Lecture `matlab1`: Introduction to MATLAB

August 22, 2022

Summary: Fundamental MATLAB evaluations and assignments. Vectors, matrices, function evaluations. `sum`, `prod`, `:` operators. Array indexing.

References: Appendix B of T. Sauer's textbook *Numerical Analysis*; Chapter 1 of Cleve Moler's textbook *Numerical Computing with MATLAB*; Monika Nitsche's tutorial (posted on the `UNM Learn` page)

History

MATLAB got its start at UNM.

> *When we were developing EISPACK and LINPACK, I was a math professor at the University of New Mexico, teaching numerical analysis and matrix theory. I wanted my students to be able to use our new packages without writing Fortran programs, so I studied a book by Niklaus Wirth to learn about parsing computer languages.*
>
> *In the late 1970s, following Wirth's methodology, I used Fortran and portions of LINPACK and EISPACK to develop the first version of MATLAB. The only data type was "matrix." From* THE ORIGINS OF MATLAB*, by Cleve Moler, MathWorks.*

Basic operations and functions

MATLAB or MATrix LABoratory is a computer program used to perform (mostly) numerical computations. A strength of MATLAB is its ability to perform matrix and vector operations with easy–to–understand commands. It is in widespread use, and also used to perform non–numerical computations. In this course, we will use MATLAB for *scientific computing*, i.e. for performing numerical computations that arise in scientific applications. Before considering the mathematical tools we'll develop, let's first concentrate on familiarizing ourselves with MATLAB.

We begin with basic operations. MATLAB can be used as a basic calculator. For example, consider the following commands entered in the *command window* at the prompt.

```
>> format compact
>> 3+4
ans =
     7
>> 5.5-7
ans =
   -1.5000
>> 3*3.5
ans =
   10.5000
```

```
>> 1/2
ans =
    0.5000
```

You don't need the initial `format compact` statement, but it makes the output single, rather than double, spaced. To get single-spaced outputs, we've used this command in all that follows (once entered, it "stays on").

Let us now look at the basic data structure in MATLAB: the *vector*. In MATLAB, vectors can be formed with square brackets, `[, ]`. For example, `[7 5]` and `[7,5]` both create the 1×2 vector $[7\ 5]$. The vector can be defined as a one–dimensional version of an *array*, i.e. a collection of numbers. A two–dimensional array of numbers is usually called a *matrix* in mathematics. Indeed, in MATLAB, vectors are simply one–dimensional arrays or matrices.

Elements in the same row are separated either by whitespace or by a comma (e.g. `[7 5]` and `[7,5]`). To form an array, we separate the rows by semicolons. For example, to form the matrix

$$\begin{pmatrix} 3 & 8 \\ 4 & 10 \end{pmatrix},$$

into MATLAB type the following.

```
>> [3 8; 4 10]
ans =
     3     8
     4    10
```

In order to define variables, we make use of the `=` operator. For example, to define `x` to be $\frac{34}{7}$, type

```
>> x = 34/7
x =
    4.8571
```

Now every time we type `x` into MATLAB, it will be as if we had typed in `34/7`. Note that MATLAB automatically converts the fraction $\frac{34}{7}$ into its corresponding decimal, or *floating–point* representation. We will discuss this conversion later in the course.

Array multiplication: `*`, `.*`, `^`, and `.^`

As mentioned, in MATLAB the arithmetic operation of multiplication between scalar values is implemented via the asterisk operator `*`. However, the situation is more complicated for matrices, for which we have the concept of matrix multiplication: given two matrices A and B, we can form their product AB, provided that the number of columns in A equals the number of rows in B. Now suppose that we have two 2×2 matrices A and B, given by

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}.$$

There are two types of multiplication between these matrices. The first type is the usual mathematical matrix multiplication. That is, for $C = AB$ we have

$$C = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}. \tag{1}$$

An alternative is element–by–element multiplication (called the *Hadamard product*), defined by

$$D = \begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \end{pmatrix}. \tag{2}$$

Both multiplications are well–defined operations on square matrices of the same dimension. How does Matlab differentiate between these multiplications? The convention is that the asterisk operator `*` yields matrix multiplication (i.e. matrix C). The operator used to denote elementwise multiplication is still the asterisk, but with a preceding period: `.*`. For example, take A and B to be

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 0 & 2 \\ 0 & 0 \end{pmatrix}.$$

Then according to equations (1) and (2), the matrices C and D are

$$C = \begin{pmatrix} 0 & 2 \\ 0 & 0 \end{pmatrix}, \qquad D = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

The corresponding Matlab code for these products is as follows.

```
>> A = [1 0; 0 1]
A =
     1     0
     0     1
>> B = [0 2; 0 0]
B =
     0     2
     0     0
>> C = A*B
C =
     0     2
     0     0
>> D = A.*B
D =
     0     0
     0     0
```

