Name: Eng Sci 53
Date: Lab 4
TF:

Matlab Tutorial 2

This tutorial will be an overview of resolution, random numbers, matrix manipulation, Boolean operators, functions, how to import data into Matlab, some basic data analysis, and how to export figures. **NOTE: All bold sections are to be answered.**

Part 0 – Nice Shortcuts

In the Command Window, if you press the up-arrow-key, it will pull up the last command entered there.

In the Command Window, if you start typing a variable or function name, before you complete it, you can press 'tab' and it will give you a list of potential names that you can choose from, or if there's only one option, it will automatically fill that in for you. For example, if you type in 'tutor' and press tab, it'll fill in 'tutorial_1.'

Ctrl-R comments the whole line.
Ctrl-T uncomments the whole line.
Ctrl-I adjusts the indentation to the proper position.

As always you can use '`help`' to learn more about the following commands.
'`figure`' creates a figure window
'`close`' closes a figure window
'`clear`' clears variables and functions from the workspace
'`shg`' brings the current figure window forward.
'`why`' gives answers to life's questions.

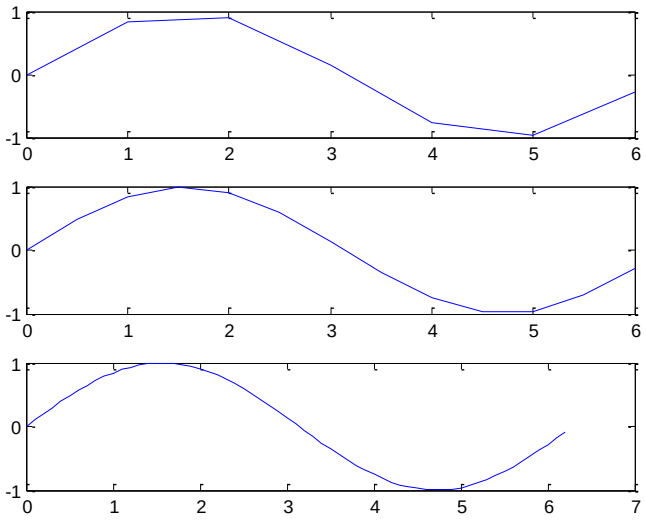
F5 – Runs the script or program you have open in the M-file editor.
F9 – Runs the script that is currently selected inside the M-file editor.

Part 1 – Resolution

One concept that will come up every now and then is resolution. We all have an intuitive feel for this word because of how often it comes up in technology. If a TV has good resolution, then it means that it can display edges and small objects very clearly. The general concept of resolution is very similar – it's the degree of sharpness of an image or a measurement based on the closeness of the pixels or data points involved. To investigate this concept, as well as to practice some more graphical manipulation, we'll look at sine waves using scripts.

Create a script (sinewaves.m) that defines three different sine waves, all of them with the same amplitude (1), frequency (1 Hz), offset (0), and range (0 to 2*pi), but with different

resolutions (increment values of 1, 0.5, and 0.1). Plot all three sine waves in the same figure but in different subplots displayed 3x1 (`help subplot`)**.**



Once you've done this, you should hopefully be able to see that the smaller the increment the cleaner/smoother the image. Click on the zoom in tool and explore the "granularity." **What are some reasons why you *wouldn't* want a really high resolution?**

Part 2 – Random numbers

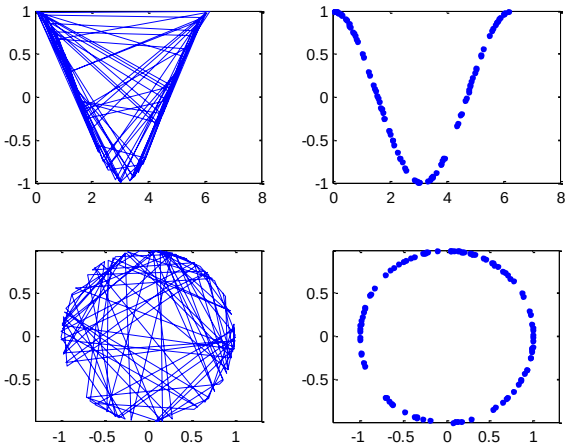
Matlab can generate random numbers very easily. If you would like to generate a random number between 0 and 1 from a uniform distribution, you would type in 'rand(1)'. This will generate a 1x1 matrix of random numbers.

