Rehearsal Lab

Contents

1 Getting Started

This lab serves as a rehearsal of the basic MATLAB programming concepts covered in the TNSL24 course. It is intended to help you refresh your MATLAB skills before proceeding to the subsequent labs.

For each subsequent lab, please study the relevant algorithm in advance using the video lectures available on the course website and come prepared to work on the assigned tasks.

2 Policy on AI Assistance

The use of AI tools (e.g., ChatGPT or MATLAB Copilot) is permitted in the Basic logistics algorithms course as a support for learning. For example, AI may be used to explain algorithms or help identify errors.

However, AI tools should not be used simply to generate complete solutions to assignments. You are expected to develop the solution yourself,

critically evaluate any AI-generated content, and ensure that you understand all parts of your submitted work. You must be able to justify your code independently during evaluation sessions.

If you use any AI tool, briefly describe how you used it in comments within your MATLAB solution scripts.

3 Vectors and matrices warm-up

Vectors and matrices are fundamental data structures in MATLAB. A vector is a one-dimensional array of values, while a matrix is a two-dimensional array organised into rows and columns.

A row vector can be created using square brackets. Its elements can be accessed, modified, or removed using indices:

```
v = [1 2 3 4 5];

v(3)            % third element
v([1 3 5])      % first, third, and fifth elements
v(2:4)          % elements from index 2 to index 4

v(2) = 10;      % modify the second element
v(3) = [];      % remove the third element

length(v)       % number of elements
```

The colon operator can also be used to create a regularly spaced vector:

```
a = 1:5         % [1 2 3 4 5]
```

A matrix can be created by separating elements in the same row with spaces or commas and separating rows with semicolons:

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

Matrix elements, rows, columns, and submatrices can be accessed using row and column indices:

```
A(2, 3)         % element in row 2 and column 3
```

```
A(2, :)        % second row
A(:, 3)        % third column
A([1 3], :)    % first and third rows
A(:, [1 2])    % first and second columns
```

The size of a matrix can be obtained using the `size` function. A matrix filled with zeros and having the same size as an existing matrix can be created using `zeros`:

```
size(A)
Z = zeros(size(A));
```

4 `for` and `while` loops warm-up

Loops are used to repeat a group of commands.

A `for` loop repeats commands for a specified sequence of values:

```
for i = 1:5
    disp(i)
end
```

The loop variable can also take the values stored in a vector:

```
v = [4 7 2 8];

for value = v
    disp(value)
end
```

In this example, `value` takes each value stored in `v`, one at a time.

Nested `for` loops can be used to go through the elements of a matrix:

```
A = [
    1 2 3;
    4 5 6
];

for i = 1:size(A, 1)
    for j = 1:size(A, 2)
        disp(A(i, j))
    end
end
```

The outer loop goes through the rows, while the inner loop goes through the columns.

A `while` loop repeats commands as long as a condition is true:

```
i = 1;

while i <= 5
    disp(i)
    i = i + 1;
end
```

The loop variable must be initialized before the loop and updated inside it. Otherwise, the loop may continue indefinitely.

Use a `for` loop when the sequence of values or the number of iterations is known. Use a `while` loop when repetition should continue until a condition changes.

5 Visualising graphs warm-up

MATLAB provides the `graph` and `digraph` objects for representing undirected and directed graphs. A graph can be created from an adjacency matrix or an edge list and displayed using the `plot` function.

Plotting a single graph

The following adjacency matrix represents an undirected graph:

```
A = [
    0 1 0 1;
    1 0 1 0;
    0 1 0 1;
    1 0 1 0
];

G = graph(A);

figure;
plot(G);
```

For a directed graph, use the `digraph` function:

```
A = [
    0 1 0 1;
    0 0 1 0;
    0 1 0 0;
    0 1 1 0
];

G = digraph(A);

figure;
plot(G);
```

The command `figure` opens a new figure window. The command `close all` closes all open figure windows and is often used at the beginning of a script:

```
clear;
close all;
```

Plotting several graphs

Use `figure` before each `plot` command to display graphs in separate figure windows:

```
figure;
plot(G);

figure;
plot(G2);
```

Alternatively, use `pause` to display graphs successively in the same figure window:

```
plot(G);
pause;
plot(G2);
```

The `pause` command stops the execution of the program until the user continues it.