Depending on the array structures, one or both of $*$ and $.*$ might not be defined. For example, consider now

```
>> A = [1 0; 0 1]
A =
     1     0
     0     1
>> B = [0 2 1; 0 0 0]
B =
     0     2     1
     0     0     0
>> C = A*B
C =
     0     2     1
     0     0     0
>> D = A.*B
Matrix dimensions must agree.
```

Can you think of arrays `A` and `B` for which `A*B` is not defined, but `A.*B` is define? How about the following example.

```
>> A = [1 0 3; 1 0 1]
A =
     1     0     3
     1     0     1
>> B = [0 2 1; 0 0 0]
B =
     0     2     1
     0     0     0
>> C = A*B
Error using  *
Incorrect dimensions for matrix multiplication. Check that the number of columns in the first
matrix matches the number of rows in the second matrix. To perform elementwise multiplication,
use '.*'.
>> D = A.*B
D =
     0     0     3
     0     0     0
```

The convention of preceding an operator with a period to denote elementwise operations extends to the exponent operator. In Matlab, exponentiation is implemented via the caret mark: `^`. Thus, 3^5 is coded as `3^5`. However, as with multiplication, we must be careful with matrices: if `A` is a square matrix, then what does `A^2` mean? Is it `A*A` or `A.*A`? Again, by convention, `A^2` is equivalent to `A*A`, and `A.^2` is equivalent to `A.*A`.

Other Built–in Functions

Matlab has many common functions built–in. For example, common trigonometric functions such as

```
>> sin(pi/2)   % pi is a built-in Matlab variable
ans =
      1
>> cos(3*pi/4)
ans =
   -0.7071
>> tan(pi/4)
ans =
     1.0000
```

Note that we have used `pi` to mean π. This is a built–in Matlab expression. Evaluating the expression `pi` will always return the floating–point representation of π, unless you assign that variable to be something else. In addition, we have used parentheses after the function names to call the functions. For example, `sin(pi/2)`. Parentheses are the usual ways to call functions in Matlab. Matlab has many other built–in functions. Although not an exhaustive list, the table below lists some.

e^x	`exp(x)`
$\log_{10} x$	`log10(x)`
$\log_e x$	`log(x)`
$\sin x$	`sin(x)`
$\cos x$	`cos(x)`
$\tan x$	`tan(x)`
$\sqrt{x}$	`sqrt(x)`

Sums, Products, and Initialization

Suppose we have a vector `v` defined as

```
>> v = [1 2 3 4 5 6 7 8 9 10]
v =
     1     2     3     4     5     6     7     8     9    10
```

If we want to take the sum of all the elements of `v`, then the `sum` function can do that for us,

```
>> sum(v)
ans =
    55
```

Similarly, the `prod` function will take the product of all the elements,

```
>> prod(v)
ans =
     3628800
```

Now suppose we wish to create a 3×5 matrix of all zeros. The brute–force approach is as follows:

```
>> A = [0 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0]
A =
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

Of course, entering so many zeros would be too cumbersome were we creating a 34×976 matrix. Matlab comes with a function which makes this process easy: `zeros(m,n)` creates an $m \times n$ matrix of all zeros.

```
>> A = zeros(3,5)
A =
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

The `ones` function does the same thing, but with all entries set to `1`.

Suppose that we want to sum of the integers from 1 to 100. We have the `sum` function which we can use to sum a vector with elements `1 2,...,100`. However, it would be too cumbersome to enter the numbers 1 through 100 individually. The functions `zeros` and `ones` cannot help us directly. However, Matlab has another operator which performs increments. This is the colon operator, and is best explained via examples:

```
>> 1:10
ans =
     1     2     3     4     5     6     7     8     9    10
>> 1:0.1:2
ans =
  Columns 1 through 8
    1.0000    1.1000    1.2000    1.3000    1.4000    1.5000    1.6000
1.7000
  Columns 9 through 11
    1.8000    1.9000    2.0000
```

The first example creates a row vector with elements ranging from `1` to `10`, evidently incremented by `1`. The second example starts at `1`, and goes up to `2` by increments of `0.1`. Thus, `a:b` creates the vector `[a, a+1, ..., b]`. The command `a:c:b` creates the vector `[a, a+c, a+2c,...,b]`, so long as $b - a$ is a multiple of c. In this way, we can find the sum of the numbers 1 through 100 with the command:

```
>> sum(1:100)
ans =
        5050
```

Array indexing

We have discussed some basic operations in Matlab. This section focuses on an intermediate topic: indexing of arrays.

Standard conventions

Let us define a vector `v` in Matlab:

```
>> v = 11:20
v =
    11    12    13    14    15    16    17    18    19    20
```

If we wish to access the entire vector, we simply type `v` at the Matlab command prompt. However, to only access the number 16 in the sixth column, we would type the following.

```
>> v(6)
ans =
    16
```

The expression

```
>> v([3 6])
ans =
    13    16
```

evidently returns the numbers 13 and 16 in the column locations 3 and 6.