All **bold** sections are to be answered.

How would you generate a 4x4 matrix of random numbers ranging from -10 to 10, uniformly distributed? (`help rand`).

```
>> _____________________________ (?)
```

Attach a section to the previous script (tutorial_1###.m) to make a figure with subplots arranged 2x2. The top row should be plots of a cosine wave (one using a line, and one using dots), but plotted with 100 randomly generated points. The bottom row should be plots of a circle (one using a line, one using dots), but also plotted with 100 randomly generated points (`help subplot`). Why are the graphs using lines so messy? Imagine we used a function to sort our 100 points in increasing or decreasing order (for example, using `sort`). How would the graph look like? You don't have to write anything down for this question.



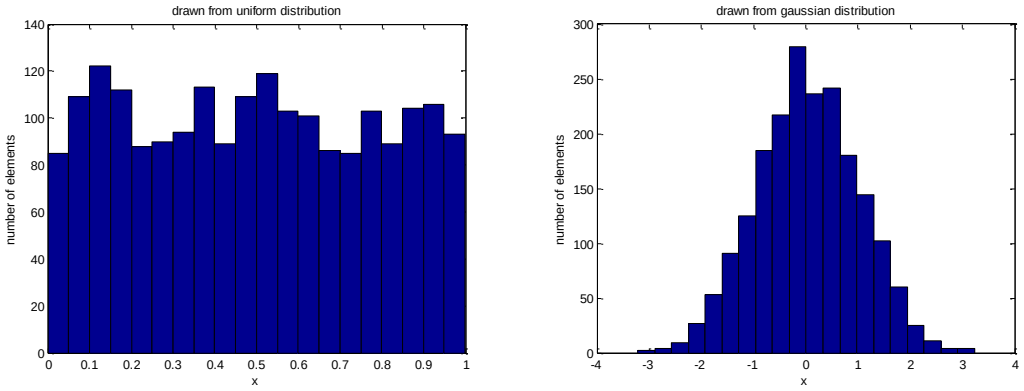
So, `rand` draws numbers which follow a uniform distribution between 0 and 1. Another useful function is `randn`, which draws numbers following a Gaussian distribution (also known as normal distribution "bell curve").

How would you generate a size 12 vector of random numbers, normally distributed with a mean of 0.5 and standard deviation of 0.2?

>> _____ **(?)**

Imagine now that, we took some measurements and we want to have a picture on how our data is distributed. A function which is really helpful in that case is `hist`.

Attach a new section on the script as follows: Create two random vectors of observations: one which follows a uniform distribution, and another one that follows a Gaussian one. The two vectors should be 2000 elements long. Plot their histograms next to each other using `hist`, with 25 bins. Label accordingly. Note that what you get is not a plot of the distribution function itself – the y axis just represents the number of elements that have values within each bin – but it should have a similar shape.



Part 3 – Boolean Operators

What are Boolean operators? Perhaps you've encountered them when you've searched for things online. AND, OR, NOT are all part of Boolean logic. In Matlab, these operators are represented by &, |, and ~, respectively. As well, TRUE and FALSE have a role – 1 represents TRUE and 0 represents FALSE. The question is, where do you use these?

Imagine that we generate a vector of 8 random numbers, uniformly distributed from 0 to 1. Now, let's say that we want to find which of those numbers are below 0.5. We can type the following:

```
>> r = rand(1,8)

r =

  0.2683  0.1030  0.1905  0.1467  0.3779  0.1149  0.7904  0.8925

>> r < 0.5

ans =

     1     1     1     1     1     1     0     0
```

Here, we see that the last two numbers of our example vector were GREATER than 0.5. Remember, 1 corresponds with TRUE, 0 corresponds with FALSE. **Now, how would we find the vector elements that are both less than 0.5 AND greater than 0.3?**

