
Command History Window	Open Command History window

Enter Statements in Command Window

As you work in MATLAB®, you can enter individual statements in the Command Window. For example, create a variable named `a` by typing this statement at the command line:

```
a = 1
```

MATLAB immediately adds variable `a` to the workspace and displays the result in the Command Window.

```
a =

     1
```

When you do not specify an output variable, MATLAB uses the variable `ans`, short for *answer*, to store the results of your calculation.

```
sin(a)
```

```
ans =

    0.8415
```

The value of `ans` changes with every command that returns an output value that is not assigned to a variable.

If you end a statement with a semicolon, MATLAB performs the computation, but suppresses the display of output in the Command Window.

```
b = 2;
```

To enter multiple statements on multiple lines before running any of the statements, use **Shift+Enter** between statements. This action is unnecessary when you enter a paired keyword statement on multiple lines, such as `for` and `end`.

You also can enter more than one statement on the same line by separating statements. To distinguish between commands, end each one with a comma or semicolon. Commands that end with a comma display their results, while commands that end with a semicolon do not. For example, enter the following three statements at the command line:

```
A = magic(5),  B = ones(5) * 4.7;  C = A./B
```

```
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

C =
    3.6170    5.1064    0.2128    1.7021    3.1915
    4.8936    1.0638    1.4894    2.9787    3.4043
    0.8511    1.2766    2.7660    4.2553    4.6809
    2.1277    2.5532    4.0426    4.4681    0.6383
    2.3404    3.8298    5.3191    0.4255    1.9149
```

MATLAB displays only the values of `A` and `C` in the Command Window.

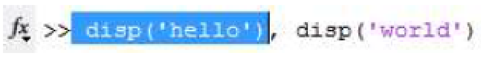
To recall previous lines in the Command Window, press the up- and down-arrow keys, ↑ and ↓. Press the arrow keys either at an empty command line or after you type the first few characters of a command. For example, to recall the command `b = 2`, type `b`, and then press the up-arrow key.

To clear a command from the Command Window without executing it, press the Escape (**Esc**) key.

You can evaluate any statement already in the Command Window. Select the statement, right-click, and then select **Evaluate Selection**.

In the Command Window, you also can execute only a portion of the code currently at the command prompt. To evaluate a portion of the entered code, select the code, and then press **Enter.**

For example, select a portion of the following code:



```
>> disp('hello'), disp('world')
```

```
hello
```

Format Output

MATLAB® displays output in both the Command Window and the Live Editor. You can format the output display using several provided options.

Format Line Spacing in Output

By default, MATLAB displays blanks lines in command window output.

You can select one of two numeric display options in MATLAB.

- `loose` — Keeps the display of blank lines (default).

```
>> x = [4/3 1.2345e-6]

x =

    1.3333    0.0000
```

- `compact` — Suppresses the display of blank lines.

```
>> x = [4/3 1.2345e-6]
x =
    1.3333    0.0000
```

To format the output display, do one of the following:

- On the **Home** tab, in the **Environment** section, click **Preferences**. Select **MATLAB** > **Command Window**, and then choose a **Numeric display** option.
- Use the `format` function at the command line, for example:

```
format loose
format compact
```

Note

Line spacing display options do not apply in the Live Editor.

Format Floating-Point Numbers

You can change the way numbers display in both the Command Window and the Live Editor. By default, MATLAB uses the short format (5-digit scaled, fixed-point values).

For example, suppose that you enter `x = [4/3 1.2345e-6]` in the Command Window. The MATLAB output display depends on the format you selected. This table shows some of the available numeric display formats, and their corresponding output.

Numeric Display Format	**Example Output**
`short` (default)	`x = 1.3333 0.0000`
`short e`	`x = 1.3333e+00 1.2345e-06`
`long`	`x = 1.333333333333333 0.000001234500000`
`+`	`x = ++`

Note

The text display format affects only how numbers are shown, not how MATLAB computes, or saves them.

To format the way numbers display, do one of the following:

- On the **Home** tab, in the **Environment** section, click **Preferences.** Select **MATLAB** > **Command Window**, and then choose a **Numeric format** option.
- Use the `format` function, for example:

  ```
  format short
  format short e
  format long
  ```

  See the `format` reference page for a list and description of all supported numeric formats.

