ENGR101 Lab 2

Submission Requirements:

All labs are due at 11:59pm the day after your lab section. See Part 3 of this lab on how to submit these files.

- Exercise 1: Save your work as lab2_1.m
- Exercise 2: Save your work as lab2_2.m
- Exercise 3: Save your work as lab2_3.m

Part 1: Vectors and Matrices

Introduction

Matlab stores data in arrays that are groups of one or more items. In MATLAB, we will spend most of our time working with numbers, but MATLAB can also store other types of data into arrays. We will learn about these other types during the semester. Nonetheless, learning how to use and manipulate these arrays is essential to working with MATLAB.

As learned in lecture, we can store numbers into variables. Each variable will have a **size**. The size is the number of rows and columns. Variables can have the following formats:

- A **matrix** is stored as a ROWxCOLUMN two-dimensional array in MATLAB.
- A **vector** is stored as a 1xCOLUMN or ROWx1 one-dimensional array in MATLAB.
- A **scalar**, or single number, is stored as a 1x1 array in MATLAB.

1	2	8
19	4	2
9	6	7

Matrix

2
5
8

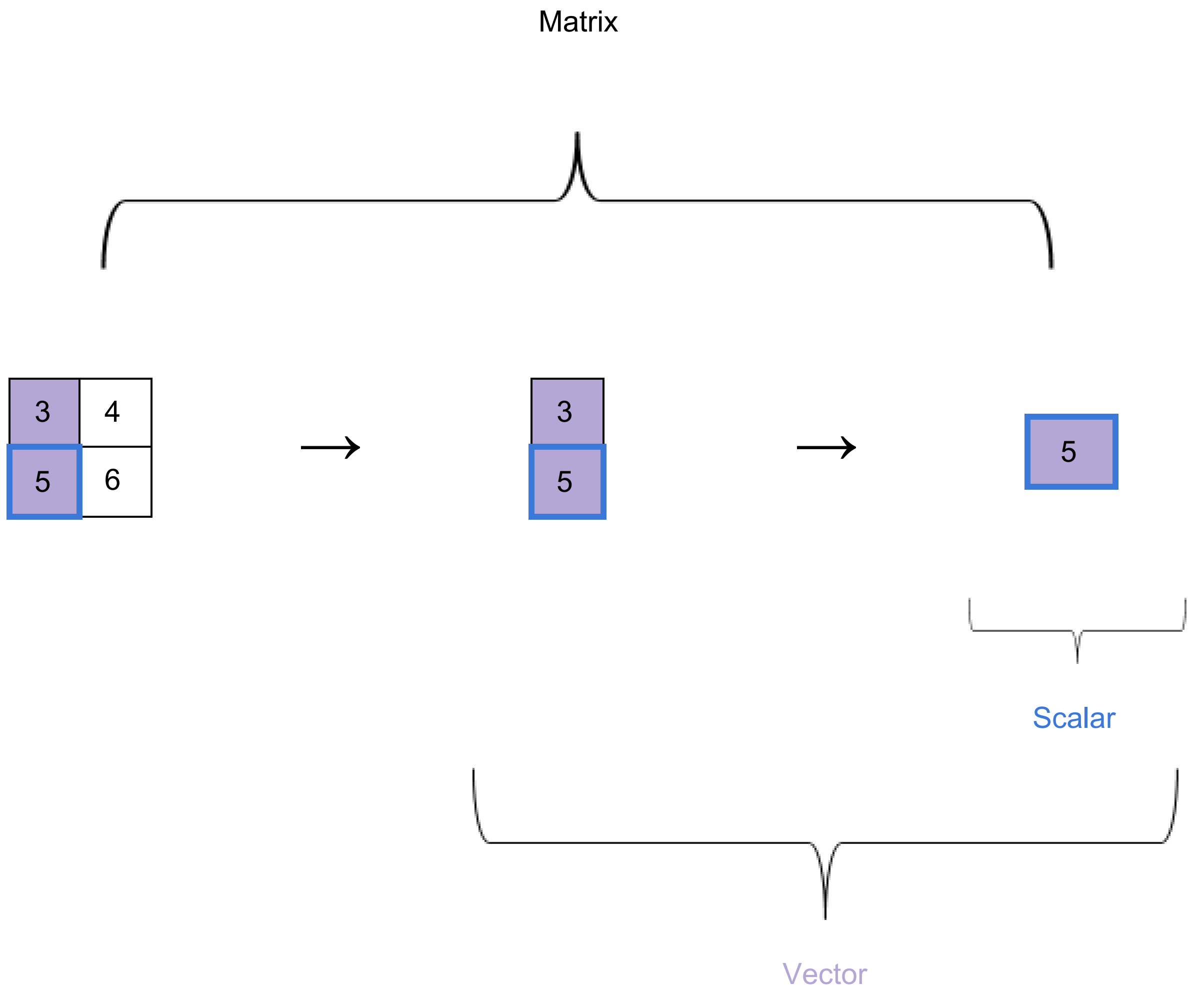
Column Vector

4	5	6	7	8	9

Row Vector

5

Scalar



This means a scalar is a vector and a matrix, and a vector is also a matrix. As a result, all arrays support the same arithmetic operations. These operations happen on an element-by-element basis. Here are some common array operators:

Note: The equal signs in the table are NOT assignment operators

Operation	Array Operator	Description/Example
Addition	+	Adds arrays if they have the same size or adds a scalar to each element Ex: [1 2]+5 = [6 7] [1 2]+[2 3] = [3 5]
Subtraction	-	Subtracts arrays if they have the same size or subtracts a scalar from each element Ex: [1 2]-5 = [-4 -3] [2 4]-[2 3] = [0 1]
Multiplication	`.*`	Multiplies arrays element by element if they have the same size or multiplies a scalar to each element Ex: [1 2].*5 = [5 10] [2 4].*[2 3] = [4 12]
Division	`./`	Divides arrays element by element if they have the same size or divides a scalar to each element Ex: [10 5]./5 = [2 1] [6 8]./[3 4] = [2 2]
Exponentiation	`.^`	Takes the elements in the left array element by element to the power of the right array or takes all the elements in the left array to the power of a scalar Ex: [2 5].^[3 2] = [8 25] [2 3].^3 = [8 27]
