MATLAB Tutorial

The following tutorial is an adaptation and integration, according to our purposes, of the one you can find at "https://it.mathworks.com/help/matlab/getting-started-with-matlab.html?lang=en".

What is Matlab?

Matlab is an IDE and a programming language primarily usedfor numerical computation, simulation, and data analysis. The name "Matlab" is an acronym for "MATrix LABoratory," which reflects its strong emphasis on **matrix operations** and numerical data manipulation. In particular, this will allow us to perform analysis and operations on networks in an efficient way by means of their adjacency matrices.

Desktop basics:

The desktop includes the following panels:

- **Current folder:** This window shows the files and directories in your current working directory. You can navigate through your file system and manage your Matlab files here.
- **Command window:** The Command Window is where you can enter Matlab commands and scripts. It's an interactive interface where you can perform calculations, run scripts, and see the results.
- **Workspace:** This allows to explore data you create (such as variables) or import from files.

As you work with MATLAB, you issue commands that create variables and call functions. For example, create the following variables by typing these statements at the command line:

```
a = 1
```

```
a = 1
```

```
b = 2
```

```
b = 2
```

```
c = a + b
```

```
c = 3
```

```
d = cos(a)
```

```
d = 0.5403
```

Note that all these variables are stored in the workspace.

When you do not specify an output variable, MATLAB uses the variable `ans`, short for *answer*, to store the results of your calculation:

```
sin(a)
```

```
ans = 0.8415
```

If you end a statement with a semicolon, MATLAB performs the computation, but suppresses the display of output in the Command Window:

```
e = a*b;
```

Note that no output is displayed here on the right.

Matrices and arrays:

All MATLAB variables are multidimensional *arrays*, no matter what type of data. A *matrix* is a two-dimensional array often used for linear algebra.

To create an array with four elements in a single row, separate the elements with either a comma (`,`) or a space:

```
a = [1 2 3 4]
```

```
a = 1×4
     1     2     3     4
```

```
%equivalently:
% a = [1,2,3,4]
```

This type of array is a *row vector* with dimensions (1,4).

To create a matrix that has multiple rows, separate the rows with semicolons.

```
a = [1 3 5; 2 4 6; 7 8 10]
```

```
a = 3×3
     1     3     5
     2     4     6
     7     8    10
```

Alternatively, you can use newlines to separate rows:

```
a = [1 3 5
    2 4 6
    7 8 10]
```

```
a = 3×3
     1     3     5
     2     4     6
     7     8    10
```

Another way to create a matrix is to use a function, such as `ones`, `zeros`, or `rand`. For example, create a 5-by-1 column vector of zeros:

```
z = zeros(5,1)
```

```
z = 5×1
     0
     0
     0
     0
     0
```

```
x = ones(5,1)
```

```
x = 5×1
     1
     1
     1
     1
     1
```

```
y = rand(5,1)
```

```
y = 5×1
    0.9706
    0.9572
    0.4854
    0.8003
    0.1419
```

`zeros` and `ones` create constant matrices with the specified dimensions where each entry is respectively 0 or 1. `rand` samples values in the (0,1) interval from an uniform distribution.

If you want to generate matrices of random integers you should use the `randi` function:

```
r = randi([10,50],1,5)
```

```
r = 1×5
    27    47    42    49    36
```

Here the first argument specifies the interval (extemes are included), while the other two arguments specify the dimensions of the matrix.

Other simple matrix you can easily create include the identity matrix and diagonal matrices:

```
eye(6)
```

```
ans = 6×6
     1     0     0     0     0     0
     0     1     0     0     0     0
     0     0     1     0     0     0
     0     0     0     1     0     0
     0     0     0     0     1     0
     0     0     0     0     0     1
```

```
diag(r)
```

```
ans = 5×5
    27     0     0     0     0
     0    47     0     0     0
     0     0    42     0     0
     0     0     0    49     0
     0     0     0     0    36
```

Matrix and arrays operations:

MATLAB allows you to process all of the values in a matrix using a single arithmetic operator or function. In this way the operation or function is broadcast to each element of the matrix:

```
a + 10
```

```
ans = 3×3
    11    13    15
    12    14    16
    17    18    20
```

```
sin(a)
```

```
ans = 3×3
    0.8415    0.1411   -0.9589
    0.9093   -0.7568   -0.2794
    0.6570    0.9894   -0.5440
```

Of course, you can also perform traditional binary operations between matrices:

```
s = a+a
```

```
s = 3×3
     2     6    10
     4     8    12
    14    16    20
```

```
p = a*a
```

```
p = 3×3
    42    55    73
    52    70    94
    93   133   183
```

```
inv(a) %inv finds the inverse of its input matrix (it must be a non-singular square matrix)
```

```
ans = 3×3
    4.0000   -5.0000    1.0000
  -11.0000   12.5000   -2.0000
    6.0000   -6.5000    1.0000
```

```
i = a*inv(a)
```

```
i = 3×3
    1.0000    0.0000   -0.0000
         0    1.0000   -0.0000
         0    0.0000    1.0000
```

Notice that `i` is not a matrix of integer values. MATLAB stores numbers as floating-point values, and arithmetic operations are sensitive to small differences between the actual value and its floating-point representation.