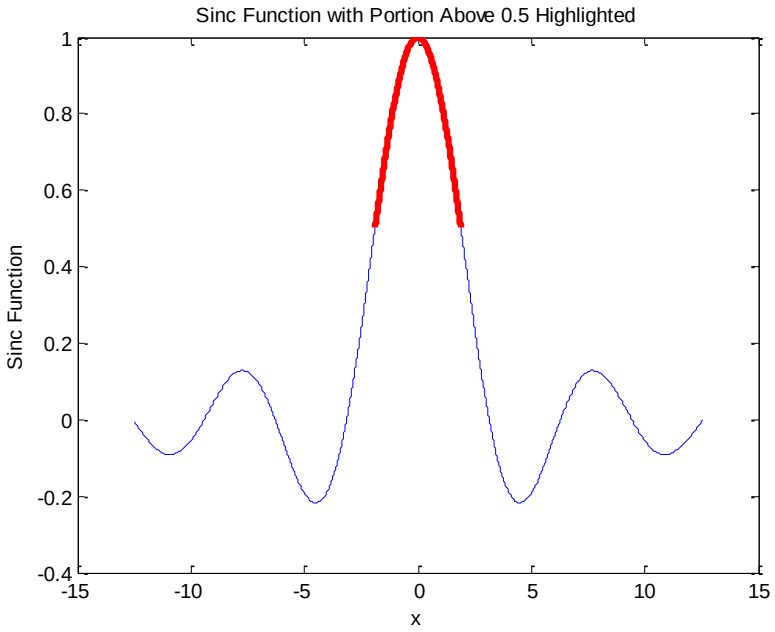
```
>> _____________ (?)
```

If we want to find the actual vector indices of these elements, we can use the function 'find,' which pulls out all of the indices that are TRUE, or 1 (help find). We can then put these indices back into the vector to find the actual values. **How would we do this?**

```
>> r(_________________) (?)
```

Another use for find is to identify certain parts of a curve. **For example, let's say that we want to plot a blue sinc function (sin(x)/x)) in the range -4*pi to 4*pi, and highlight the portion of the curve above 0.5 with a red line of line width 3. How would we do this, with everything properly labeled? (You can also do this in a script, if you'd like. Maybe call it sinc_script.m?)**

```
>> x = _________ (?)
>> y = _________ (?)
>> figure; plot(________) (?)
>> i = find(__________) (?)
>> ________ (?)
>> plot(__________)
>> Labels, etc.
```



Next, we want to find the distance between successive peaks and troughs of the sinc curve. Also, where is the slope maximal of y(x)? What is the vector index at this point? What are the x & y values for this point? (hint: at some point, you will use the '`diff`' function)

Distances between peaks: _____ (?)
Distances between troughs: _____ (?)
Maximal slope: _____ (?)
Vector index: _____ (?)
X and Y values for this point: _____ (?)

Part 4 – If-else statements

Sometimes we want to give commands to matlab or make some calculations only if several conditions are satisfied. The way to implement this is by using if-else statements. An if statement checks if a particular condition holds true, and if so, executes some code enclosed by the statement.

```
if (n > 1), n = n-1; end
```

Lots of n's! A verbal interpretation of this single line statement is "if n is greater than one, then subtract one from it". The condition and the code to be potentially executed are separated by a comma (they can also be separated by a line change), and the word `end` is used to signify the end of the things we want matlab to do conditional to whether n > 1 holds true or not. The general structure is

if (condition is true)**,**{list of commands to do in this case}**end**

The condition above is described by a Boolean operator, taking values TRUE(1) of FALSE (0).

```
if (1), n = n-1; end
```

What would the previous statement do? _____ (?)

Suppose now we have a list of things to do on different occasions – for example, if it's nice and sunny, go for a walk; else, if it's cloudy, go for a coffee; else, otherwise, stay home and do your ES 53 assignment. We can express that kind of statements using if – else:

if (condition 1 is true)
{list of commands to do in case 1}
else if (condition 2 is true)
{list of commands to do in case 2}
…
else
{list of commands to do if none of the other conditions holds true – default choice}
end

Some important things to note:

- Conditions are checked one after the other – if the first is true, then the first list of commands will be executed and then the program will jump to the point after the **end** statement. That is, the computer will bother to check the second condition only if the first one is false, and so on.
- We can have any number of conditions to test – just add more "else if".
- It is not necessary to have a final "default" statement – in this case, if no one of the conditions holds true, nothing will happen.
- You can "nest" if-else statements – for example, the list of commands to do in case 1 might include if-else statements of its own, and so on. That's not much likely to be necessary for the purpose of this lab!