Transpose	`’`	Turns row into columns and vice versa Ex: [1 2]’ = [1; 2] [1 2 3; 4 5 6]’ =[1 4; 2 5; 3 6]

Indexing Elements

Once we have an array, how do we manipulate only certain parts of it? Indexing is the way MATLAB allows access to elements. Each element in an array is given an index going top to bottom and then left to right. Here is an example of each index in a 3x3 matrix:

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

A more useful way to index a two-dimensional array is by row/column indexing. You can think of each element as having a coordinate based off of its row and column. Here is an example of this kind of indexing for a 3x3 matrix:

1,1	**1,2**	**1,3**
2,1	**2,2**	**2,3**
3,1	**3,2**	**3,3**

While indexing gives a way for us to access elements in an array, sometimes it can be tedious. MATLAB provides the keyword *end* and the colon operator to help with indexing. The *end* keyword designates the last element in a particular dimension of an array (example shown below). This keyword can be useful in instances where your program does not know how many rows or columns there are in a matrix. The colon operator is used in place of an index to designate a range. To use an index on an array, give the name of the array followed by the index in parentheses. Here are examples of indexing a 2x3 matrix.

```
>> tuna = [2 4 6; 1 3  5]     % create matrix tuna
>> tuna(1)              % indexes the first element which is 2
>> tuna(2, 2)           % indexes the element in row 2, column 2
                        % (In this case 3)
>> tuna (1, 2:3)        % indexes the elements in row 1, in column 2 and 3
                        % (In this case [4 6])
>> tuna(1, 2:end)       % indexes the element in row 1, column 2 to the end
                        % (Same as above [4 6])
>> tuna(1, :)     % gets all elements in the first row which is [2 4 6]
>> tuna(:, 1)     % gets all elements in the first column which is [2; 1]
```

Part 2: Functions

Functions are an extremely useful tool in programming. Functions allow the programmer to write an algorithm once and use it many times. Just like functions in math, they both take a number as input and output a number. An example of a function in both math and MATLAB is the absolute value function (abs). This function takes a number as input and outputs the absolute value.