Storing edge lists

An edge list can be stored in an $n \times 2$ matrix. Each row contains the two endpoints of one edge. For example, the edges $(1, 2)$, $(2, 3)$, $(2, 1)$, and $(3, 4)$ can be stored as:

```
edge_list = [
    1 2;
    2 3;
    2 1;
    3 4
];
```

The first column contains the starting vertices and the second column contains the ending vertices.

Plotting a graph using an edge list

A graph can be created directly from the two columns of an edge list:

```
edge_list = [
    1 2;
    2 3;
    2 1;
    3 4;
    1 4
];

G = graph(edge_list(:, 1), edge_list(:, 2));

figure;
plot(G);
```

Use `digraph` instead of `graph` to create a directed graph:

```
G = digraph(edge_list(:, 1), edge_list(:, 2));
```

Plotting a graph using an edge list and edge weights

Edge weights can be stored in a vector. The order of the weights must correspond to the order of the edges in the edge list:

```
edge_list = [
    1 2;
    2 3;
    2 1;
    3 4;
    1 4
];

edge_weights = [1 1 2 2 3];

G = graph(edge_list(:, 1), edge_list(:, 2), ...
          edge_weights);

figure;
plot(G, 'EdgeLabel', G.Edges.Weight);
```

Plotting a graph using an edge list and node labels

Names can be assigned to graph nodes:

```
edge_list = [
    1 2;
    2 3;
    2 1;
    3 4
];

G = graph(edge_list(:, 1), edge_list(:, 2));

G.Nodes.Name = {
    'First'
    'Second'
    'Third'
    'Fourth'
};

figure;
plot(G);
```

Adding edges to a graph

The `addedge` function adds an edge to an existing graph. For example, the following command adds the edge $(1, 3)$ with weight 4:

```
G = addedge(G, 1, 3, 4);
```

Adding nodes to a graph

The `addnode` function adds nodes to an existing graph. The following command adds one new node:

```
G = addnode(G, 1);
```

After this operation, the graph contains one more node than before.

Removing edges from a graph

The `rmedge` function removes an edge specified by its endpoints:

```
G = rmedge(G, 2, 1);
```

An edge can also be removed using its number in the graph's edge table:

```
G = rmedge(G, 3);
```

For graphs with named nodes, the node names can be used:

```
G = rmedge(G, 'Second', 'First');
```

Removing nodes from a graph

The `rmnode` function removes a node and all edges connected to it:

```
G = rmnode(G, 1);
```

For graphs with named nodes, the node name can be used:

```
G = rmnode(G, 'First');
```

6 Tasks

Task 1

Create the following vector:

```
v = [1 4 -2 7 -4 5 10 15];
```

Perform the following operations:

- Display the third element of `v`.
- Display the elements at indices 2, 4, and 7.
- Display all elements from the third to the sixth element.
- Change the fourth element to 20.
- Display the number of elements in `v`.

Then use a `for` loop to go through the values stored in the modified vector and display each value. Use the following form of the loop:

```
for value = v
    % display value
end
```

In this loop, the variable `value` takes, one by one, all values stored in `v`.

Task 2

Create an adjacency matrix for an **undirected** graph with 6 vertices. Plot a graph using this adjacency matrix. Create an edge list for the same graph. Plot a graph using this edge list. Check that it is the same graph.

Task 3

Modify the script from task 2 so that it works with a **directed** graph, using an adjacency matrix representing a directed graph of your choice.

Task 4

1. Write a program that converts the adjacency matrix of a directed graph into an edge-list representation. Test your program using several graphs and plot each graph.

2. Write a program that converts a given edge list into an adjacency matrix. Test it using the edge lists obtained in Part 1 and verify that the resulting adjacency matrices are identical to the original matrices. You may either add the code to the script from Part 1 or create a new script. Plot the resulting graphs.

Hint: if you want to create a new matrix of the same size as an existing one and fill it with zeros, you can use the following code, where `A` is the existing matrix.

```
new_A = zeros(size(A));
```

Hint: in order to plot several graphs use `figure` or `pause`