

MATHSOC's Interactive MATLAB Guide - v1 (T2 2019)

INTRODUCTION

Hiya! Welcome to UNSW MathSoc's interactive MATLAB guide, created by James Gao and Jeffrey Yang.

In this interactive guide, we'll be going through **4** main self-guided mini-projects: **(1) Predator and Prey Modelling**; **(2) Eigenvalue and Eigenvector Solver**; **(3) Markov Chain Stationary Distribution Solver** (beyond the scope of first year maths) and **(4) Graph and Network Algorithms** (MATH1081 content).

The ideology behind these projects is the full coverage of the MATLAB Self-Paced modules required for MATH1251 students to peruse, plus a little bit extra. Hopefully through this, they can do some revision whilst gaining a better understanding of the beauty and intricacies within MATLAB. We shall start off with a quick review of basic MATLAB commands, and more advanced information and functionality will be introduced later when relevant.

MATLAB COMMANDS REVIEW

Special thanks goes to Dr Quoc Thong Le Gia and his MATH2301 course materials which much of the below content are adapted from.

We first note that every in-built function in MATLAB comes with their respective documentation that can be accessed through the `help` function from the command line. This is done by typing `help` followed by the name of the function in question, for example:

```
help help
```

`help` Display help text in Command Window.

`help` NAME displays the help for the functionality specified by NAME, such as a function, operator symbol, method, class, or toolbox. NAME can include a partial path.

If NAME is not specified, help displays content relevant to your previous actions.

Some classes require that you specify the package name. Events, properties, and some methods require that you specify the class name. Separate the components of the name with periods, using one of the following forms:

```
help CLASSNAME.NAME
help PACKAGENAME.CLASSNAME
help PACKAGENAME.CLASSNAME.NAME
```

If NAME is the name of both a folder and a function, `help` displays help for both the folder and the function. The help for a folder is usually a list of the program files in that folder.

If NAME appears in multiple folders on the MATLAB path, `help` displays information about the first instance of NAME found on the path.

NOTE:

In the help, some function names are capitalized to make them

stand out. In practice, type function names in lowercase. For functions that are shown with mixed case (such as `javaObject`), type the mixed case as shown.

EXAMPLES:

```
help close           % help for the CLOSE function
help database/close  % help for CLOSE in the Database Toolbox
help database        % list of functions in the Database Toolbox
                    % and help for the DATABASE function
help containers.Map.isKey % help for isKey method
```

See also `doc`, `docsearch`, `lookfor`, `matlabpath`, `which`.

Reference page for help

It is a often good idea to refer back to the documentation if you are unsure about syntax, functionality, arguments, exceptions or even if you just need a quick refresher.

Vectors

Vectors in MATLAB are denoted by square brackets, $vec = [x, y, z; a, b, c]$, wherein the commas separate elements horizontally and semicolons separate the elements of the vector vertically.

```
A = [1,2,3]; B = [1;2;3]; C = [1,2,3; 4,5,6; 7,8,9];
```

In the above example, A is a row vector, B is a column vector and C is a 3×3 matrix. If you would like to try yourself, please enter the code in the gray box below and click the green 'Run' button in the 'LiveEditor' tab above.

```
% Type your commands below:
```

Note that the indexing of a element in a vector is done through round brackets, `()`, at the end of the vector, for example $C(2,2) = 5$.

Functions

Function files in MATLAB, called (M-files) allow one to create ones own MATLAB commands. A function file is a separate file that has these lines:

```
function [yout1, yout2, ...] = funcname(xin1, xin2, ...)
% comment line 1
% comment line 2
% ...
yout1 = ...;
yout2 = ...;
...
end
```

To run a function, we merely have to call on it using the defined name and input variables of choice for instance $[output1, output2, \dots] = funcname(x, y, \dots)$ in the command line, where the results of the function will be deposited as a vector with variable names $output1, output2, \dots$. This is the same as any in-built function like $y = sqrt(x)$.

Notes:

- The function M-file names and the function name that appears in the first line of the file of the file should be identical. In reality, MATLAB ignores the function name in the first line and executes functions based on the file name.
- All variables inside a function are local and are erased after execution of the function.