Let's see this simple example:

```
n = rand-0.5;

if (n>0)
    n_abs = n;
elseif (n==0)
    n_abs = 0;
else
    n_abs = -n;
end
```

This script creates the variable n_abs, which is the absolute value of n (of course, Matlab itself has a function that does it for you – `help abs`).

Create a small script that rounds a number to its nearest integer.

Note: when you assign variable names, you cannot use any Matlab control words like `else`, `if`, `end`, `function`, `for` etc. Those are the words that turn `blue` in your script. Another set of words that you can use, but you should avoid, are words used to call Matlab functions. For example, if you name a variable called "sin", the `sin` function won't work until you delete that variable.

Another statement that is in a way similar to if is `while`. Where as an `if-end` statement will have the list of commands it encloses executed at most once, a `while` loop will do it over and over again as long as the condition is satisfied:

```
time = 1;
while (time < 100)
    out(time) = 2*time;
    time = time + 1;
end
```

To make sure you understand how while works, think: what will the value of "time" be after this script runs? What will the final element in "out" be? You can then check by running the code.

Define a vector from 0 to 20 in 0.5 increments. Write code that outputs another vector containing the squares of these numbers, only if those squares are smaller than 100. Use `while`.

Part 5 – For loops

In the first part of our tutorial a few weeks ago we saw an important element of Matlab (and programming languages in general), the `for` loop. It can provide us with extra capabilities, especially when we are handling large data sets. Let's begin with a simple example. Assume we need to create the maximum value of a vector *x*. You can create a random vector of any size and look at the short script next page (`help length`):

```
maxValue = x(1);
for i = 2:length(x)
    if (x(i) > maxValue)
        maxValue = x(i);
    end
end
```

Add two lines to the above code to get, the index of the maximum value within the vector (i.e. do something similar to `find`).

We will further explore for loops in the next part.

Follow instructions and answer all questions for the rest of this lab.

Part 6 – Functions

In the first tutorial, we learned how to write scripts, which are basically compilations of line-by-line commands for the Command Window written in the separate M-file editor. Functions are also written in the M-file editor, but are a bit different – running scripts gives you full access to all of the variables in the workspace, but running functions restricts your access to the variables being used by the script used to call the function. The function takes in certain variables from the calling script and outputs certain variables to the calling script. It also leaves the input variables unaffected, (unless you explicitly want to overwrite them by using the same variable names as

input and output). Another important idea is that functions can call other functions. This might be a bit confusing, so why don't we try some examples first, and then hopefully this will become clearer as we move on.

Let's make a function that takes in a vector of numbers and shoots out its standard deviation. First, we need to create the file.

Create a new M-file and save it as 'standard_dev.m' IN YOUR OWN FOLDER UNDER ES53!!

At the top of the file, type in

```
% YOUR NAME
% ES 53, Lab 2, Standard Deviation Function
% standard_dev.m
```

In Matlab, the percentage sign is an indicator to the compiler that you want the following words on that line to be commented out. You can also put the percentage sign in the middle of the line – for example,

```
X = [4 0 9 8]; % Even after defining X, I can comment.
```

Because we saved the M-file as 'standard_dev.m', that is now going to be our function name. We want this function to take a vector x and output its standard deviation. As you might recall, the standard deviation σ of n samples x(i), i = 1…n, is given by

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2} \Rightarrow \sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where $\bar{x}$ is the mean value of all x's.

We first need to tell Matlab the number of outputs, inputs, and the name of our function. This is all done in the function header. (go to the help file on '`function`')

```
function sigma = standard_dev (x)
```

Matlab now knows that this M-file will be a function because the first non-commented word it sees is '`function`.' As well, it knows that this function will be outputting one thing, and it will take in one thing, 'x'.

