PSYCH 20A (PROF. FRANE) – LECTURE 4
Introduction to Matrices (Continued) and Other Multidimensional Arrays

`x = [1 2 4 ; 0 3 5] ;` first, let's define a 2 × 3 matrix called `x` that looks like this:

1	2	4
0	3	5

Arithmetic operations using matrices

We can add/multiply/etc. a matrix with a number, just like we did with vectors. The computation is simply applied to each element of the matrix. For example, after defining `x` as above, typing `x + 1` outputs:

2	3	5
1	4	6

We can add (or subtract) two vectors or matrices if they are the same size. Each value in one matrix is added to (or subtracted from) the corresponding value in the other matrix. For example, after we define `y` as [2 1 ; 0 2] and `z` as [1 0 ; 1 0] then typing `y + z` outputs:

3	1
1	2

We can do the same thing for multiplication, division, and exponentiation, but we need to put a period before the operator. For example, after defining `y` and `z` as above, typing `y .* z` outputs:

2	0
0	0

If you don't put the period before the `*` operator when multiplying two vectors or matrices, Matlab performs an operation called *matrix multiplication*—which is totally different from multiplying corresponding elements as above. Because division and exponentiation are types of multiplication (in a sense), the same applies to the `/` operator and the `^` operator. For an explanation of matrix multiplication, see:
https://www.youtube.com/watch?v=2spTnAiQg4M

Applying logical statements to matrices

`x > 2` this outputs a matrix of 0s and 1s (falses and trues), indicating whether the given value is greater than 2:

0	0	1
0	1	1

Recall that we can use the `sum` function to find the *number* of values in a vector for which a statement is true, and we can use the `mean` function to find the *proportion* of values in a vector for which a statement is true. We can do the same for a matrix if we vectorize it:

`sum( x(:) > 2)` this outputs 3 (the number of values in x that are greater than 2)
`mean(x(:) > 2)` this outputs 0.5000 (the proportion of values in x that are greater than 2)

Remember, if we *don't* vectorize, then Matlab will apply the `sum` or `mean` function to each column (if we specify dimension 1) or to each row (if we specify dimension 2). For example:

`sum(x > 2, 1)` this outputs 0 1 2 (the number of values greater than 3 in each column)

Using the `find` function with matrices

Recall that the `find` function gives you the indexes of values for which some statement is true.
Thus, for example, if `x` is defined as above:

`find(x > 2)` the 4th, 5th, and 6th values are greater than 3, so this outputs:

4
5
6

Note that the above use of the `find` function gives you the *linear indexes*. We can also use the find function to get the *two-dimensional* (row, column) indexes. The following code produces two vectors (a vector of row indexes called `rowIndex`, and a corresponding vector of column indexes called `columnIndex`):

```
[rowIndex, columnIndex] = find(x > 2) ;
```

Notice we are defining two variables at once in that code by putting the variable names in brackets (the comma isn't necessary, but is conventionally used in this context. Because the row,column indexes that we "find" are 2,2 and 1,3 and 2,3 the `rowIndex` vector will look like this:

2
1
2

And the `colIndex` vector will look like this:

2
3
3

Concatenating

Concatenate means "chain together." For example, when we make the vector [4 8 6], we are concatenating the individual values 4, 8, and 6 into a single object. Similarly, we can concatenate two or more matrices to form one big matrix. For example, if we've defined matrices x, y, and z as before, then typing `[y ; z]` outputs the following 4 × 2 matrix:

2 1
0 2
1 0
1 0

And typing `[x y z]` outputs the following 2 × 7 matrix:

1 2 4 2 1 1 0
0 3 5 0 2 1 0

Another way to concatenate is using the `cat` function. The inputs to the `cat` function are the dimension to concatenate in followed by the objects we want to concatenate. For example, to make the same 2 × 7 matrix as above, we could type `cat(2, x, y, z)` because we are concatenating x, y, and z in the 2nd dimension. That may not seem very useful, since typing `[x y z]` is simpler. But we'll see that the `cat` function is more useful when we move beyond matrices to arrays with three or more dimensions.

Arrays With Three or More Dimensions

We've discussed vectors, which are one-dimensional arrays of values. And we've discussed matrices, which are two-dimensional arrays of values. An array can also have three dimensions (imagine several two-dimensional matrices "stacked" like pages in a book) or even more than three dimensions.