It is good to note that there is another class of functions called **anonymous functions** which can be defined straight at the MATLAB command line, but also in a M-file if you so choose. As you can imagine, anonymous functions are much less powerful and versatile in functionality, and they are of the form:

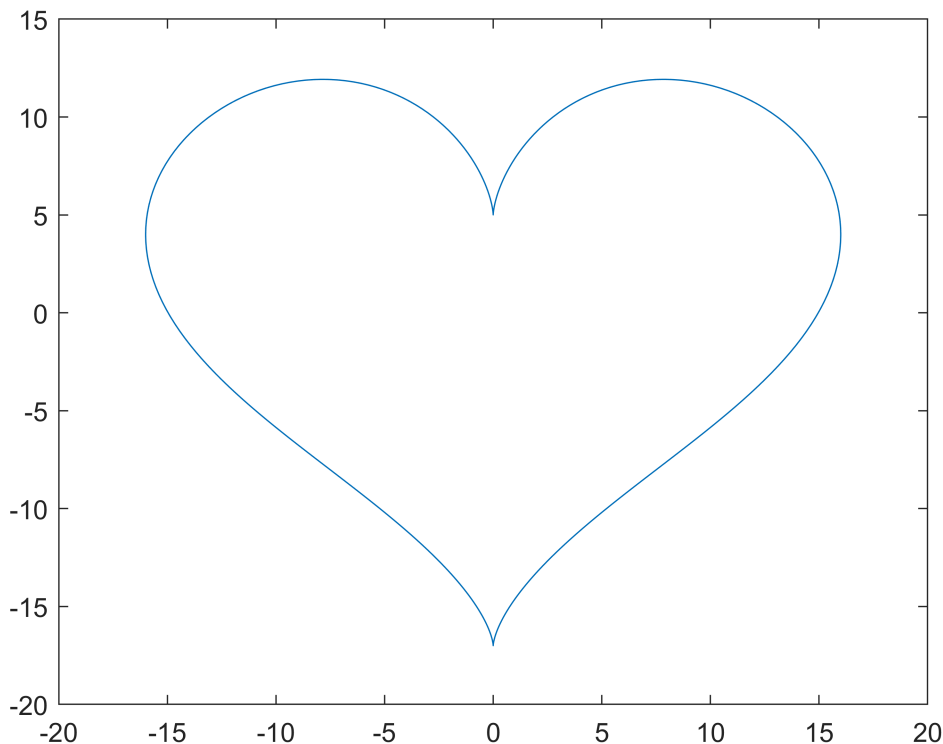
```
fun1 = @(x) x.^2;  
fun2 = @(x,y) x+y;
```

To see how we may use this, let us consider a parametric equation with implicit definitions:

```
x_vals = @(t) 16*(sin(t)).^3;  
y_vals = @(t) 13*cos(t) - 5*cos(2*t) - 2*cos(3*t) - cos(4*t);
```

Defining an interval t , we can thus plot the curve through the given implicit functions:

```
t = linspace(-10,10,1000);  
%generate a row vector of 1000 points linearly spaced between -10 and 10  
plot(x_vals(t), y_vals(t))
```



In this guide, we will be going the step-by-step creation of 2 function files that will be separately included for reference.

MORE MATLAB COMMANDS

Conditional Statements:

Conditional statements (**if statements**) provide a mechanism for branching in a computer program. They are in the following form:

```
if <logical expression 1>
    <statement 1>
elseif <logical expression 2>
    <statement 2>
else
    <statement 3>
end
```

Zero or more than one *elseif* branch is permitted. The *else* branch can also be omitted.

Loop Statements:

A loop is a programming construct in which a group of statements is executed repeated.

For Loops has the form:

```
for n = 1:N %a value for N has to be pre-defined
    <statement>
end
```

While Loops has the form:

```
while <logical expression>
    <statement>
end
```

For practice, try outputting the first 1000 square numbers below:

```
%Type your commands below:
```

SECTION 1: PREDATOR - PREY MODELLING

Special thanks goes to Dr John Murray and his MATH6781 course materials which much of the below content are adapted from.