Wrap Lines of Code to Fit Window Width

A line of code or its output can exceed the width of the Command Window, requiring you to use the horizontal scroll bar to view the entire line. To break a single line of input or output into multiple lines to fit within the current width of the Command Window:

1. On the **Home** tab, in the **Environment** section, click **Preferences.** Select **MATLAB** > **Command Window.**
2. Select **Wrap Lines.**
3. Click **OK.**

Note

Line wrapping options do not apply in the Live Editor.

Suppress Output

To suppress code output, add a semicolon (;) to the end of a command. This is useful when code generates large matrices.

Running the following code creates `A`, but does not show the resulting matrix in the Command Window or the Live Editor:

```
A = magic(100);
```

View Output by Page

Output in the Command Window might exceed the visible portion of the window. You can view the output, one screen at a time:

1. In the Command Window, type `more on` to enable paged output.
2. Type the command that generates large output.
3. View the output:
   - Advance to the next line by pressing **Enter.**
   - Advance to the next page by pressing **Space Bar.**

1. In the Command Window, type `more on` to enable paged output.
2. Type the command that generates large output.
3. View the output:
   - Advance to the next line by pressing **Enter.**
   - Advance to the next page by pressing **Space Bar.**
   - Stop displaying the output by pressing **q.**

To disable paged output, type `more off`.

Note

Paged output options do not apply in the Live Editor.

Clear the Command Window

If the Command Window seems cluttered, you can clear all the text (without clearing the workspace) by doing one of the following:

- On the **Home** tab, in the **Code** section, select **Clear Commands** > **Command Window** to clear the Command Window scroll buffer.
- Use the `clc` function to clear the Command Window scroll buffer.
- Use the `home` function to clear your current view of the Command Window, without clearing the scroll buffer.

Call Functions

These examples show how to call a MATLAB® function. To run the examples, you must first create numeric arrays `A` and `B`, such as:

```
A = [1 3 5];
B = [10 6 4];
```

Enclose inputs to functions in parentheses:

```
max(A)
```

Separate multiple inputs with commas:

```
max(A,B)
```

Store output from a function by assigning it to a variable:

```
maxA = max(A)
```

Enclose multiple outputs in square brackets:

```
[maxA, location] = max(A)
```

Call a function that does not require any inputs, and does not return any outputs, by typing only the function name:

```
clc
```

Enclose text inputs in single quotation marks:

```
disp('hello world')
```

Continue Long Statements on Multiple Lines

This example shows how to continue a statement to the next line using ellipsis (`...`).

```
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 ...
      - 1/6 + 1/7 - 1/8 + 1/9;
```

Build a long character vector by concatenating shorter vectors together:

```
mytext = ['Accelerating the pace of ' ...
    'engineering and science'];
```

The start and end quotation marks for a character vector must appear on the same line. For example, this code returns an error, because each line contains only one quotation mark:

```
mytext = 'Accelerating the pace of ...
            engineering and science'
```

An ellipsis outside a quoted text is equivalent to a space. For example,

```
x = [1.23...
4.56];
```

is the same as

```
x = [1.23 4.56];
```

Stop Execution

To stop execution of a MATLAB® command, press **Ctrl+C** or **Ctrl+Break**.

On Apple Macintosh platforms, you also can use **Command+.** (the Command key and the period key).

Ctrl+C does not always stop execution for files that run a long time, or that call built-ins or MEX-files that run a long time. If you experience this problem, include a `drawnow`, `pause`, or `getframe` function in your file, for example, within a large loop.

Also, **Ctrl+C** might be less responsive if you start MATLAB with the `-nodesktop` option.

Note

For certain operations, stopping the program might generate errors in the Command Window.

Rerun Favorite Commands

Create and Run Favorite Commands

MATLAB® favorite commands (previously called command shortcuts) are an easy way to run a group of MATLAB language statements that you use regularly. For example, you can use a favorite command to set up your environment when you start working, or to set the same properties for figures you create.

To create a favorite command:

1. On the **Home** tab, in the **Code** section, click **Favorites** and then click **New Favorite.** The Favorite Command Editor dialog box opens.
2. In the **Label** field, enter a name for the favorite command. For this example, enter `Setup Workspace`.
3. In the **Code** field, type the statements you want the favorite command to run. You also can drag and drop statements from the Command Window, the Command History Window, or a file. MATLAB automatically removes any command prompts (`>>`) from the **Code** field when you save the favorite command.