The transposed of a matrix is obtained by using a single quote:

```
a'
```

```
ans = 3×3
     1     2     7
     3     4     8
     5     6    10
```

The matrix operators for multiplication, division, and power each have a corresponding array operator that operates element-wise. The first example below shows the element-wise product, while the second example raises each element of a to the third power:

```
p = a.*a
```

```
p = 3×3
     1     9    25
     4    16    36
    49    64   100
```

```
a.^3
```

```
ans = 3×3
           1          27         125
           8          64         216
         343         512        1000
```

Concatenations:

Concatenation is the process of joining arrays to make larger ones. In fact, you made your first array by concatenating its individual elements. The pair of square brackets [] is the concatenation operator.

```
A = [a,a]
```

```
A = 3×6
     1     3     5     1     3     5
     2     4     6     2     4     6
     7     8    10     7     8    10
```

```
%as with numbers, you can concatenate matrices with whitespaces: A = [a a]
```

Concatenating arrays next to one another using commas is called *horizontal* concatenation. Each array must have the same number of rows. Similarly, when the arrays have the same number of columns, you can concatenate *vertically* using semicolons:

```
A = [a; a]
```

```
A = 6×3
     1     3     5
     2     4     6
     7     8    10
     1     3     5
     2     4     6
     7     8    10
```

Array Indexing and Sorting:

The most common ay to refer to a particular element in an array is to specify its row and column. Note that the numbers of rows and columns starts with 1:

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
```

```
A = 4×4
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

```
A(4,2)
```

```
ans = 14
```

To refer to multiple elements of an array, use the colon operator, which allows you to specify a range of the form `start:end`. For example, list the elements in the first three rows and the second column of `A`:

```
A(1:3,2)
```

```
ans = 3×1
     2
     6
    10
```

The colon alone, without start or end values, specifies all of the elements in that dimension. For example, select all the columns in the third row of `A`:

```
A(3,:)
```

```
ans = 1×4
     9    10    11    12
```

The colon operator also allows you to create an equally spaced vector of values using the more general form `start:step:end`.

```
B = 0:10:100
```

```
B = 1×11
     0    10    20    30    40    50    60    70    80    90   100
```

In order to sort arrays, use `sort:`

```
v = [2 4 3 6 0 0.5];
[w, isort] = sort(v)
```

```
w = 1×6
         0    0.5000    2.0000    3.0000    4.0000    6.0000
isort = 1×6
     5     6     1     3     2     4
```

Now `w` contains the elements of `v` in *ascending* order. The vector *isort* describes the arrangement of the elements of `v` into `w`.

Saving and Loading:

Workspace variables do not persist after you exit MATLAB. Save your data for later use with the `save` command.

Saving preserves the workspace in your current working folder in a compressed file with a `.mat` extension, called a MAT-file.

To clear all the variables from the workspace, use the `clear` command.

Restore data from a MAT-file into the workspace using `load`.

```
save myfile.mat
```

```
clear
```

```
load myfile.mat
```

Calling Functions:

MATLAB® provides a large number of functions that perform computational tasks. Functions are equivalent to *subroutines* or *methods* in other programming languages.

There is a very large number of functions already implemented in Matlab. You may want to check at: "https://it.mathworks.com/help/matlab/referencelist.html?type=function&category=getting-started-with-matlab&s_tid=CRUX_topnav".

To call a function, such as `max`, enclose its input arguments in parentheses:

```
A = [1 3 5];
max(A)
```

```
ans = 5
```

Return output from a function by assigning it to a variable:

```
maxA = max(A)
```

```
maxA = 5
```

When there are multiple output arguments, enclose them in square brackets:

```
[minA,maxA] = bounds(A)
```

```
minA = 1
maxA = 5
```

To call a function that does not require any inputs and does not return any outputs, type only the function name:

```
clc
```

The `clc` function clears the Command Window.