Drawing a parallel to chemical equations and the law of mass action, the population law of mass action has been empirically formulated to describe how two populations interact,

The rate of change of one population due to the interaction with another is proportional to the product of the two populations.

In its simplest form, the predator-prey relationship can be modelled through **DEs** as:

$$\frac{dx}{dt} = -ax + bxy$$

$$\frac{dy}{dt} = cy - dxy$$

Intuitively, we can see that the x population is the **predator** population as without interaction with the **prey** population y , the rate of change of the x population is negative as evidenced by the negative sign in front of the ax term, $a, b, c, d \geq 0$.

This means that if left alone, population x naturally decreases and y naturally increases. Correspondingly, x population increases if allowed to interact with y population (positive xy coefficient), and x population will decrease with mutual interaction, in line with the behaviour of the prey population.

More complicated versions of the model can be developed through the adding and modification of x and y terms to simulate: overcrowding; cooperation; satiation etc... Today we will be focusing on this particular relationship:

$$\frac{dx}{dt} = -x + \frac{xy}{10}$$

$$\frac{dy}{dt} = y - \frac{xy}{5}$$

TLDR: We will be modelling the system described by the above differential equations.

Here, we will be modelling the solution orbits on the x, y phase plane, around a steady state (quantity of x, y that does not change with time), which in this case turns out to be $(5, 10)$ - trust me.

```
ss = [5,10]; %[c/d, a/b]
%generate a grid of values around the steady state
x0 = linspace(0, 5*ss(1), 20);
y0 = linspace(0, 5*ss(2), 20);
[X, Y] = meshgrid(x0,y0);
```

Next we calculate the values that x, y takes over a certain time horizon.

```
U = (1-0.2*X).*Y; %the prey derivative
V = (0.1+1*Y).*X; %the predator derivative
%Note that '.' is needed before any operators to perform element wise
%operation.

xinit=[1 2 5; 2 2 2]; %initial x and y values for the orbit curve
tspan=[0 10]; %time horizontal to plot the curves on

%we use ode solver to calculate trajectories of x and y values
```

```

for i=1:3
    [t,x]=ode45(@predprey_de,tspan, xinit(:,i));
    xsol(i)=[t,x];
end

```

For plotting, we will use the *quiver* plot function:

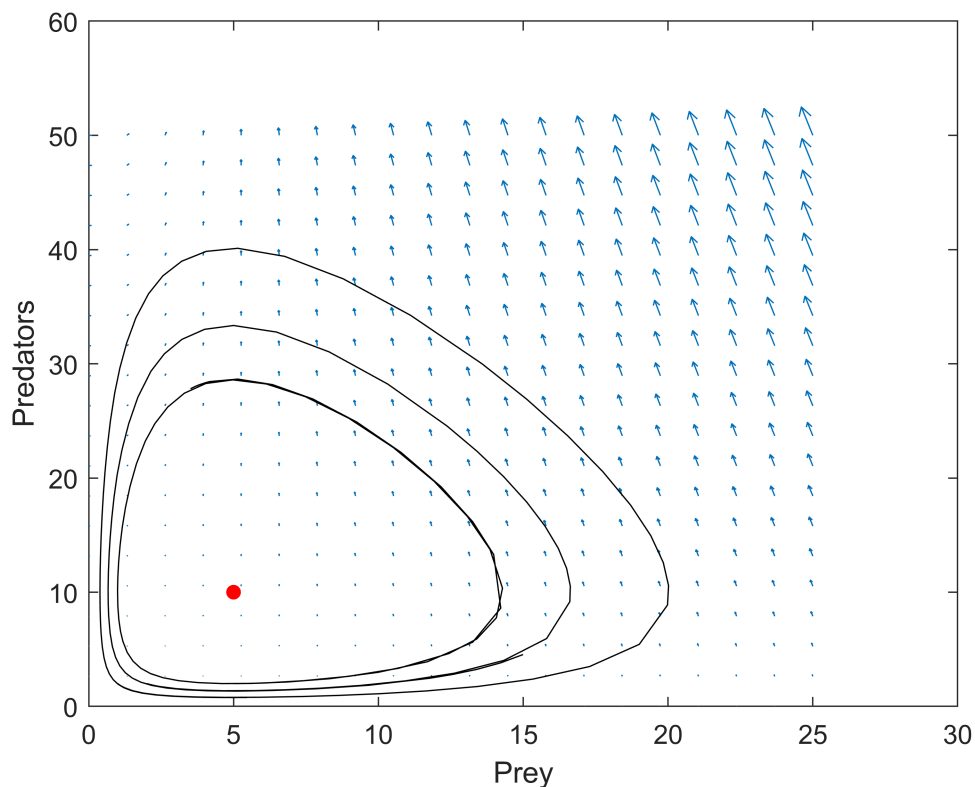
```

quiver(X,Y,U,V)
hold on
plot(5,10,'ro','MarkerFaceColor','r','MarkerSize',5) %steady state

for i=1:3
    sol=xsol{i};
    x=sol(:,2);
    y=sol(:,3);
    plot(x,y,'k-')
end

hold off
xlabel('Prey')
ylabel('Predators')