   For example, enter these statements:

   ```
   format compact
   clear
   workspace
   filebrowser
   clc
   ```

4. In the **Category** field, type the name of a new category or select an existing category from the drop-down list. If you leave this field blank, the favorite command appears in the default **Favorite Commands** category.
5. In the **Icon** field, select an icon.
6. To add the favorite command to the quick access toolbar, select both the **Add to quick access toolbar** and **Show label on quick access toolbar** options.
7. To run the statements in the **Code** section and ensure they perform the desired actions, click **Test**.
8. When you are done configuring the favorite command, click **Save.**

To run a favorite command, on the **Home** tab, click **Favorites** and then click the icon for the desired favorite command. All the statements in the **Code** field of the Favorite Command Editor execute as if you ran those statements from the Command Window, although they do not appear in the Command History window.

To edit a favorite command, click the ✎ icon to the right of the favorite command. To delete a favorite command, click the 🗑 icon to the right of the favorite command. You also can right-click the favorite command and select **Edit Favorite** or **Delete Favorite.**

Organize Favorite Commands

You can organize your favorite commands by storing them in different categories.

To create a new category:

1. On the **Home** tab, in the **Code** section, click **Favorites** and then click **New Category.** The Favorite Category Editor dialog box opens.
2. In the **Label** field, enter a name for the category. For this example, enter `My Favorite Favorites`.
3. In the **Icon** field, select an icon.

4. To add the category to the quick access toolbar, select both the **Add to quick access toolbar** and **Show label on quick access toolbar** options.
5. Click **Save.**

To move a category up or down in the list of categories, or to move a favorite command within a category, drag the category or favorite command to the desired location. You also can use the ▲ and ▼ buttons to the right of the category.

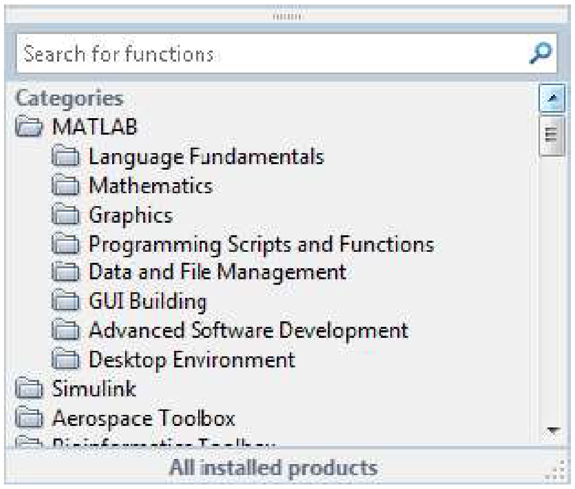
To change whether a single category or favorite command appears in the quick access bar, click the and icons to the right of the category or favorite command. To add all favorite commands to the quick access bar, on the **Home** tab, right-click **Favorites** and select `Add to quick access toolbar`.

To further configure which favorite commands and categories appear in the quick access bar, on the **Home** tab, in the **Code** section, click **Favorites** and then click **Quick Access.** Adding and configuring favorite commands and categories in the quick access bar is not supported in MATLAB Online™.

Find Functions to Use

This example shows how to find the name and description of a MathWorks® function from the Command Window or Editor using the Function browser. The Function browser is not supported in the Live Editor.

1. Click the Browse for functions button, *fx*. In the Command Window, this button is to the left of the prompt. In the Editor, the button is on the **Editor** tab, in the **Edit** section. The Function browser opens.

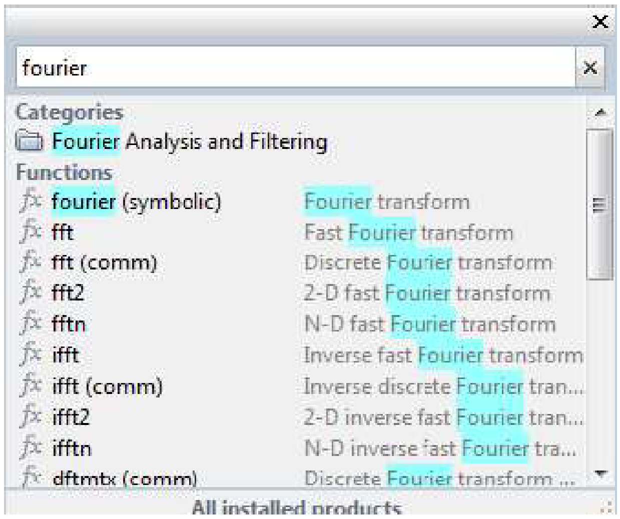


Tip

The Function browser closes when you move the pointer outside of it. To keep the browser open, drag it by the top edge to a different location.