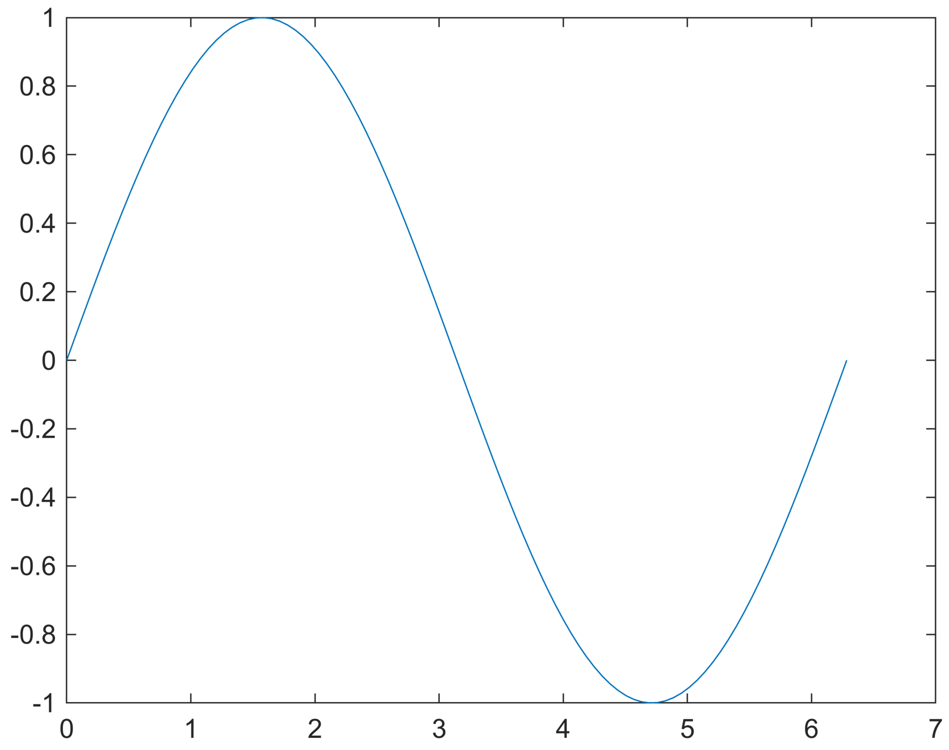
Plotting:

For a more extensive guide to plotting, check the link at "https://it.mathworks.com/help/matlab/ref/plot.html".

Line Plots:

To create two-dimensional line plots, use the `plot` function.

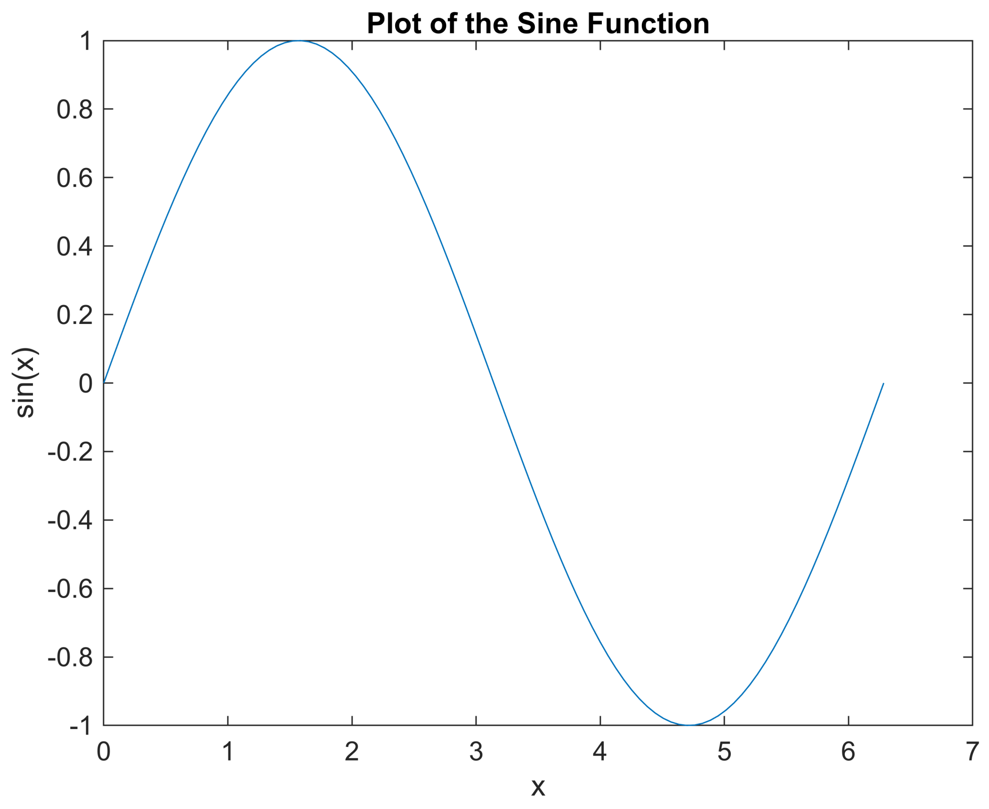
```
x = linspace(0,2*pi);
y = sin(x);
plot(x,y)
```



Here `linspace(x1, x2, n)` generates a vector of `n` evenly spaced numbers between x1 and x2 (included). If `n` is not specified, then 100 numbers are created.