```



Here, the black line represents the trajectory of predator and prey population over time, whilst the blue arrows are the magnitude and direction of change at each point of population. This is governed by the DEs that we described early and the bigger the arrow, the bigger the change over time.

This kind of circular orbit portrait - a linear planar autonomous system - is classified as a **Centre** or **Vortex**. Note that the population of prey and predators follow a fixed cycle over time in a closed system and any changes in either will simply bump them into a different layer of curve instead of along the particular curve.

If the mathematics behind this kind of modelling is of particular interest to you, check out MATH2221, MATH3201 and MATH6781.

SECTION 2: EIGENVALUE AND EIGENVECTOR SOLVER

Now, let's get started with our **Eigenvalue and Eigenvector Solver**. We will attempt to re-create the in-built `eig()` MATLAB function from formulae and steps derived in MATH1251 for the simplest 2×2 matrix as a demonstration.

Remember from lectures that to find eigenvalues, we need to first find values of λ which satisfies the **characteristic equation** of a square matrix A , namely those values of λ for which $\det(A - \lambda I) = 0$, where I is the 2×2 **identity matrix**.

Note that the characteristic polynomial of a 2×2 matrix is: $P_2(x) = (A_{11}A_{22} - A_{12}A_{21}) - x(A_{11} + A_{22}) + x^2$. Knowing this, we can easily create a function to find and solve for the values of x in the characteristic polynomial. That will be done using the `solve` function, which offers both symbolic and numeric equation solvers. The easiest version is `solve(eqn, x)` which we will use below to solve `eqn` for variable x .

```
function [solx] = eigenvalues(A)
    syms x
    a = 1; b = -(A(1,1) + A(2,2)); c = A(1,1)*A(2,2) - A(1,2)*A(2,1);
    eqn = a*x^2 + b*x + c == 0;
    solx = solve(eqn,x)
end
```

To see this in action, first let us define the 2×2 matrix A :

```
A = [0,1; -2,-3];
```

Then, we can use the `solve` function to find the solutions to the given characteristic polynomial, this is:

```
syms x
a = 1; b = -(A(1,1) + A(2,2)); c = A(1,1)*A(2,2) - A(1,2)*A(2,1);
eqn = a*x^2 + b*x + c == 0;
solx = solve(eqn,x)
```

```
solx =
```

$$\begin{pmatrix} -2 \\ -1 \end{pmatrix}$$

Note that we can check the solutions by finding the roots of the polynomial through the quadratic equation:

```
a = 1; b = -(A(1,1) + A(2,2)); c = A(1,1)*A(2,2) - A(1,2)*A(2,1);
```