Let's make a 3 × 3 × 4 array by concatenating four 3 × 3 matrices along the 3rd dimension, like pages in a book:

```
page1 = [1 0 3 ; 2 1 9 ; 2 6 5] ;
page2 = [2 7 8 ; 3 0 5 ; 2 1 0] ;
page3 = [3 2 7 ; 1 0 1 ; 8 2 4] ;
page4 = [5 4 1 ; 4 2 0 ; 9 7 3] ;

book = cat(3, page1, page2, page3, page4) ;
```

Three-dimensional indexing works by the same principles as two-dimensional indexing:
To set the value in row 3, column 1, page 2 equal to 5, you would type `book(3, 1, 2) = 5 ;`
To set every value in the last row of page 1 equal to 5, you would type `book(end, :, 1) = 5 ;`
To make page 2 equal to page 4, you would type `book(:, :, 2) = book(:, :, 4) ;`
To set the first row of page 2 equal to [7 8 9], you would type `book(1, :, 2) = [7 8 9] ;`

However, the `find` function does not output three-dimensional indexes. If you want to find three-dimensional indexes, you have to get linear indexes and then convert them.

Vectorization and linear (one-dimensional) indexing:
You can apply vectorization and linear indexing to an array with three or more dimensions, just as to a matrix. For instance, if we've defined the array `book` as before, then `book(3)` is the value in the bottom-left corner of the first page, and `book(10)` is the value in the top-left corner of the second page. `book(:)` is all the values from `book`, "unwrapped" into a single column vector.

Arithmetic operations work the same way as with matrices:
For example, `book * 2` outputs an array the same size as `book`, in which each value is 2 times the corresponding value in `book`. If we have another array the same size as `book` (maybe it's called `otherBook`), then we can add/subtract/multiply/etc. the corresponding values of the two arrays; don't forget to use the period before the multiplication, division, and exponentiation operators, e.g., `book .* otherBook`

Getting the size of an array

To get the "size" (length in each dimension) of an array, use the `size` function. For example, after defining `book` as before, typing `size(book)` outputs `3 3 4` (because `book` has 3 rows, 3 columns, and 4 "pages"). `size([8 1 0])` outputs `1 3` (because the row vector [8 1 0] can be considered as a matrix with 1 row and 3 columns). Note that by default the size function outputs 2 values when the input is a vector or a single value.

To output the length of an array in a particular dimension, just enter the dimension number as the second input to the `size` function. For example, to output the number of columns in `book` you can type `size(book, 2)`

Alternatively, you can say `width(book)` to output the number of columns. `height(book)` outputs the number of rows. `length(book)` outputs the length of `book` in whichever dimension is longest (in this case, it outputs 4).

To get the total number of values in an array, use the `numel` function: `numel(book)`

Short cuts for making arrays with repeating patterns

Sometimes we want to use one value, or one array of values, as a "tile," replicating it multiple times to form a bigger array. We can do that using the `repmat` function, which takes the following inputs: tile to replicate, number of replications in 1st dimension, number of replications in 2nd dimension (and so on, if there are more than two dimensions).

For example, let's say we defined h as the vector [1 7 8]. Then `repmat(h, 2, 3)` outputs a matrix that tiles that vector twice vertically and once horizontally:

```
1 7 8 1 7 8 1 7 8
1 7 8 1 7 8 1 7 8
```

And `repmat(h, 2, 3, 5)` outputs a 5-page array in which each page looks like the above matrix.

The `repelem` function works similarly, but instead of replicating the tile as a block, it replicates each element in the tile individually. For example, if we've defined h as the vector [1 7 8], then `repelem(h, 2, 3)` outputs:

```
1 1 1 7 7 7 8 8 8
1 1 1 7 7 7 8 8 8
```

The `zeros` function makes an array of zeros. For example, to make a 2 × 3 × 5 array of zeros called `q`:

```
q = zeros(2, 3, 5)
```

The `ones` function makes an array of ones. For example, to make a 40-value column vector of ones called `r`:

```
r = ones(40, 1)
```

We can make a matrix of "NaN" values by using the `NaN` function. In Matlab, NaN means "not a number." It's often used to indicate a missing numeric value, or as a placeholder for a numeric value that will be filled in later. To make a 4 × 2 matrix of NaN values called `s`:

```
s = NaN(4, 2)
```

Similarly, the `Inf` function makes a matrix full of "Inf" (infinity) values. But that is rarely useful, e.g.:

```
v = Inf(4, 2)
```

NOTE: When using functions like `repmat`, `zeros`, `ones`, `NaN`, and `Inf` (and sometimes `repelem`, depending on the input) if you only enter a single number of reps, Matlab will assume you want that many reps in both the 1st and 2nd dimension. Thus, for example, if you want a 4 × 4 matrix of zeros, you can just type `zeros(4)` but to avoid confusion I recommend typing `zeros(4, 4)` instead.

Reshaping arrays

`reshape` preserves the one-dimensional indexes of the values in an array while changing its shape.
Let's say `g` is defined as the following 2 × 3 matrix:

```
2     7     3
1     0     8
```

Then `reshape(g, 3, 2)` outputs the values in `g` in the same order, but "reshaped" into a 3 × 2 matrix:

```
2     0
1     3
7     8
```