More on the Sum Function and Other MATLAB Functions

Leigh J. Little SUNY Brockport September 30, 2020

# Contents



# 1 The sum Function

We have seen that the sum function can be used to sum the elements of a vector.

```
>> x = [3 -1 7 4 -2]
x =
     3    -1     7     4    -2
>> sum(x)
ans =
    11
```

It doesn't matter what the orientation of the vector is.

What about a matrix? What happens if we send in a matrix as the input to the sum function?

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
    1     2     3
    4     5     6
    7     8     9
>> sum(A)
ans =
    12    15    18
```

Note that the output is different. If the input to the sum function is a matrix, the output will be the sum of the elements down the columns.

This behavior is typical of many MATLAB functions. The output of a function can depend on the type of input.

## 1.1 Another Way to Use the (:) Operator

We have seen that we can use the (:) operator to create array indices. However, there is another very handy use for this operator. Enter the following:

```
>> x = [3 -1 7 4 -2]
x =
     3    -1     7     4    -2
>> x = x(:)
x =
    3
   -1
    7
    4
   -2
```

Here we see that the (:) operator had the same effect as the apostrophe transpose operator.

Consider the following case:

```
>> y = [1;2;3]
y =
     1
     2
     3
>> y = y(:)
y =
     1
     2
     3
```

In this case the (:) operator had no effect.

What about a matrix?

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
>> A = A(:)
A =
     1
     4
     7
     2
     5
     8
     3
     6
     9
```

In this case, the output is a single column. The (:) operator reshaped `A` by stacking the columns of `A` in order.

This experiment shows the following:

- If a vector is a column vector, the (:) operator does nothing.
- If a vector is a row vector, the (:) operator transposes the vector.
- It applied to a matrix, the (:) operator reshapes the matrix column wise into one long vector.

## 1.2 Back to the `sum` Function for a Matrix

Recall that when the input is a matrix, the `sum` function outputs a row vector containing the column sums of the matrix. What if we actually want to sum all of the elements of a matrix? We have two ways to do this:

```
>> A = [1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
>> sum(sum(A))
ans =
    45
>> sum(A(:))
ans =
    45
```

The first version sums the columns, then sends that output to the sum function again to sum the elements of the resulting row vector. The second version stacks the columns of `A`, then sums this column.

The syntax

```
>> sum(sum(A))
```

is very important. Here we are sending the output of one function `sum(A)` directly as input to another function. In mathematics we would refer to this as *function composition*. In programming, this is referred to as *inlining* functions.

## 2 The `abs` Function

The `abs` function will take the absolute values of the input variable.

```
>> x = [3 -1 7 4 -2]
x =
    3    -1     7     4    -2
>> abs(x)
ans =
    3     1     7     4     2
```

## 3 The `min` and `max` Functions

As their name implies, these functions will find the minimum or maximum elements of a vector or matrix. However, there is more to these functions than it initially appears. Suppose we have the vector `x` above and we compute its maximum value:

```
>> x = [3 -1 7 4 -2]
x =
    3    -1     7     4    -2
>> max(x)
ans =
    7
```

Not surprisingly, this gives 7 as the output because 7 is the largest element of `x`. We can also assign this output to some new variable:

```
>> x = [3 -1 7 4 -2]
x =
    3    -1     7     4    -2
>> xmax = max(x)
xmax =
    7
```

However, there is another way to call the `max` function. Consider the following:

```
>> x = [3 -1 7 4 -2]
x =
    3    -1     7     4    -2
>> [xmax,locmax] = max(x)
xmax =
     7
locmax =
     3
```

In this example, the `max` function is returning *two* outputs. The first of these is the maximum of `x`. What about the second? The `locmax` variable contains the location of the maximum in the vector `x`.

```
>> x(locmax)
ans =
    7
```

It turns out that in many cases, the location of the maximum value is of more interest than the maximum value itself. In the event that the maximum occurs more than once, only the location of the first occurrence of the maximum is given.

```
>> x = [3 -1 7 4 7]
x =
    3    -1     7     4    7
>> [xmax,maxloc] = max(x)
xmax =
     7
locmax =
     3
```

This behavior of the `max` function is typical of many MATLAB functions. We say that the `max` function has a *variable number of outputs*. If we only want the maximum value, we can do

```
>> y = [5 3 -2  9  6]
y =
    5     3    -2    9     6
>> ymax = max(y)
ymax =
    9
```

If we want both the maximum value and its location, we can do

```
>> [ymax,locmax] = max(y)
ymax =
    9
locmax =
    4
```