$$x1 = (-b + \sqrt{b^2 - 4*a*c})/(2*a)$$

$$x1 = -1$$

$$x2 = (-b - \sqrt{b^2 - 4*a*c})/(2*a)$$

$$x2 = -2$$

Clearly, our two eigenvalues are -1 and -2 . Once the **eigenvalues** of the matrix A has been found, we can find the corresponding **eigenvectors** by Gaussian Elimination. For each eigenvalue, we have $(A - \lambda I)x = 0$. Essentially, we are looking for the nullspace of the matrix $(A - \lambda I)$, which can be found using the `null()` function in MATLAB. Note that we cannot use the `A\b` function because the matrix $(A - \lambda I)$ is rank-deficient, thus the solution is not unique and also often displayed as $(0;0)$.

```
[m, n]= size(A);
A1 = A - solx(1,1)*eye(m); b1 = zeros(1,m)';
A2 = A - solx(2,1)*eye(m); b2 = zeros(1,m)';
eig1 = null(A1)
```

eig1 =

$$\begin{pmatrix} -\frac{1}{2} \\ 1 \end{pmatrix}$$

```
eig2 = null(A2)
```

eig2 =

$$\begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

Thus, the final version of our solver function is:

```
function [solx, eig1, eig2] = eigenvalues(A)
    syms x
    a = 1; b = -(A(1,1) + A(2,2)); c = A(1,1)*A(2,2) - A(1,2)*A(2,1);
    eqn = a*x^2 + b*x + c == 0;
    solx = solve(eqn,x)

    [m, n]= size(A);
    A1 = A - solx(1,1)*eye(m); b1 = zeros(1,m)';
    A2 = A - solx(2,1)*eye(m); b2 = zeros(1,m)';
    eig1 = null(A1)
    eig2 = null(A2)
end
```

As we said previously, MATLAB has an inbuilt function for this which is `eig()`. Run it below to check the validity of our solution:

```
[V, D] = eig(A)
```



```
V = 2x2
    0.7071    -0.4472
   -0.7071     0.8944
D = 2x2
    -1     0
     0    -2
```

This clumsy method can be easily extended to any $n \times n$ matrix assuming you can be bothered to go through and find the characteristic polynomial through solving for the determinant. However, it should be clear that this is not how the function `eig` does it, which is way quicker and more efficient.

SECTION 3: MARKOV CHAIN STATIONARY DISTRIBUTION SOLVER

This last section is dedicated with love to the students from my ACTL2102 PASS class whom I have made solve for endless amounts of stationary distributions.

A **Markov Process** has the **Markov Property** which can be precisely stated as:

$$\mathbb{P}(X(t_n) \leq x_n | X(t_1) = x_1, X(t_2) = x_2, \dots, X(t_{n-1}) = x_{n-1}) = \mathbb{P}(X(t_n) \leq x_n | X(t_{n-1}) = x_{n-1})$$

for $t_1 < t_2 < \dots < t_n$. That is, given the present state, the past states do not have influence on the future.

A **Markov Chain** is a **Markov Process** on a discrete index set $T = 0, 1, 2, \dots$ which we will denote by $X_n, n = 0, 1, 2, \dots$.

For an irreducible (only one class, all states communicate), ergodic (positive recurrent, aperiodic) Markov Chain, $\lim_{n \rightarrow \infty} P_{ij}^n$ exists and is independent of i . Thus, π_i , the long run proportion of time the Markov Chain is in state i can be calculated using the equations: $\pi P = \pi$; $\sum_i \pi_i = 1$, where $\pi = (\pi_1, \pi_2, \dots, \pi_n)$ and P the probability transition matrix.

TLDR: We want to solve for $\pi = (\pi_1, \pi_2, \dots, \pi_n)$ in the equation $\pi P = \pi$.