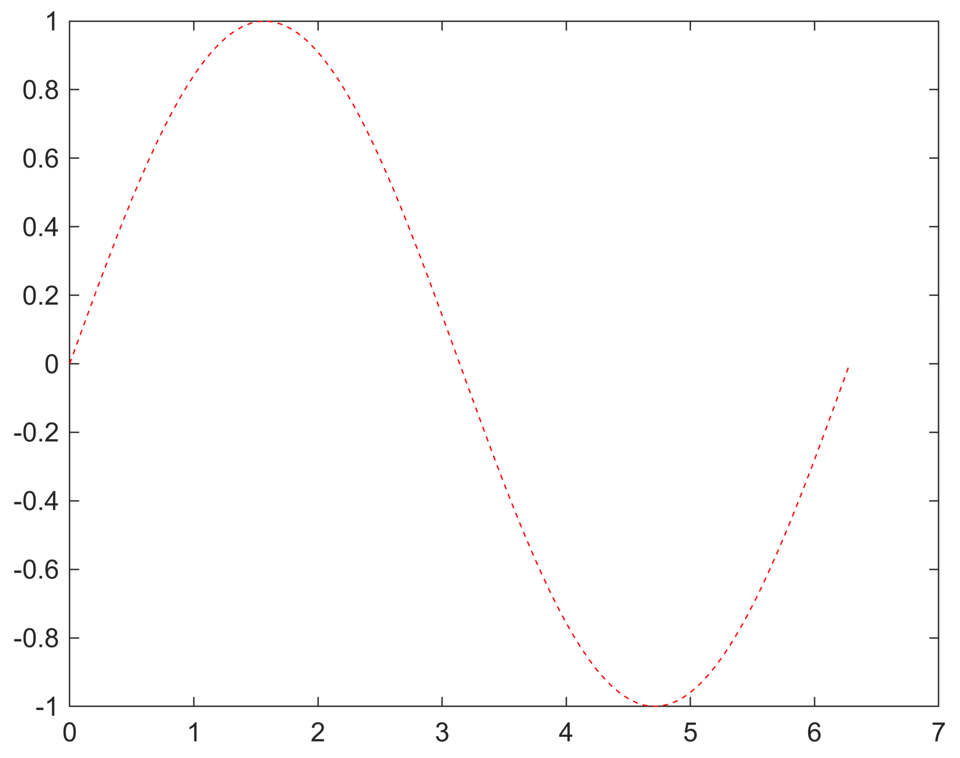
You can label the axes and add a title:

```
xlabel("x")
ylabel("sin(x)")
title("Plot of the Sine Function")
```



By adding a third input argument to the `plot` function, you can plot the same variables using a red dashed line.

```
plot(x,y,"r--")
```



At the link above you can find a list of several different line specifications that can be given to the plot function.

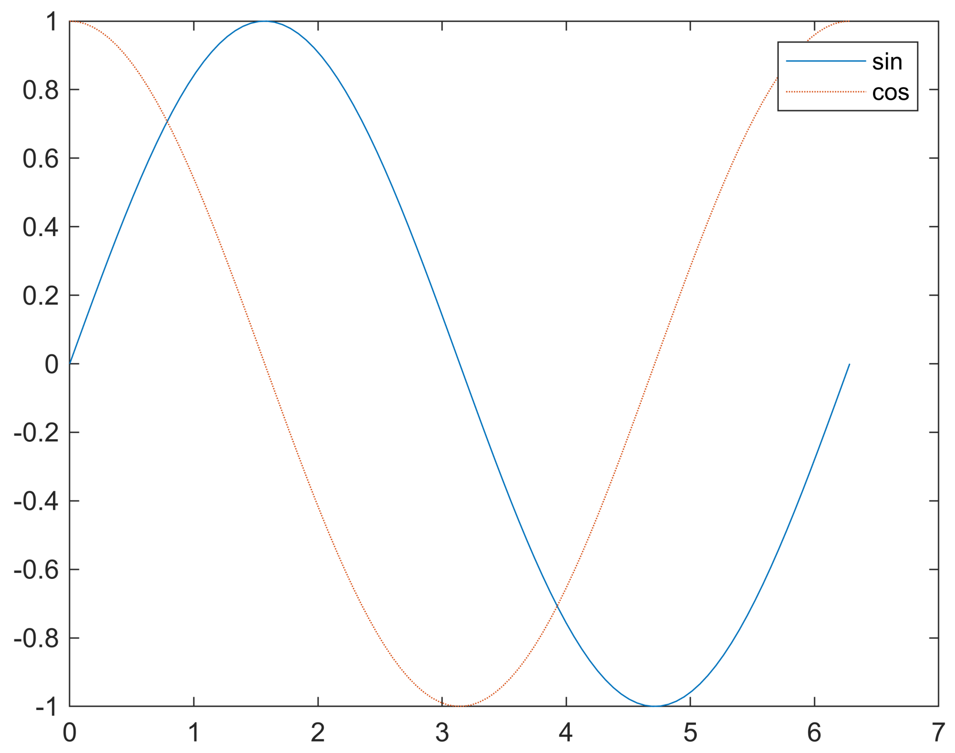
To add plots to an existing figure, use `hold on`. Until you use `hold off` or close the window, all plots appear in the current figure window.

```
x = linspace(0,2*pi);
y = sin(x);
plot(x,y)

hold on

y2 = cos(x);
plot(x,y2,":")
legend("sin","cos")

hold off
```