Alright, let's now create the body of our function. How would you do this? Complete the following template. Useful Matlab functions to include: `length, mean` (or anything except `std`).

```
function sigma = standard_dev (x)

%find the mean
_________________?

%calculate the sum

n = length(x);

for i = 1:n
    _________________?
end

%divide the sum by (number of samples - 1) and take the square root

sigma = _________________?
```

Give your function some inputs and cross-check using the built – in function `std`.

Now let's create a function that outputs a bunch of useful information about a vector x. Save the file as stats.m, in the same directory where you saved standard_dev.m .

```
% YOUR NAME
% ES 53, Lab 2, Stats Function
% stat.m

function [minimum,maximum,average,standard_deviation] = stat(x)

minimum = _________________?
maximum = _________________?
average = _________________?
standard_deviation = _________________?
```

You can use the function(s) you defined yourself. Notice, while we were able to simply have all calculations done explicitly within the fuction stats itself, its far more organized (and easier to read when you are checking your code) to include calls to specific functions. In our code, stat would call `standard_dev`. In order to perform its job, `standard_dev` will call, for example, `length` etc. Imagine that huge bunch of stuff thrown in the same .m file code! Now, let's write a calling script.

Create a new M-file and name it 'tutorial_2.m'. At the top of the script, write the appropriate information (name, class, lab number, script name).

Define a vector of any size you want, with any sort of real numbers in it.
(e.g. `x = [3 9 5 2 0 -3 -5 9 1];`)

Put that into the function 'stat.m' and set the output of 'stat.m' to some variables.

'`[a b c d] = stat(x);`'

Note that it is VERY important to have the same number of elements on the left side of the equal sign as the number of outputs of the function. Otherwise, data that you want to store won't be saved.

With that done, run tutorial_2.m and then in the command window, find out what the values of a, b, c, and d are!

Part 7 – Structures

So far, we've been dealing with simple variables that you can only define as one vector or matrix. Matlab has other data structures that are more powerful, such as structures and cells. We'll not be working with cells in this class, but structures will definitely be useful to us once we start analyzing lab data. The general idea of a structure (help for 'building structure arrays') is that there's an overall variable name, and then within that variable are individual field names. For example, if I work as a statistician and I want to store all of my information about a particular data set, I could use structures to make that easier.

We currently have a, b, c, and d equal to the min, the max, the mean, and the std of our vector 'x'. Let's store those in a structure.

```
set_values.min = a;
set_values.max = b;
set_values.avg = c;
set_values.std = d;
```

This way, we're able to store all of our data into a single variable and if we want to recall all that information, all we have to do is type in 'set_values' into the Command Window and we get:

```
set_values =
    min: -4
    max: 9
    avg: 2.4545
    std: 4.0091
```

Pretty nice, eh?

Part 8 – Importing data

Let's create a calling script and a function that will take in a data curve, plot it, and then perform some analysis on it, storing the information into structures.

Save a script, reader.m, and a function, analysis.m.

The reader script will take in the data file and save it into vectors, plot it, and then feed it into the analysis function. First, let's get our data.

Go to http://people.seas.harvard.edu/~gcsing/ES53/Lab and download the dat file there. Save it in your ES53 directory.

Next, in the reader.m script, load in the data file.

```
'data = load('tutorial_2_data_set');'
```

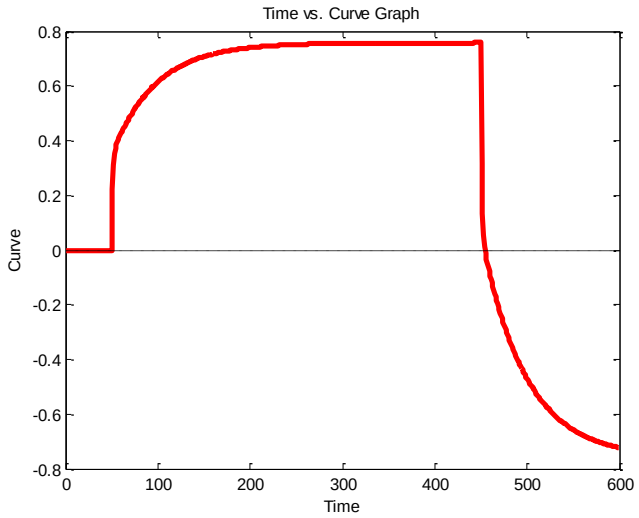
The data set is a 600x5 array, where the first column contains time and the fifth column contains the data points for a curve (we'll get to the other columns later). We want to pull out these two vectors from the others, so type in the following:

```
'time = data(:,1);
net_response = data(:,5);'
```