We first define the matrix P that we will use for our following calculations:

```
P = [0.6,0.3,0.1;0.4,0.4,0.2;0.2,0.1,0.7];

% [m,n] creates a vector storing the row (m) and column size (n) of the matrix P
[m, n] = size(P);
```

We can then create a function that outputs the stationary distribution like magic:

```
function[stat_dist] = stat_dist_solver(P)
    [m, n] = size(P);
    Q = P' - eye(m); %eye(n) creates a nxn identity matrix
    R = [Q(1:(m-1),:); ones(1,m)];
    V = [zeros(1,m-1),1]';
    stat_dist = R\V
end
```

Here, the Q matrix is created by expanding $\pi P = \pi$ into a set of n simultaneous equations with π_n on the LHS. Moving it around, we will have $Q\pi = 0$. Since we have n equations to solve for n unknowns, when we add the condition $\sum_i \pi_i = 1$, we can delete the last line of Q creating R below where $R\pi = V$. Solving through the backslash operator, we obtain the stationary distribution. This is demonstrated below:

```
Q = P' - eye(m); %eye(n) creates a nxn identity matrix
R = [Q(1:(m-1),:); ones(1,m)];
V = [zeros(1,m-1),1]';
stat_dist= R\V
```

```
stat_dist = 3x1
    0.4211
    0.2632
    0.3158
```

To display the values in a clearer format, we can use a For loop:

```
for n = 1:size(stat_dist)
    fprintf('pi_%i = %.20f\n', n, stat_dist(n,1))
end
```

```
pi_1 = 0.42105263157894745607
pi_2 = 0.26315789473684214617
pi_3 = 0.31578947368421039776
```

SECTION 4: GRAPH AND NETWORK ALGORITHMS

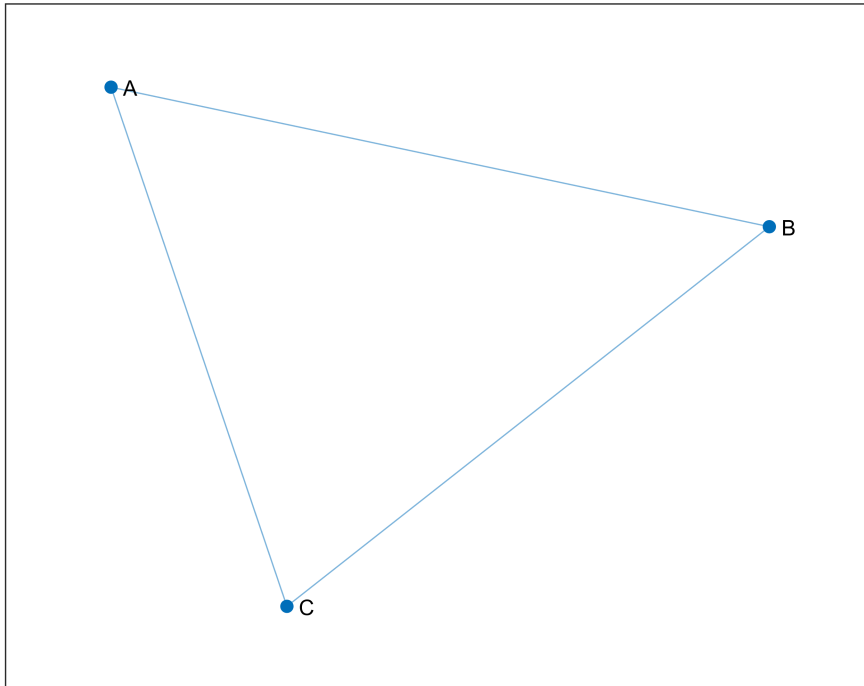
This section is intended for students who have done MATH1081 Discrete Mathematics. Graph theory and network theory have a wide range of applications in computer science and the natural sciences, as well as in linguistics and the social sciences. Note that the functions discussed below are chosen specifically to be of relevance to those studying/ have studied MATH1081 and that MATLAB has a lot more tools for working with graphs.

We begin with an introduction on how to construct and visualise directed and undirected graphs in MATLAB before discussing the graph algorithms that are available as functions in MATLAB.

