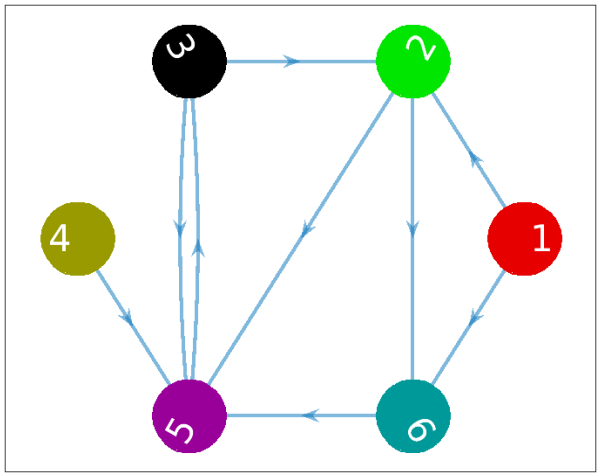
CS319: Scientific Computing (with MATLAB)

Classes and Objects in MATLAB (V2.0)

Niall Madden

Week 11: **9am**, and **4pm**, 22 March 2023



Important: you should read:

- The MATLAB Guide, Chapters 19 and 21: https://doi-org.nuigalway.idm.oclc.org/10.1137/1.9781611974669

This week...

1. 1. Projects (yet again)
2. 2: Classes
   - Encapsulation
   - Implementation
   - `classdef`
3. 3. Example: MyStack
   - Example 1
   - Example 2: palindromes
4. 4. Overloading equality operator
5. 5: Another example - VarioDie
6. 6: Graphs and Networks
7. 6: PageRank

1. Projects (yet again)

The slides for this section are at https://www.niallmadden.ie/2223-CS319/2223-CS319-Projects.pdf

To date, the programming we have done in MATLAB has been *procedural*: it has been driven by algorithms and involved on very simple data structures.

Encapsulation

Idea: create a single entity in a program that combines data with the program code (i.e., functions) that manipulate that data.
In MATLAB, a description/definition of such entities is called a **class**, and an instance of such an entity is called an **object**.

That is, like a variable is a single instance for a `float` (for example), then an object is a single instance of a `class`.

A class should be thought of as an **Abstract Data Type** (ADT): a specialised type of variable that the user can define, which includes

- different types of data fields, called "properties".
- functions, called "methods", which can modify these fields.

There are many important examples of "built-in" MATLAB classes, such as those that represent graph and digraphs (from Graph Theory), and plots.

But we'll leave those until later, and first study how to make our own.

In MATLAB, *encapsulation* is implemented using the `classdef` keyword.

A class has **four** types of members:

- **properties**, i.e, variables, etc, for storing data;
- **methods**, i.e, functions for changing or returning properties.
- **events** (another time, perhaps).
- **enumeration** (ditto).

To define a class, make a `.m` file with the same name as the class, and with code that follows the following syntax:

```
classdef  ClassName
   properties  % new keyword
      PropertyName1
      PropertyName2 ...
   end
   methods      % new keyword
      function obj = methodName(obj,arg2,...)
         ...
      end
   end
end
```

ClassName becomes a new object type—one can now declare objects to be of type *ClassName*.

obj: special variable that refers to the object to which the method is applied.

Some particulars:

1. If a method changes a property, it must return *obj*
2. If there is a method with the same name as the class, it is a **constructor**, and is called automatically when an object is defined. Usually used to initialise variables.
3. The usual assignment operator, =, is automatically defined, but no others are.
4. Operator overloading as achieved by defining certain functions, such as *plus()*, *times()*, *power()*, *lt()*, *eq()*, and many more. These can then be used by name, or by the usual symbol. Full list at:
   https://uk.mathworks.com/help/matlab/matlab_oop/implementing-operators-for-your-class.html
5. Can do inheritance, but we won't study this.

3. Example: MyStack

The example we'll consider is a **stack** – a *LIFO* (Last In First Out) queue.

The name of our class will be `MyStack`.

It has a single *`property`*: an array called *`contents`*.

The class permits two primary operations:

- an item may be added to the top of the stack: `push()` ;
- an item may be removed from the top of the stack: `pop()`.