Programming and Scripts:

The simplest type of MATLAB® program is called a *script*. A script is a file that contains multiple sequential lines of MATLAB commands and function calls. You can run a script from you Current Folder by typing its name at the command line.

Remember that ";" characters at the end of lines are not necessarily required, but most often than not you may want to use them in order to suppress outputs.

Loops:

Loops are useful for creating sequences. For example, create a script named `fibseq` that uses a `for` loop to calculate the first 100 numbers of the Fibonacci sequence. In this sequence, the first two numbers are 1, and each subsequent number is the sum of the previous two, *Fn* = *Fn-1* + *Fn-2*.

```
N = 100;
f(1) = 1;
f(2) = 1;

for n = 3:N
    f(n) = f(n-1) + f(n-2);
end
f(1:10)
```

```
ans = 1×10
```

```
     1     1     2     3     5     8    13    21    34    55
```

To programmatically exit the loop, use a `break` statement. To skip the rest of the instructions in the loop and begin the next iteration, use a `continue` statement.

Conditionals:

Conditional statements execute only when given expressions are true. For example, assign a value to a variable depending on the size of a random number: `'low'`, `'medium'`, or `'high'`. In this case, the random number is an integer between 1 and 100.

```
num = randi(100)
```

```
num = 4
```

```
if num < 34
   sz = 'low'
elseif num < 67
   sz = 'medium'
else
   sz = 'high'
end
```

```
sz = 
'low'
```

Working example: Dolphin

Importing data

Let's analyze the *doplhin* network. This is a very well-known undirected, unweighted graph in network science, in which nodes represent dolphins and they are connected by edges if they have been seen frequently swimming together. In order to follow this tutorial, you should download the csv dataset from "https://networks.skewed.de/net/dolphins#None_draw" and put the contents in your current folder. Then rename all files by anteposing the prefix "dolphin_" to each file.

After that, double click *dolphin_edges.csv* in the current folder tab, opening the import panel. In the chooser in the 'Output Type' section on top of the window, select column vectors. Then select the relevant columns in the shown data and click 'Import selection'. You should obtain two 159x1 vectors "Source" and "target" in your workspace.

Additionaly, import the names of the dolphins by selecting the column `label` while importing `dolphins_nodes.csv`.

Check the Source and target vectors: notice that indices start from 0: instead, we want them to start from 1.

```
Source = Source + 1
```

```
Source = 159×1
```

```
     9
    10
    10
    11
    11
    14
    14
    14
    15
    15
     ⋮
```

```
target = target + 1
```

```
target = 159×1
     4
     6
     7
     1
     3
     6
     7
    10
     1
     4
     ⋮
```

Warning: in general, when importing data from external datasets, be sure to check what is actually represented in there, and think about how you should "pre-process" your data.

Instantiating the graph:

There are multiple ways to instantiate a graph. Note that you can obtain a graph by its adjacency matrix or adjacency list and viceversa. For a comprehensive list of ways to instantiate graphs check this link "https://it.mathworks.com/help/matlab/ref/graph.html". For our purposes, the most useful ways to do so are:

- `G = graph(A),` where A is an adjacency matrix;
- `G = graph(s, t),` where `s` and `t` specify pairs of nodes' indices.

If you want or have to use directed graphs, use `digraph` instead in the same way.