Recall that a graph consists of a set of nodes (or vertices) that are connected by edges. Graphs may be directed (if the edges each have a direction) and weighted (if the edges each have a weight). To create a graph, you can either provide the adjacency matrix or the edge list. We present both methods below:

Adjacency Matrix

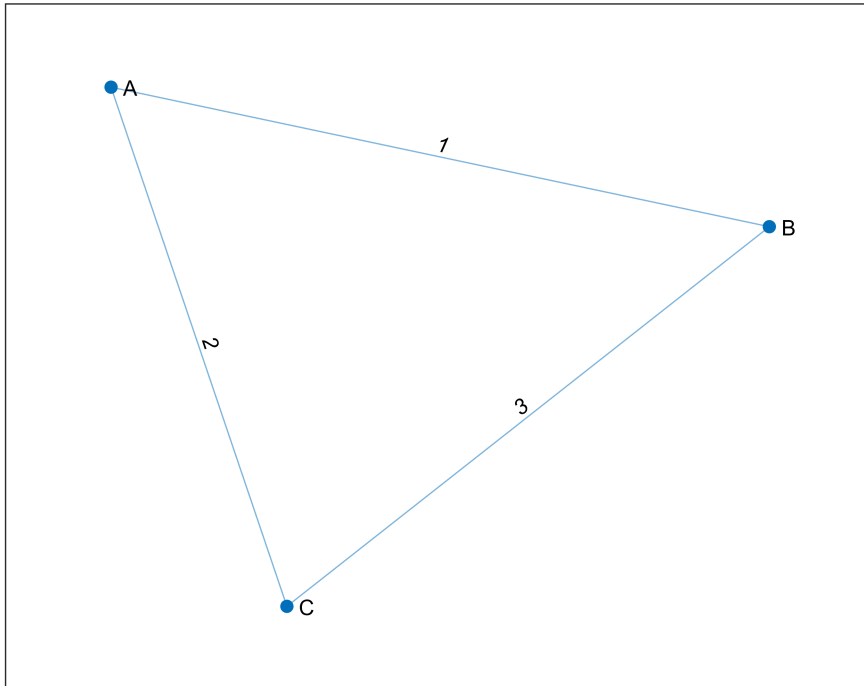
```
A = [0 1 2; 1 0 3; 2 3 0];
node_names = {'A','B','C'};
G = graph(A,node_names);
H = plot(G);
```



Note that the adjacency matrix method cannot create a multigraph (graphs with multiple edges between the same source and target node pair) and will instead assign weights to edges accordingly.

Edge List

```
source_nodes = {'A','A','B'};  
target_nodes = {'B','C','C'};  
edge_weights = [1 2 3]; % Weights of edges AB, AC and BC respectively  
G = graph(source_nodes, target_nodes, edge_weights);  
H = plot(G);  
labeledge(H,1:numedges(G),edge_weights);
```

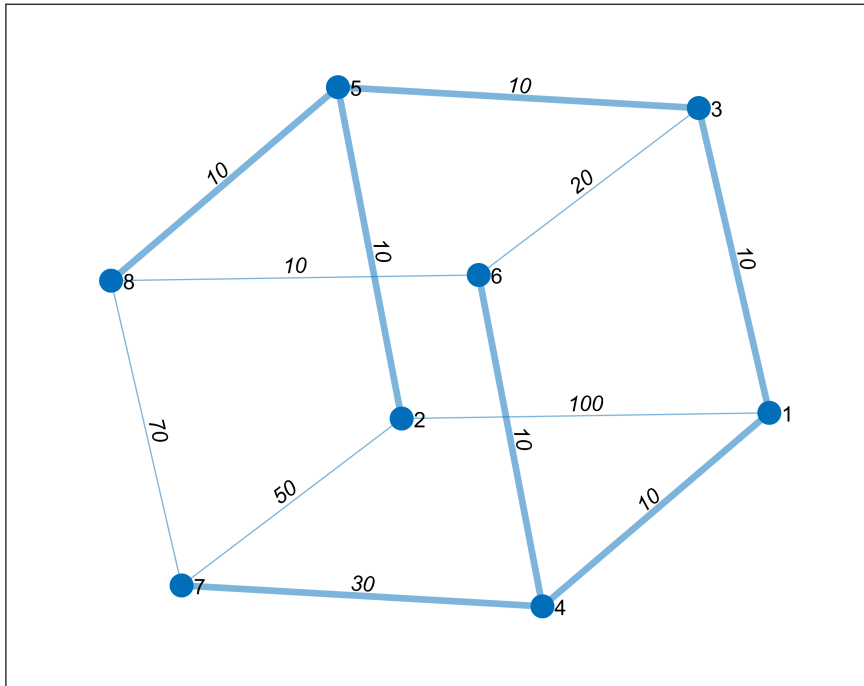


Notice that *plot* labels nodes automatically but not edges. We can use the *labeledge* function to label the edges with their corresponding weights or write `plot(G, 'EdgeLabel', G.Edges.Weight)`. There exists a range of functions to modify the nodes and edges of an existing graph as well as extracting subgraphs.