Math: $y = |x|$

MATLAB: y = abs(x)

We will discuss functions in depth in the next lab. They are not exactly like functions in math. In the meantime, here are a few useful functions that are pre-built into MATLAB:

Note: The equal signs in the table are NOT assignment operators

`size()`	Outputs the number of elements in each dimension of a matrix. Ex: size([1 2; 3 4]) = [2 2] size([1; 2]) = [2 1] size([4]) = [1 1]
`length()`	Outputs the largest dimension of a matrix. This will always be a scalar. Ex: length([1 2; 3 4]) = 2 length([1; 2]) = 2 length([1 2 3]) = 3
`zeros()`	Creates a matrix of zeros. Ex: zeros(2) = [0 0; 0 0] zeros(2,3) = [0 0 0; 0 0 0]
`ones()`	Creates a matrix of ones. Ex: ones(2) = [1 1; 1 1] ones(2,3) = [1 1 1; 1 1 1]
`sum()`	Outputs the sum of a vector. If used with a 2D matrix, it will output the sum of each column. Ex: sum(1:10) = 55 sum([2 3; 4 5]) = [6 8]
`mean()`	Outputs the average of a vector. If used with a 2D matrix, it will output the average of each column. Ex: mean(1:10) = 5.5 mean([5 2 9 4]) = 5
`median()`	Outputs the median value of a vector. If used with a 2D matrix, it will output the median of each column. Ex: median(1:10) = 5.5 median([5 2 9 4]) = 4.5
`max()`	Outputs the maximum element of a vector. If used with a 2D matrix, it will output the max of each column. Ex: max(1:10) = 10 max([2 3; 4 5]) = [4 5]
`min()`	Outputs the minimum element of a vector. If used with a 2D matrix, it will output the min of each column. Ex: min(1:10) = 1 min([2 3; 4 5]) = [2 3]
`magic()`	Outputs an n-by-n matrix constructed from the integers 1 through n^2 with equal row and column sums. N must be greater than or equal to 3 Ex: magic(3) = [8 1 6; 3 5 7; 4 9 2]

Vector Examples

```
>> m = [2 3 4 5 6 7 8 9]
>> m(1)                % gives back the 1st element of vector m, 2
>> m(0)                % will give an error
>> max(m)              % gives back 9, the largest element
>> m(length(m))        % will give the last element of the vector m (9)
>> m(end)              % will also give the last element of the vector m (9)
>> m(:)                % gives back the entire vector as a column
>> m(:,:)              % gives back the entire vector as a row
>> m(3:5)              % gives back the 3rd through 5th elements in a vector
>> m(1:2:end)          % gives back the odd-indexed elements in a vector
>> m(1) = 7            % reassigns the 1st element to a value of 7
>> m(3:5) = [-1 -2 -3]  % reassigns the 3rd, 4th, and 5th elements
>> m(5) = m(1)         % reassigns the 5th element with the 1st element
>> m(5) = []           % deletes the 5th element of the vector
```