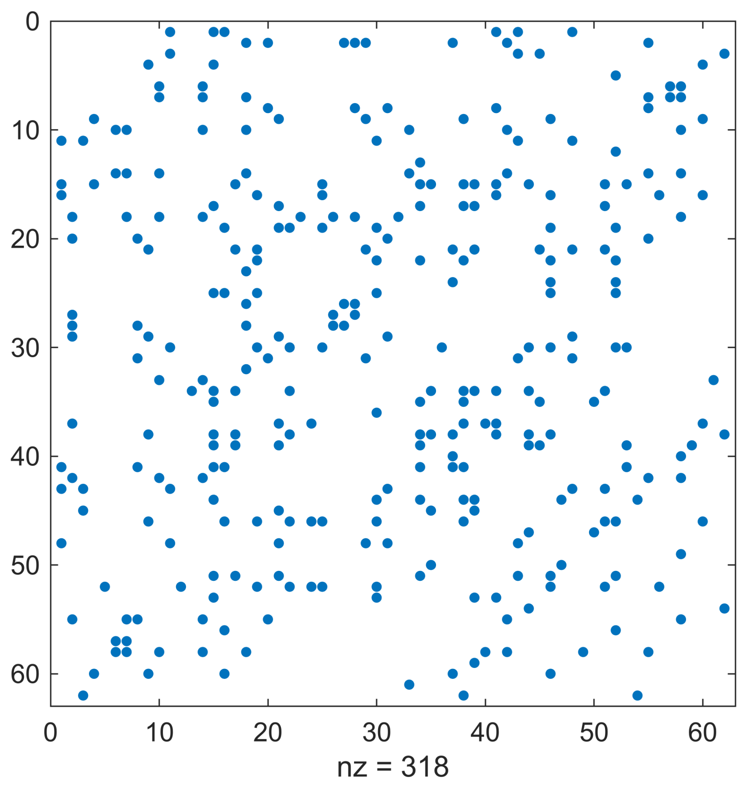
```
G = graph(Source, target)
```

```
G = 
  graph with properties:

    Edges: [159×1 table]
    Nodes: [62×0 table]
```

Now we can obtain its adjacency matrix:

```
A = adjacency(G);
spy(A)
```



Spy plots the sparsity pattern of its matrix input. Nonzero values are colored while zero values are white. `nz` displays the number of nonzero elements in the matrix.

Note that A is symmetric: this is to be expected since `graph` generates **undirected** graphs.

Checking the Degrees and plotting:

Let's check the degrees of the nodes and who is the dolphins with most "friends":

```
e = ones(62,1);
d = A*e %d is the vector of the degrees
```

```
d = 62×1
     6
     8
     4
     3
     1
     4
     6
     5
     6
     7
     ⋮
```

```
[dsort, isort] = sort(d, 'descend')
```

```
dsort = 62×1
    12
    11
    11
    10
    10
     9
     9
     9
     9
     8
     ⋮
isort = 62×1
    15
    38
    46
    34
    52
    18
    21
    30
    58
     2
     ⋮
```

```
dsort(1)
```

```
ans = 12
```

By definition, dsort(1) is the maximum of the degrees in the newtork. We can check the name of the dolphin:

```
label(isort(1))
```

```
ans = 
"Grin"
```

Of course, we can also ask for the top 3 nodes with highest degree:

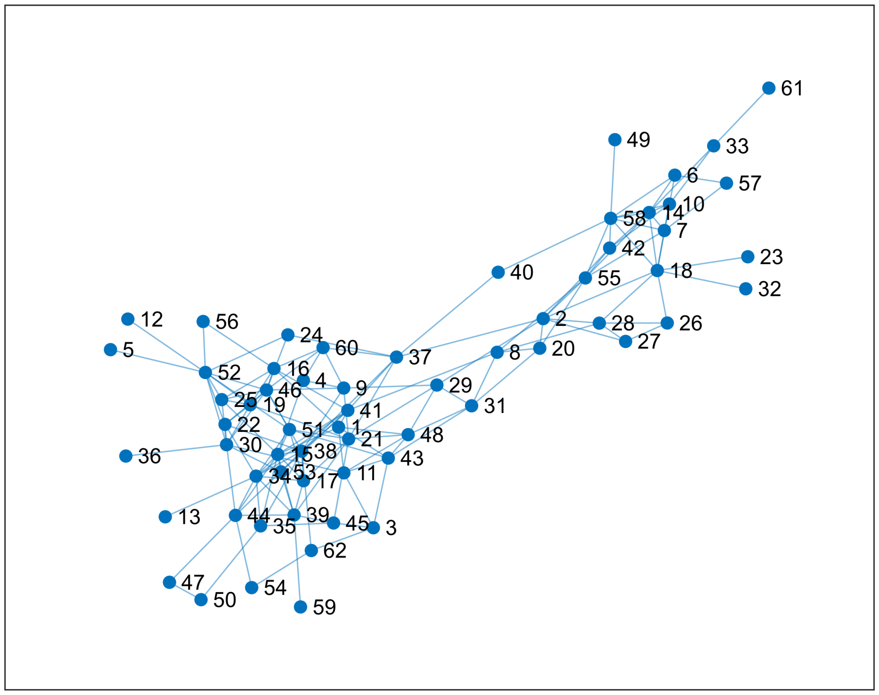
```
label(isort(1:3))
```

```
ans = 3×1 string
"Grin"
"SN4"
"Topless"
```

Connected Components:

Now we can plot our network. We use the `plot` functions for graphs. You may want to check "https://it.mathworks.com/help/matlab/ref/graph.plot.html" for more details about it. For example, several different layouts are available, such as `circle` and `force`. which produces a plot by using the Fruchterman-Reingold algorithm.

```
plot(G)
```



In this case, the layout is chosen automatically based on various factors such as the size and structure of the graph.