What if we *only* want the location of the maximum value? In this case, we need to call the `max` function in an unusual way. We need a dummy placeholder in the first position. We use the `~` operator as this placeholder.

```
>> [~,locmax] = max(y)
locmax =
    4
```

## 3.1 The `max` Function and Matrices

How does the `max` function behave when the input is a matrix?

```
>> A = [4 8 1; 9 2 -1; 3 2 7]
A =
    4     8     1
```

```
    9     2    -1
    3     2     7
>> [amax,locmax] = max(A)
amax =
    9     8     7
locmax =
    2     1     3
```

In this case, the `max` function finds the maximum of each column of `A` and returns these values in the row vector `amax`. Similarly, the `locmax` variable contains the rows where each maximum value was located. For example, the maximum value of column 2 is 8 and it appears in row 1.

As with vectors, if we only want the locations of the maximum values, we can use the `~` as a placeholder.

```
>> A = [4 8 1; 9 2 -1; 3 2 7]
A =
    4     8     1
    9     2    -1
    3     2     7
>> [~,locmax] = max(A)
locmax =
    2     1     3
```

# 4 The `sort` Function

It is frequently necessary to sort the elements of a vector. Consider

```
>> x = [3 -1 7 4 -2]
x =
    3    -1     7     4    -2
>> sort(x)
ans =
    -2    -1     3     4     7
```

We can see that this function sorts the elements of the vector in increasing order.

What if we need to sort the elements of the vector in decreasing order? In this case, we need to send in another input to the `sort` function. Functions can not only have a variable number of outputs, they can have variable numbers of *inputs*.

```
>> sort(x,'descend')
ans =
     7     4     3    -1    -2
```

We have already seen variable numbers of inputs before when we used the norm function.

- `norm(x)` - compute the 2-norm of `x`.
- `norm(x,1)` - compute the 1-norm of `x`.
- `norm(x,'Inf')` - compute the $\infty$-norm of `x`.

## 4.1 How was the Vector Sorted?

As with the `max` function, the `sort` function has an optional second output. Consider

```
>> x = [3 -1 7 4 -2]
x =
    3    -1     7     4    -2
>> [sx,ix] = sort(x)
sx =
    -2    -1     3     4     7
ix =
     5     2     1     4     3
```

The variable `sx` contains the elements of `x` sorted in increasing order. What about the `ix` variable? This variable is a set of subscripts. They indicate the original locations of the elements of `x`. For example,

```
ix(1) = 5
```

This means that the element `sx(1)` was originally in position 5 of `x`. Similarly,

```
ix(3) = 1    -------->  element sx(3) was originally in position 1.
```

If we do

```
>> x(ix)
ans =
    -2    -1     3     4     7
```

We see that the `ix` vector subscript will sort the elements of `x` in increasing order.

## 4.2 Why Keep Track?

Why would we want to keep track of how a vector was sorted? Consider the following example. Suppose we use the `rand` function to randomly generate some $x$ and $y$-coordinates that we want to plot

```
>> x = rand(20,1);
>> y = rand(20,1);
>> plot(x,y)
>> title('Plot of random coordinates')
```

The resulting plot is shown in Figure 1 (note that your plot may look different since the points are random). You can see that the plot is a jumbled mess. This is because the points are plotted in the order that they

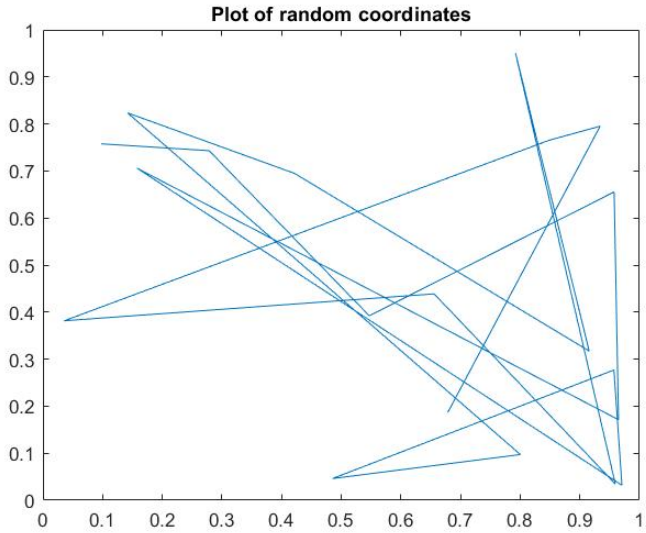


Figure 1: Plot of random coordinates.

appear in the vectors `x` and `y`. The elements of `x` are not sorted, so the connecting lines go back and forth all over the place.