We will now go through how to construct a minimum spanning tree and how to determine if two graphs are isomorphic.

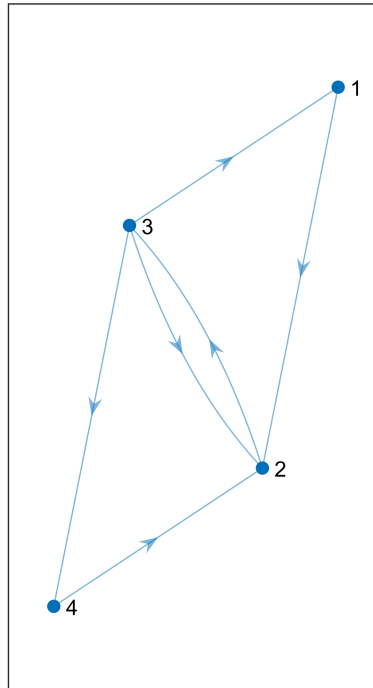
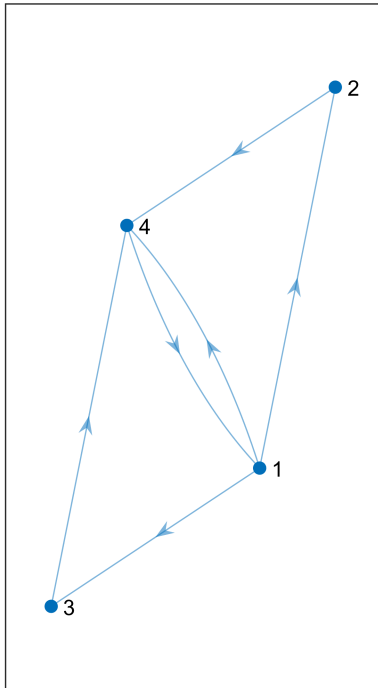
Minimum Spanning Tree

```
s = [1 1 1 2 5 3 6 4 7 8 8 8];
t = [2 3 4 5 3 6 4 7 2 6 7 5];
weights = [100 10 10 10 10 10 20 10 30 50 10 70 10];
G = graph(s,t,weights);
p = plot(G, 'EdgeLabel', G.Edges.Weight);
[T, pred] = minspantree(G); % The minimum spanning tree is stored in T.
% 'pred' here is short for predecessor nodes and specifies a directed version of the minimum spanning tree
highlight(p, T) % Highlights the nodes and edges of graph T. T must have the same nodes and a spanning tree
```



Isomorphic

```
G1 = digraph([1 1 1 2 3 4],[2 3 4 4 4 1]);
G2 = digraph([3 3 3 2 2 1 4],[1 4 2 3 2 2]);
subplot(1,2,1) % Creates axes in tiled positions
plot(G1)
subplot(1,2,2)
plot(G2)
```



```
p = isomorphism(G1,G2) % p returns a vector of edge permutations if G1 and G2 are isomorphic and
```

```
p = 4x1
     3
     1
     4
     2
```

ENDING REMARKS

First of all, thanks for reading all the way up to here, if you have any questions, suggestions or spot any mistakes, please do not hesitate to send me an email or send MathSoc a message on FB at:

jamesgao985@gmail.com
<https://www.facebook.com/unswmathsoc/>

I'll just end this by saying that the functionalities and possibilities of MATLAB is much much more than what we have seen today. In future versions of the guide, I hope to be able to go into greater details about things of greater interest to me like exploring packages, classification algorithms and graphing with real time data. Thanks again and cya!