We can already visualize that our graph is fully connected. We can also check this via `conncomp`:

```
[bins, binsize] = conncomp(G)
```

```
bins = 1×62
     1     1     1     1     1     1     1     1     1     1     1     1     1···
binsize = 62
```

```
max(bins)
```

```
ans = 1
```

`bins = conncomp(G)` returns the *connected components* of graph `G` as `bins`. The bin numbers indicate which component each node in the graph belongs to.

`binsize` contains the sizes of each of the connected components (in order).

Be careful when dealing with *directed* graphs: If `G` is a directed graph, then two nodes belong to the same strong component only if there is a path connecting them **in both directions.** If instead you want to identify weak connected components, you can either simmetrize your graph, or use `weak_bins = conncomp(G,'Type','weak')`.

Had multiple components been identified, you might want to keep only the largest connected component (depending on our purposes). In this case, a solution would be to use `find.` If the largest component is the one identified by bin `i`, then you can create a vector of indices:

```
% I = find(bins == i)
% Anew = A(I,I)
% Gnew = graph(A)
```

Clustering coefficient:

By exploting the adjacency matrix of `G` we can compute the clustering coefficient of each node and, consequently, of the whole newtork. We can do so by exploiting the fact that the entries (i,j) of the k-th power of an adjacency matrix are the number of paths from node i to node j of length exactly k:

```
npairs = d.*(d-1)
```

```
npairs = 62×1
    30
    56
    12
     6
     0
    12
    30
    20
    30
    42
     ⋮
```

```
diagA3 = diag(A^3)
```

```
diagA3 =
   (1,1)       10
   (2,1)        8
   (3,1)        2
   (4,1)        2
   (6,1)        6
   (7,1)       16
   (8,1)        4
   (9,1)        8
  (10,1)       22
  (11,1)        8
  (14,1)       28
  (15,1)       34
  (16,1)       10
  (17,1)       18
  (18,1)       16
  (19,1)       22
  (20,1)        6
  (21,1)       10
  (22,1)       16
  (24,1)        2
  (25,1)       16
  (26,1)        4
  (27,1)        4
  (28,1)        8
  (29,1)        6
  (30,1)       18
```

```
   (31,1)        6
   (33,1)        2
   (34,1)       30
   (35,1)        6
   (37,1)        2
   (38,1)       26
   (39,1)       16
   (41,1)       14
   (42,1)       12
   (43,1)       10
   (44,1)       10
   (45,1)        2
   (46,1)       34
   (48,1)       12
   (51,1)       10
   (52,1)       22
   (53,1)        4
   (55,1)       16
   (58,1)       26
   (60,1)        6
```

```
clusteringcoeff = zeros(62,1);
for i = 1:62
    if npairs(i) == 0
        clusteringcoeff(i) = 0;
    else
        clusteringcoeff(i) = diagA3(i)/npairs(i);
    end
end
clusteringcoeff
```

```
clusteringcoeff = 62×1
    0.3333
    0.1429
    0.1667
    0.3333
         0
    0.5000
    0.5333
    0.2000
    0.2667
    0.5238
      ⋮
```

Let's check the maximum clustering coefficient in the network:

```
max(clusteringcoeff)
```

```
ans = 0.6667
```

```
find(clusteringcoeff == max(clusteringcoeff))
```

```
ans = 2×1
    26
    27
```

The clustering coefficient of the whole network is then:

```
clusteringnetwork = mean(clusteringcoeff)
```

```
clusteringnetwork = 0.2590
```

Centrality measures:

A rudimental measure of centrality in a network is the degree of nodes, which we have already seen.

Matlab gives you an easy wat to compute directly several centrality measures of nodes in a graph. This can be done via the `centrality` function (check it in the documentation since it has several useful options).