These then are our interfaces to the stack. Hence these will be **methods**.

In MATLAB, if there is a method that has the same name as the class is it as **constructor**: a function that is called automatically when we make an object belonging to the class. Typically used to initialise variables.

3. Example: MyStack

MyStack.m

```
classdef MyStack
    properties
        contents;
    end
    methods
        function obj = MyStack()
            obj.contents = [];
        end
        function obj=push(obj, p)
            obj.contents = [obj.contents, p];
        end
        function [p, obj]=pop(obj)
            p = obj.contents(end);
            obj.contents = obj.contents(1:end-1);
        end
    end
end
```

3. Example: MyStack

To use this:

```
>> S=MyStack;  % Creating a MyStack object, S
>> S=S.push(12)
S =
  MyStack with properties:
    contents: 12
>> S=push(S, 245)
S =
  MyStack with properties:
    contents: [12 245]
>> [p,S] = S.pop()
p =  245
S =
  MyStack with properties:
    contents: 12
>> [p,S] = pop(S)
p =
    12
S =
  MyStack with properties:
    contents: [1x0 double]
```

3. Example: MyStack

Note that methods can be called in two ways: in this example as *S1.pop()* and also as *pop(S1)*.

TestMyStack.m

```
%% TestMyStack.m
S1 = MyStack;
Code = 'CS319';
for i=1:length(Code)
   S1 = S1.push(Code(i));
end
for i=1:length(Code)
   [Edoc(i), S1] = S1.pop();
end
fprintf("%s backwards is %s\n", Code, Edoc);
```

IsPalindrome.m

```
%% IsPalindrome.m
clear;
S = MyStack;
Word = input('Enter word: ', 's');
for i=1:length(Word)
    S = S.push(Word(i));
end
for i=1:length(Word)
    [Dorw(i), S] = S.pop();
end
if (Word ==  Dorw)
    fprintf("%s is a palindrome.\n", Word);
else
    fprintf("%s is not palindrome.\n", Word);
end
```

4. Overloading equality operator

Very often we wish to extend the definition of standard operators, such as

- `==`
- `+` (two versions: *`plus`* and *`uplus`*) and `-` (same)
- `&`
- Lots more: `https://uk.mathworks.com/help/matlab/matlab_oop/implementing-operators-for-your-class.html`

This is called **overloading**. It is usually a good idea to overload at least the equality operator.

We'll now to that for a new version of the `MyStack` class.

We'll also add a method that returns the number of items on a stack.

4. Overloading equality operator

We'll call the new version *MyStackV2*. The first 18 lines are the same as for *MyStack*.

MyStackV2.m

```
        function i=numitems(obj)
20          i = length(obj.contents);
        end
22      function v = eq(obj, b)
            v = isequal(obj.contents, b.contents);
24      end
```

See also *TestMyStackV2.m* which tests this implementation, and also verifies that one can create an array of objects.

5: Another example - VarioDie

To reinforce ideas, let's introduce a new class called `VarioDie`. [Full disclosure - I stole this idea from Tobias Rossmann, and CS102.]

Our class will represent a d-sided die.

Properties:

- `sides` (number of sides on the die).
- `value` (current value)

Methods:

- `A constructor`. If an argument is passed, we the number of sides to be that value. Otherwise, set it to 6.
- `roll()`. Sets `value` to some random integer.

5: Another example - VarioDie

This example will allow us to introduce some technicalities.

- Properties are usually variables. We can restrict the `size` of a properties (i.e., the number of rows and columns). For us, both `sides` and `value` should the 1×1 matrices, e.g., scalars.
- We can set the datatype of a property (to avoid the default type `double`.
- Similarly, we can force a property to be (for example), positive.
- How to set default values and types for function arguments.

5: Another example - VarioDie

VarioDie.m

```
%% VarioDie : represent a d-sided die
classdef VarioDie
    properties
        sides (1,1) uint32;
        value (1,1) uint32 {mustBePositive} = 1;
    end
    methods
        function obj = VarioDie(d)
            arguments
                d (1,1) uint32 {mustBePositive} = 6;
            end
            obj.sides = d;
        end
        function obj = roll(obj)
            obj.value = randi(obj.sides);
        end
    end
end
```

5: Another example - VarioDie

Now that we have this class, let's use it to determine, experimentally, the most likely outcome of rolling some given number of d-sided dice.