The colon symbol in the array signifies that we want to access all of the data points in that dimension of the array. You can think about it as data(all,1) in terms of words.

Now, let's plot this curve to see what it looks like. The plot should have appropriate labels, and the net_response should be a red solid line with line width of 3. As well, we want a thin, dashed black line signifying y = 0.

If we want to paste the figure into a Word document, we can do it in two ways. The first way is to go to the File menu of the figure and click on copy figure. The other way is to save the figure and then paste it in. There are advantages to both ways. An advantage to the first way is that it's quicker and you don't have to take up space in saving the figure. The disadvantage is that you copy the figure as a JPEG, which means once you paste it into the document, any adjusting the size will distort the image. If you save it as an .emf file, or an enhanced metafile, and then paste it in, resizing the image won't distort it because the actual data represented is still embedded in the figure, and so resizing will just cause the image to automatically rescale the data. Try it out for yourself.



Part 9 – Analyzing Data

This part presents a situation similar (and correspondingly demanding) as the ones you will come along (or came along already) in the actual lab assignments.

Introduction

The curve shown above is the result of inputting a particular stimulus into a model representing how humans learn how to make arm reaching movements in a force field. In this model, we consider the magnitude of the force field as an input, and we take the response (red curve) as the level of adaptation – learning of the human motor system to it. In this curve, full adaptation would mean that the response reaches the input force field magnitude.

More specifically, the model states that in learning how to make these movements, the human motor system actually has two processes that are learning in parallel – a fast system and a slow system. Their *sum* is what gives the net response represented by the red curve.

What are we asking our model to learn? In this case, we're asking our model to learn how to respond to a particular stimulus, which is contained in Column 2 of our data file (a bunch of zeros, a bunch of +1's, and then a bunch of -1's). **What can analysis of the model's response tell us about the behavior in the human motor system to this experimental paradigm?** Columns 3 and 4 contain the fast and slow systems mentioned above, and again, Column 1 is the time and Column 5 is the sum of the fast and slow systems.

1. Plot the disturbance (black solid line), the net response (red solid line), the fast response (green dashed line), and the slow response (blue dashed line) on the same graph with appropriate scaling and labels/legend.
2. Input time, stimulus, and net response into our analysis function (analysis.m) and have the function do the following:
    a. Output the length of time that the stimulus is +1. ('`find`' will be helpful here)
    b. Output a vector containing the values of the net response corresponding to all the times when the stimulus is +1. (Save this vector as 'learning_curve').
    c. Output the zero-crossing point of the net response. (If this zero-crossing point happens between two time points, you can just take the average of those two time points. If you want to be really accurate, you'll have to interpolate to find out exactly where the zero-crossing lies in between the two time points.)
    d. Output two vectors containing the times and values of the net response that include and are after the net response's zero-crossing point. (This is basically the zero_crossing point and the part of the curve that is below zero – save these as 'opposite_learning_time' and 'opposite_learning_curve').
3. Plot the learning_curve (solid red line) and opposite_learning_curve (dashed red line) on the same graph (using `hold on`), but with both curves starting at zero, and with the opposite_learning_curve rectified (made positive). Again, proper limits and labels are required.

Results

Give the results / graphs for each of the assigned questions (DON'T give us vectors and vectors of data! If it's a lot of numbers, just plot it. If it's a single number, then give that value.) Talk about how you found these answers. (Whenever you use Matlab, make sure to include the code at the end!)