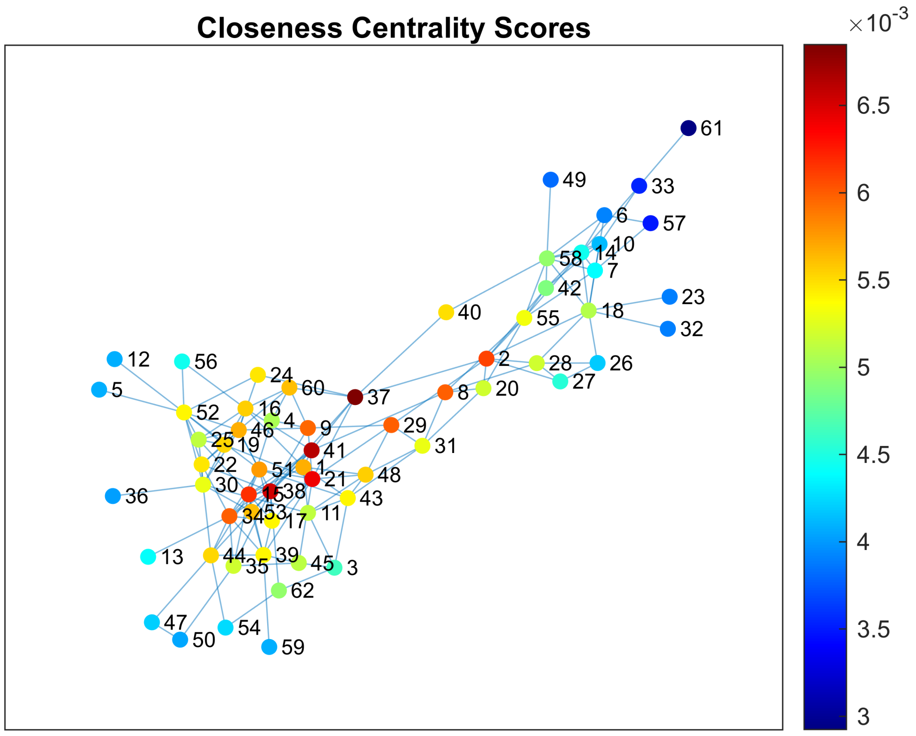
Let's begin with the **closeness** centrality:

```
ucc = centrality(G,'closeness')
```

```
ucc = 62×1
    0.0057
    0.0061
    0.0046
    0.0051
    0.0041
    0.0039
    0.0044
    0.0060
    0.0060
    0.0041
      ⋮
```

We can visualize the results as well:

```
p = plot(G,'MarkerSize',5);
p.NodeCData = ucc;
colormap jet
colorbar
title('Closeness Centrality Scores')
```

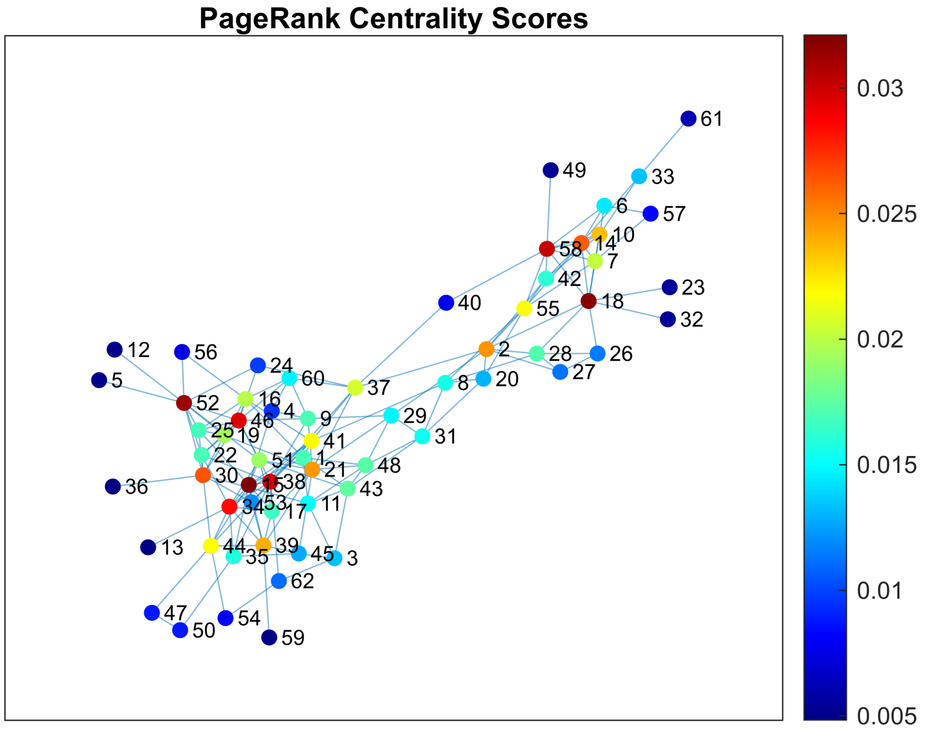


Another remarkably important centrality measure is the **eigenvector** centrality and its variants (such as **PageRank**):

```
pgr = centrality(G,'pagerank')
```

```
pgr = 62×1
    0.0169
    0.0247
    0.0133
    0.0096
    0.0051
    0.0145
    0.0201
    0.0157
    0.0171
    0.0235
      ⋮
```

```
p = plot(G,'MarkerSize',5);
p.NodeCData = pgr;
colormap jet
colorbar
title('PageRank Centrality Scores')
```


PageRank Centrality Scores
0.03
0.025
0.02
0.015
0.01
0.005