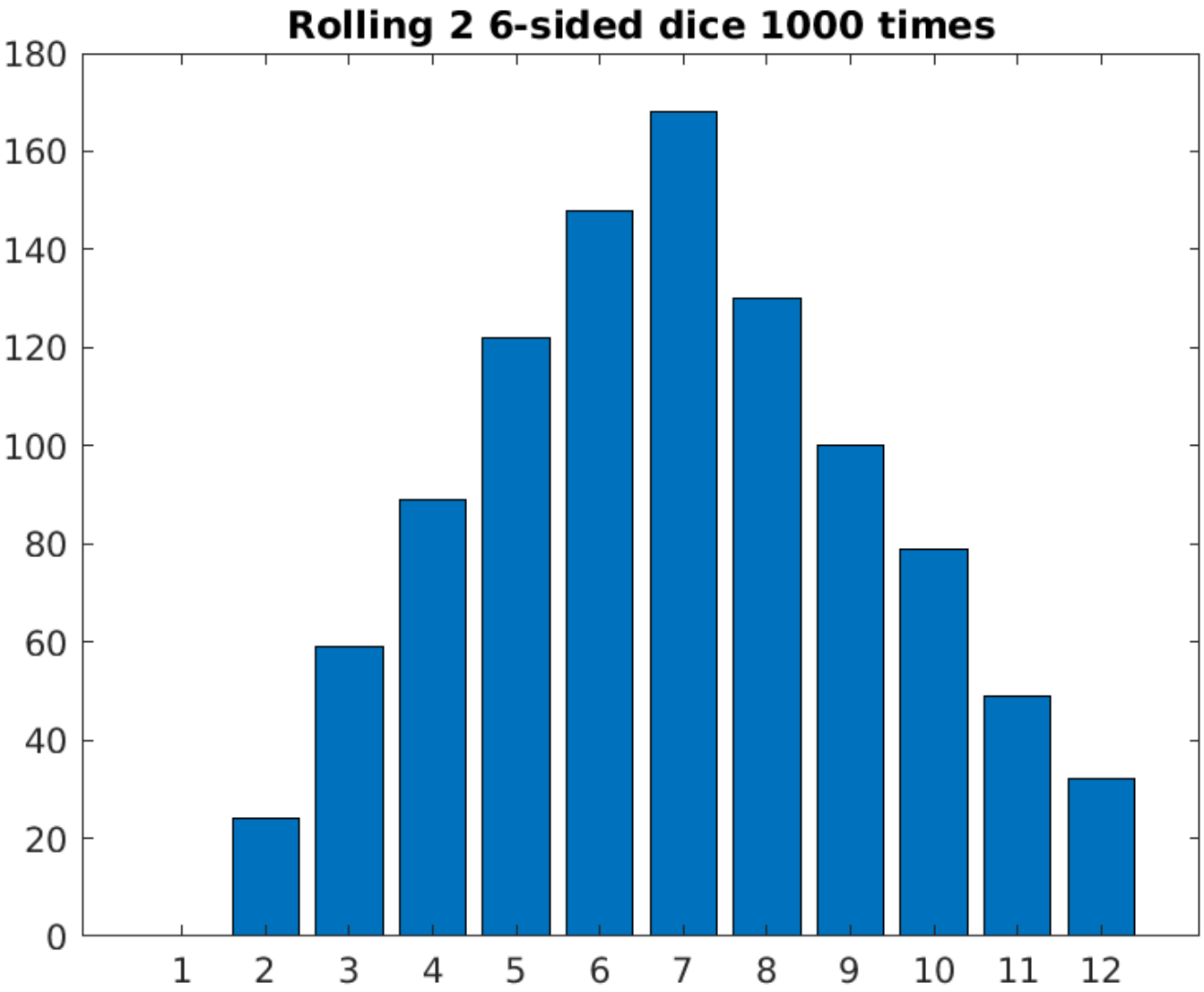
5: Another example - VarioDie

TestVarioDie.m

```
%% Test VarioDie
d = 6; % number of sides
m = 2; % number of dice per roll
t = 1000; % number of rolls

rolls = zeros(1,m*6);
x = VarioDie(d);
for i = 1:t
    total = 0;
    for j = 1:m
        x = x.roll();
        total = total + x.value;
    end
    rolls(total)=rolls(total)+1;
end
bar(rolls)
```

5: Another example - VarioDie



6: Graphs and Networks

MATLAB has many built-in classes. Two of the more obviously useful ones are *graph* and *digraph*, used to define undirected and directed graphs, respectively.