Discussion

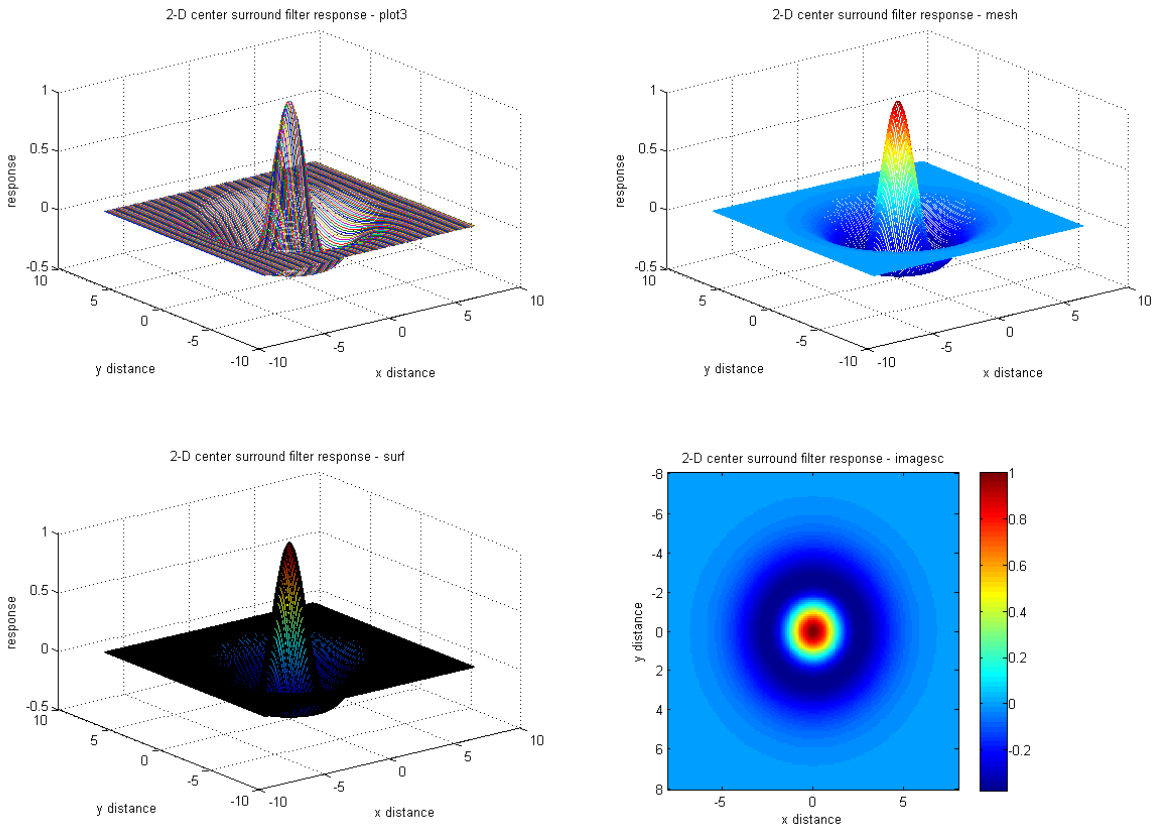
Recall that this model describes the human motor system's response to a zero, +1, -1 experimental paradigm. Discuss importance of different processes at different points of the stimulus. (e.g. beginning of learning +1, end of learning +1, beginning of learning - 1, end of learning -1) What does the graph of the learning curve and the opposite learning curve show? Explain.

Part 10 – 3D representation (Bonus)

Imagine you have a set of 3D data – it could be the location of some points in space, or the average value of real estate with respect to some different locations on the map (where you would have two dimensions – x, y for the location, and a third one for the real estate value at each (x,y) point).