Take note that the square brackets in the expression `v([3 6])` are necessary. (See what happens if you omit them.) In other words, the input into the parentheses needs to be a vector. Now if we wanted the first five numbers in `v`, we could type `v([1 2 3 4 5])`, but we can also use the command `1:5` to generate the array `[1 2 3 4 5]` and then feed that into the indexing for `v`:

```
>> v(1:5)
ans =
    11    12    13    14    15
```

In this case the square brackets are unnecessary. `v([1:5])` and `v(1:5)` do the same thing; the command `1:5` automatically creates the vector `[1 2 3 4 5]`. However, `1 2 3 4 5` does not create a vector.

What if the vector `v` has n elements, and we want all elements except the first? Then we could type `v(2:n)`. However, what if we don't know what `n` is? Matlab has a built–in function which will provide `n`. That function is `length` and `length(v)` will return whatever the number `n` happens to be. In addition, Matlab has a special operator, `end`. For a vector of length `n`, the expressions `v(2:n)`, `v(2:length(v))`, and `v(2:end)` are all equivalent, but the second and third expressions do not require us to know `n` in advance.

We have already seen the end *keyword* in the context of terminating `for` and `while` loops. Inside indexing expressions, `end` refers to the maximum value that the index can take. (Those familiar with other more developed programming languages often scoff at Matlab's unrefined practice of using a reserved keyword for data structure indexing.) If we take `v = 11:20` as above, then we have

```
>> v(2:end)
ans =
    12    13    14    15    16    17    18    19    20
```

Now if we wish to access all elements in the vector, then of course we can just type `v`, but can we use the colon operator to access all elements? Sure, simply type `v(1:end)`. In this case, MATLAB provides another shortcut. `v(1:end)` and `v(:)` are *nearly* the same command. There is, unfortunately, a technicality that may cause confusion.

```
>> a = [1 2 3 4 5];
>> a(1:5)
ans =
    1     2     3     4     5
>> a(1:end)
ans =
     1     2     3     4     5

>> a(:)
ans =


     1
     2
     3
     4
     5
```

The difference is subtle: MATLAB leaves the vector alone whenever you index with `1:5` or `1:end`. However, if you index with `:`, MATLAB automatically shapes the vector into a column vector. If you do this experiment with `a = [1;2;3;4;5]`, the column–vector version of `a`, you won't see anything out of the ordinary.

What about matrices? Let us define a matrix `A`:

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

Indexing for matrices follows the same pattern as that for vectors, except that we must provide two indices: one for the row, and one for the column, in that order. So for our matrix `A` above, `A(2,1)` means "return the `A` entry in row 2, column 1," that is the number 4. We can also do indexing for multiple rows and columns as we did for vectors:

```
>> A(1,[1 3])
ans =
     1     3
>> A(:,2)
ans =
     2
     5
>> A(1,:)
ans =
     1     2     3
```

For matrices you can use `:` to access rows without reshaping!

Similar to the `length` expression for vectors, there is a function in MATLAB that returns the $m \times n$ dimensions of a matrix.

```
>> size(A)
ans =
     2     3
```

The function `size` returns two numbers: the first being the number of rows and the second the number of columns.

Note: With matrices, one must be careful with using square brackets. Recall that `[1 2]` *and* `[1,2]` *form the same vector in* Matlab. *With a row vector,* `v([2,1])` *works fine, but* `v(2,1)` *will give an error. However, with a 2×2 matrix,* `A(2,1)` *and* `A([2,1])` *are both defined, but they do very different things!*

Linear array indexing: `sub2ind` and `ind2sub` (optional material)

Define the matrix `A = [1 2; 3 4]` in Matlab. We know how to do subscript indexing of this array: `A(2,1)` returns `3`. However, we can also define a *linear*, or *lexicographic* ordering. This is best explained via an example. Type `A(1)`, `A(3)`, and `A(4)` at the Matlab command line and observe what you get. Now define the matrix `A = [1 2; 3 4; 5 6]`. Type `A(1:6)` at the command line and see what pops out. Evidently, when indexing linearly, Matlab works across rows first, and then shifts to the next column and repeats.

In order to determine a translation between linear indexing and subscript indexing, Matlab has two built–in commands `sub2ind` and `ind2sub`. Again, defining `A = [1 2; 3 4]`, we know that `A(2,1) = 3`. We also now know by experiment that `A(2) = 3`. I.e., for a 2×2 matrix `A`, the subscript index `(2,1)` is the same as the linear index `2`. Matlab can do this translation for us: type `sub2ind([2 2], 2, 1)`. In the order of inputs, this means 'return the linear index of a 2×2 matrix that corresponds to row 2, column 1'. This returns the linear index `2`. The function `ind2sub` does the reverse translation. Linear indexing makes some otherwise–difficult programming tasks easy.