If we want this plot to look more like a normal plot of a function, we need to sort the values in `x`. However, we also need to keep the pairing of the elements in `x` and `y`. We need the elements of `y` to be rearranged in the same manner that the `x` values were. We can use the optional indexing vector from the `sort` function to do this.

```
>> [x,ix] = sort(x);
>> y = y(ix);          % Rearrange the values of y the same way as x was sorted.
>> plot(x,y)
>> title('Plot of random coordinates')
```

Now the plot looks more like the plot of a standard 2-dimensional table of values.

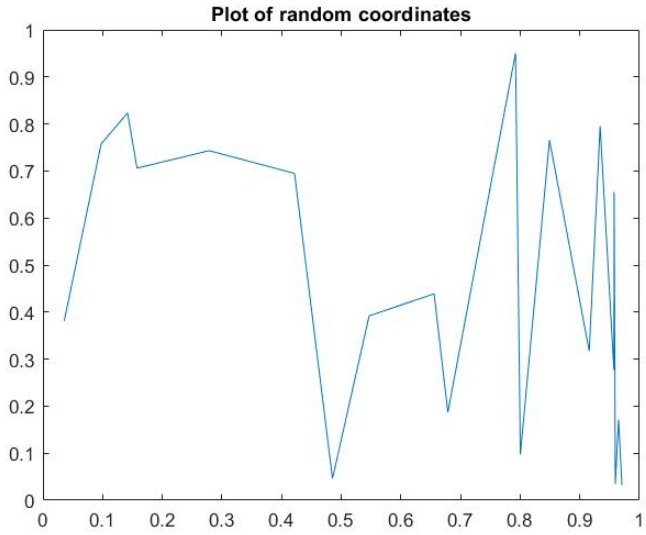


Figure 2: Plot with $x$-coordinates sorted and the $y$-coordinates sorted the same way.

# 5 Keep or Overwrite?

Consider the three ways of calling the sort function below:

```
>> sort(x)
>> x = sort(x);
>> y = sorx(x);
```

What is the difference between these?

- `sort(x);` - This will sort the elements of `x` and store the output in the generic `ans` variable.
- `x = sort(x);` - This will sort the elements of `x` and overwrite the original `x` with the sorted values.
- `y = sort(x);` - This will sort the elements of `x` and store the output in the vector `y`. The original `x` values remain unchanged.

You have many choices as to how you store the output from your function calls. Which approach you decide to use often depends on what you want to do with the data and whether or not you will need the data in its original form. This is normally a case-by-case situation.

# 6 The `mean` and `std` Commands

These commands will compute the mean and standard deviation of the elements of a vector. If the input is a matrix, these will compute the mean and standard deviation of the columns of the matrix.

```
>> A = rand(3)                  % Generate a random 3x3 matrix.
A =
    0.9390    0.6225    0.3012
```

```
    0.8759    0.5870    0.4709
    0.5502    0.2077    0.2305
>> mean(A)
ans =
    0.7884    0.4724    0.3342
>> std(A)
ans =
    0.2087    0.2299    0.1236
```

# 7 The `find` Command

The `find` command is one of the most powerful commands in MATLAB. This command is used to locate elements of a vector or matrix that satisfy a specified critera. For example,

```
>> x = [1 6 9 3 4 1 8]
x =
    1     6     9     3     4     1     8
>> ii = find(x > 5)
ii =
    2     3     7    % Subscripts of elements of x that are greater than 5.
>> x(ii)
ans =
    6     9     8    % Actual elements of x that are greater than 5.
                     % x(2), x(3), x(7)
```

Here, we have asked the `find` command to locate all elements of `x` that are greater than 5. The output from a `find` command is always a set of subscripts. These are the subscripts of the elements that satisfy the `find` criteria. In the event that no elements satisfy the criteria, the output from `find` will be an empty vector, for example

```
>> ii = find(x < 0)
ii =
10 empty double row vector
```

The `find` command is locating all negative elements in the vector `x`. Because all the elements are positive, the output (`ii`) is an empty vector.

As another example, consider a recent homework problem. Generate a vector of length 100 and determine the number of elements that are less than 0.5 or greater than or equal to 0.5. This can quickly be done using a combination of the `find` command and the `length` command.

```
>> x = rand(100,1);
>> ngreater = length(find(x >= 0.5))
ngreater =
    45
>> nless = length(find(x < 0.5))
nless =
    55
```

We will have many uses for the `find` command, but for now here are some examples:

- `find(abs(x) > 10)` - find all elements with an absolute value greater than 10.
- `find(mod(x,2) == 1)` - find all elements that are odd.
- `find(x > mean(x))` - find all elements greater than the mean of the vector.