Download and run the file '3Ddata.m' from the course website. The first part creates a model for a center-surround filter. FYI (you don't have to deeply understand how it works, even if that is always a good thing), the function `meshgrid` creates a lattice in the x-y space where our function's values can be calculated on. **Remember – use `help` for any function you want to know about.**

- Plot center_surround_filter using `plot3`, `mesh`, `surf`. Which one you think gives you the best representation of it? Label accordingly (remember, now you have a z dimension, and you also have the functions `zlabel`, `zlim` etc.). Another function which is useful for representation of 3D data is `imagesc`. Here, the third dimension is represented by color: low values can be, for example, represented by 'cool' colors and high values by 'hot' ones (or vice versa) – you can select the coloring scheme by writing `colormap`(yyy), where yyy can be one of the specified matlab colormaps (jet, hot, winter, gray etc.). Typing `colorbar` will add a representation of this colormap next to your current plot. Use `imagesc` to plot the data selecting a colormap of your choice. Display the colorbar on the right side of the plot and label accordingly. Note how you can rotate the first three plots – cool, eh?

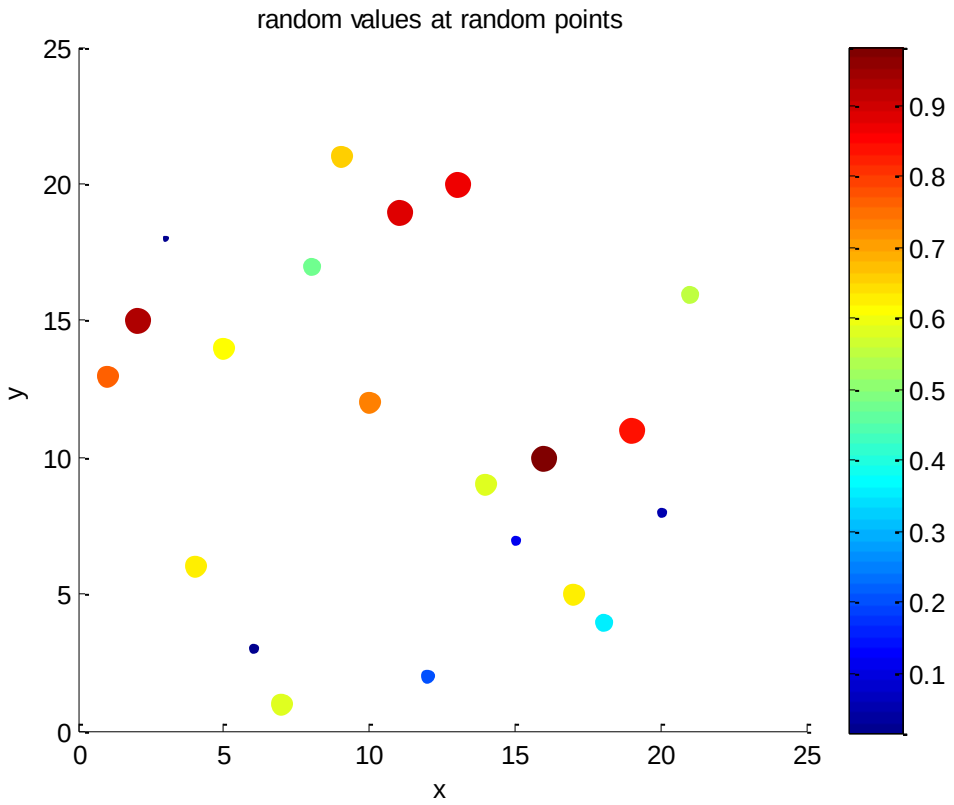


- `Scatter` is a function you might find useful in later labs. Just like `imagesc`, it represents the z-dimension as color (again, you can select the colormap) but does so only in prespecified points – i.e.

  ```
  scatter(x,y,circleSizes,colorValues)
  ```

  will represent data appearing on points (x,y) as circles. Each circle's size is determined by circleSizes, and its color by colorValues – we can represent a fourth dimension if we want to! **All those inputs are 1-dimensional vectors in our examples.**

  Plot the points of the vector z2, given in the corresponding positions (x2,y2). Adjust their color to be 'proportional' to the z2 values. If you want to experiment, you can do the same with size, or you can just select a uniform size for all circles. You can also fill them if you want (`help scatter`). An example result is given in the next page. Your answer should end up having the same format, but, of course (due to randomness) different data.


random values at random points
25
20
15
10
5
0
y
0
5
10
15
20
25
x
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1