PSYCH 20A (PROF. FRANE) – LECTURE 3: Introduction to Matrices

What is a matrix?

A matrix is a two-dimensional rectangular array of values. For example:

```
0    1    9    2
8    0    2   12
0    2   11    8
```

The size of the matrix (sometimes called the "order" of the matrix) is defined as rows-by-columns. For example, the above matrix is a 3×4 ("three by four") matrix because it has 3 rows and 4 columns.

We can define a matrix using square brackets, similarly to how we define a vector, but insert semicolons to indicate row-breaks. For example, typing `x = [1 2 4 ; 0 3 5]` defines `x` as the following 2×3 matrix:

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

Or instead of using a semicolon, you can press Return at the end of each row. That's sometimes preferable in a program, because then each row of the matrix is on a separate line of code. So if you line up the columns properly, the values in your code are visually arranged just like the values in the matrix itself.

We call the vertical dimension (i.e., which row you're in) the 1st dimension, and we call the horizontal dimension (i.e., which column you're in) the 2nd dimension.

We can think of a vector as simply a matrix in which one of the dimensions has a length of 1.

Indexing matrices

You can index a matrix two-dimensionally using the following form: row number, column number. For example, after defining x as above, typing `x(1, 3)` outputs 4 because that's the value in row 1, column 3.

You can also use the colon operator in place of a row or column to mean all the rows or all the columns. For example, typing `x(:, end)` outputs the following:

```
4
5
```

And typing `x(1, :)` outputs the following:

```
1    2    4
```

We can also index a matrix using *linear indexing*, meaning we index one-dimensionally, as if the matrix were "unwrapped" into a single vector. Linear indexes are counted as follows: Start with index 1 in the top left corner, then count down the first column, then down the second column, etc. Thus, if we've defined `x` as before, `x(1)` is 1, `x(2)` is 0, `x(3)` is 2, `x(4)` is 3, `x(5)` is 4, and `x(6)` is 5.

Using a vector of indexes outputs a vector of values. For example, typing `x(2:4)` outputs the following:

```
0    2    3
```

And typing `x( [1 3 6] )` outputs the following:

```
1    2    5
```

Vectorization

"Vectorizing" a matrix means converting it to a single vector (specifically, a column vector). The values in the resulting vector are ordered by counting down each column. Vectorization is really a special case of linear indexing using the colon operator. For example, after defining `x` as before, typing `x(:)` outputs the following:

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

Transposing and rotating matrices

The *transpose* of a given matrix has the same numbers, but arranged such that the rows are converted to columns and the columns are converted to rows. In Matlab, we transpose a matrix using a straight single-quote (apostrophe). For example, after defining `x` as before, typing `x'` produces the following 3 × 2 matrix:

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

Note that transposing is not the same as rotating. To rotate a matrix, use the `rot90` function. The first input to the `rot90` function is the matrix you want to rotate. The second input is the number of *counterclockwise 90º rotations* to apply. For example, `rot90(x, 3)` outputs the following:

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

Applying functions to matrices

Some functions, like `sqrt`, `abs`, `round`, `ismember`, and `isequal`, work the same way on matrices that they do on vectors or on single numbers.

We can also use functions like `sum`, `mean`, `sort`, `flip`, `any`, and `all` on matrices. However, because these functions are designed to operate on vectors, we must specify whether we want them to operate on the columns of the matrix or on the rows. In other words, we must specify which *dimension* of the matrix we want the function to operate on. We do that by putting the dimension number as an input to the function. For example, after defining `x` as above, typing `sum(x, 1)` sums values across the 1st dimension (that is, across each column), producing the following row-vector 1 5 9. And `sum(x, 2)` sums across the 2nd dimension (that is, across each row), producing the column-vector [7 ; 8].

The `min` and `max` functions are also designed to work on vectors, so we must specify what dimension we want them to operate on. However, unlike with the `sum`, `mean`, `sort`, `flip`, `any`, and `all` functions, we don't input the dimension as the 2nd input. Instead, we enter it as the <u>3rd input</u>. To do that, we enter an empty vector `[]` as the second input. For example, `min(x, [], 1)` outputs the smallest value in each column of x, and `max(x, [], 2)` outputs the largest value in each row of x.

For most functions that are designed to work on vectors, if we don't specify the dimension, the default is the 1st dimension, <u>unless we are applying the function to a vector</u>, in which case Matlab assumes we want the mean across whichever dimension has a length greater than 1.

If we want to take the min/max/sum/mean across an entire matrix and get a single number, we can vectorize the matrix first. For example, typing `sum(x(:))` outputs 15.