2. Optionally, select a subset of products to display in the list. Click the product area at the bottom of the browser (where the text **All installed products** appears by default), and then set the **Selected Products** preference and click **OK.** This preference also applies to the Help browser.

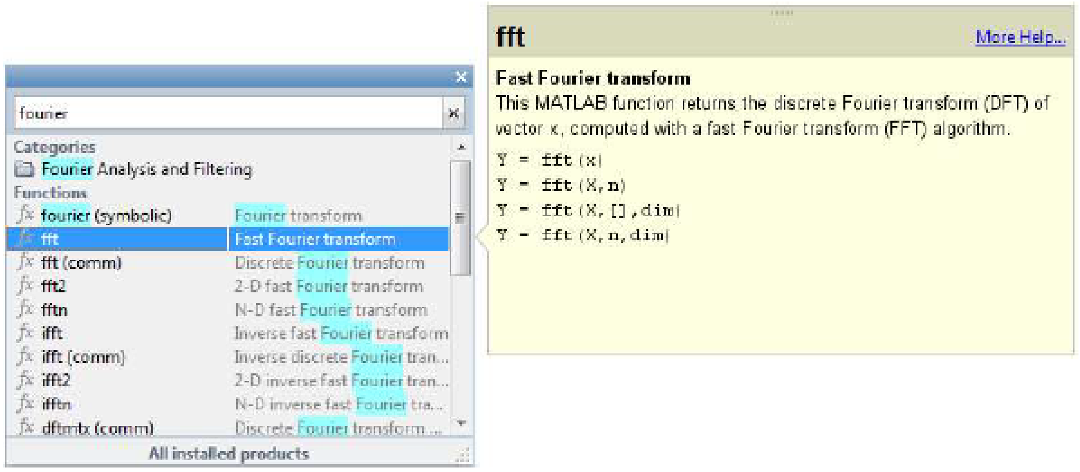
3. Find functions by browsing the list or by typing a search term. For example, search for the term *fourier*.


fourier
Categories
Fourier Analysis and Filtering
Functions
fourier (symbolic) Fourier transform
fft Fast Fourier transform
fft (comm) Discrete Fourier transform
fft2 2-D fast Fourier transform
fftn N-D fast Fourier transform
ifft Inverse fast Fourier transform
ifft (comm) Inverse discrete Fourier tran...
ifft2 2-D inverse fast Fourier tran...
ifftn N-D inverse fast Fourier tra...
dftmtx (comm) Discrete Fourier transform ...
All installed products

In the search results, a parenthetical term after a function name indicates either that the function is in a product folder other than MATLAB®, or that there are multiple functions with the same name. For example, `fft (comm)` corresponds to the `fft` function in the Communications Toolbox™ folder.

4. Select a function that you would like to use or learn more about, as follows.

   - Insert the function name into the current window by double-clicking the name. Alternatively, drag and drop the function name into any tool or application.
   - View syntax information for the function by single-clicking its name. A brief description for each of the syntax options displays in a yellow pop-up window.



fourier (symbolic)	Fourier transform
fft	Fast Fourier transform
fft (comm)	Discrete Fourier transform
fft2	2-D fast Fourier transform
fftn	N-D fast Fourier transform
ifft	Inverse fast Fourier transform
ifft (comm)	Inverse discrete Fourier tran...
ifft2	2-D inverse fast Fourier tran...
ifftn	N-D inverse fast Fourier tra...
dftmtx (comm)	Discrete Fourier transform ...

Fast Fourier transform

This MATLAB function returns the discrete Fourier transform (DFT) of vector x, computed with a fast Fourier transform (FFT) algorithm.

```
Y = fft(x)
Y = fft(X,n)
Y = fft(X,[],dim)
Y = fft(X,n,dim)
```

Tip

The pop-up window automatically closes when you move your pointer to a new item in the results list. To keep the pop-up window open, drag it by the top edge to a different location.

You can change the font that the Function browser uses by setting preferences. On the **Home** tab, in the **Environment** section, select **Preferences** > **Fonts**. By default, the Function browser uses the desktop text font and the pop-up window uses the Profiler font.