Matrix Examples

```
>> z = [1 2 3 4 5; 6 7 8 9 10]      % matrix, 2x5
>> size(z)             % gives back [2, 5]
>> sum(z)              % gives back [7 9 11 13 15]
>> mean(z)             % gives back [3.5 4.5 5.5 6.5 7.5]
>> median(z)           % gives back [3.5 4.5 5.5 6.5 7.5]
>> z(1,1)              % accesses the 1st row, 1st column
>> z(2,3)              % accesses the 2nd row, 3rd column
>> z(:,3)              % accesses the full 3rd column
>> z(2,:)              % accesses the full 2nd row
>> z(1:2,3:4)          % access the 3rd and 4th elements (columns) of
                       % the 1st and 2nd rows
>> z([1 2],[3 4])      % another way to write the step above
>> z(2,3) = -3         % reassigns the element at the 2nd row, 3rd column
>> z(1:2,3:4) = [-7 -8; -9 -10]     % reassigns the block
>> z(1,:) = []                      % deletes the first row
                       % (note: you can only delete entire rows or columns
                       % in matrices without getting an error.)
```

Part 3: Turning in Your Assignments

Labs must be turned in through the relevant assignment in Canvas. Be sure to submit all files listed under Submission Requirements.

The names of your files must be exactly as specified at the top of this lab in order to receive credit.

Suppress Output

In all exercises, suppress all output using the semicolon! This a good pattern to get used to, since the autograder requires all output to be suppressed. Additionally, it's easier to find errors in your code when you have fine-grained control over what is displayed.

Double-check your work by comparing the variables in your workspace to the answers.

Exercise 1: Working with Vectors

1. Download the `lab2_1.m` file from Google Drive where you found this lab.
2. Create a `lab2` folder in your `engr101/labs` folder and place `lab2_1.m` into this folder.
3. Navigate your Current Folder to your `engr101/labs/lab2` folder. You should see `lab2_1.m` in the list of files.
4. Complete the program by reading the comments and filling in the missing lines.
5. Run your script by typing the following into the Command Window.
   ```
   >> lab2_1
   ```
6. Attach the completed file to your Canvas submission.

Make the following variables inside `lab2_1`:

A =

1
2
3
4
5

B =

1	2	3	4	5

After completing `lab2_1`, your variables in your workspace should look like the following:

A =

1
2
4
2
1

B =

3
2
1
2
3

C =

3
4
4
4
3

Exercise 2: Working with Matrices

1. Download the `lab2_2.m` file from Google Drive where you found this lab.
2. Place `lab2_2.m` in your `engr101/labs/lab2` folder created in Exercise 1.
3. Complete the program by reading the comments and filling in the missing lines.
4. Attach the completed file to your Canvas submission.

After completing `lab2_2`, your variables in your workspace should look like the following:

D =

6	-1	4
1	3	5
2	7	0
6	-1	4

E =

4	3

F =

8	4
8	3
8	4

G =

35

H =

3.75	2	3.25

I =

4	1	4

Exercise 3: Smiley Faces

Background

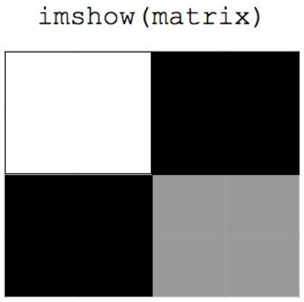
A grayscale image can be represented as a matrix. Every element in the matrix represents a pixel. For example, the element at row 1 and column 1 of our matrix is the top left pixel in an image. The values of our matrix must be between 0 and 1 (0 is black, 1 is white).

Use the `imshow(matrix)` function in MATLAB to display the matrix as an image.

`matrix`

1	1	0	0
1	1	0	0
0	0	0.5	0.5
0	0	0.5	0.5

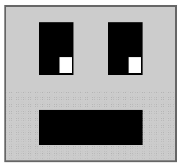
$\rightarrow$



Your Task

Create a smiley face with a matrix. The minimum requirements are:

1. A size of at least 500x500
2. Two eyes
3. A mouth



Create a `lab2_3.m` file in your `engr101/labs/lab2` folder. Attach the completed file to your Canvas submission.

Example

Use this example code to get started.
Hint: copy and paste this into your `lab2_3.m` script

```
% Create a 800x1000 white matrix
img = ones(800, 1000);

% Create a black bar in the middle of the image
img(400:600, :) = 0;

% Display the matrix as an image
imshow(img);
```