There are then numerous functions of manipulating graphs, performing such operations as

1. Adding and removing edges and vertices (which are called "nodes", in MATLAB).
2. Plotting.
3. Checking for isomophisms.
4. Extracting subgraphs
5. Finding distances between vertices.
6. Lots more...

The remaining part of this section will be presented using a MATLAB live script, *CS319_Week11_Intro_to_graphs.mlx*.

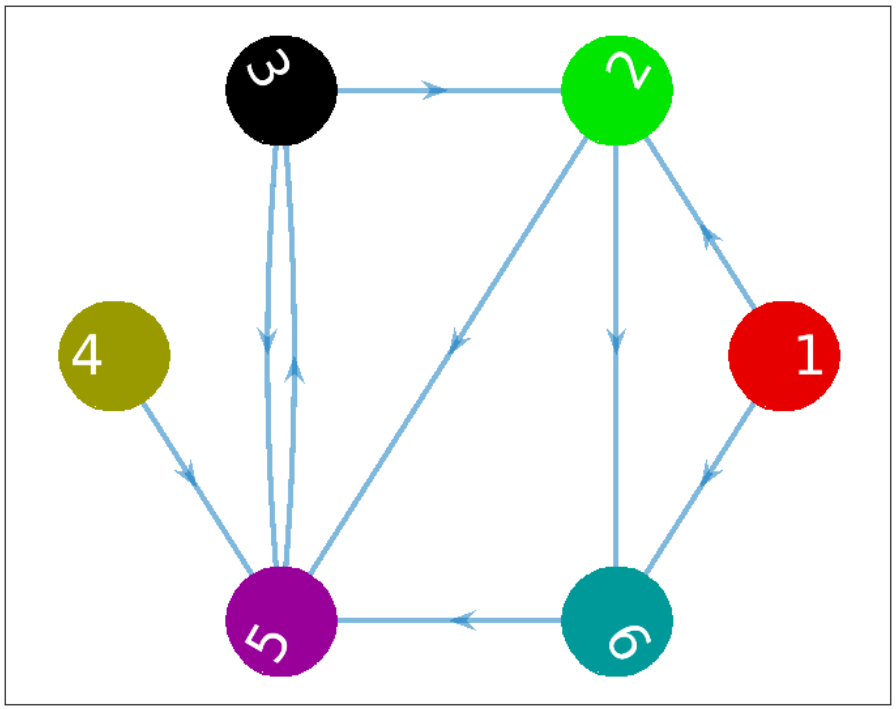
6: PageRank

Google initial break-through in search engine design was derived from their **PageRank** algorithm which gives an objective way of computing the relative importance of web-pages.

The basic idea is this: **the importance of a web-page is the probability that you are looking at it at any given time**.

6: PageRank

To see how this works, consider the following example, which was made in the live script.



6: PageRank

The method:

(1) Form the **adjacency matrix**, $A = (a_{ij})_{i=1}^{N}$, for the network:

$$a_{i,j} = \begin{cases} 1 & \text{if the graph has an edge from Node } i \text{ to Node } j; \\ 0 & \text{otherwise.} \end{cases}$$

(2) Make the associated **Markov matrix**, $S = (s_{ij})_{i=1}^{N}$, where $S_{i,j}$ is the *proportion* of vertices in A which start at i and go to j. (That is, divide the entries in row i by the sum of the entries in that row). If there are no entries in a given row of A, set the corresponding entries of S to $1/N$.

(3) Choose a "damping" value σ, e.g., $\sigma = 0.85$.

(4) Set the matrix G to be $\left(\sigma S + (1-\sigma)/N\right)^T$.

(5) Finally, find a "fixed point" vector x: should be $Gx = x$. This can be done with the **Power Method**. (Yes, it's an eigenvector...).

6: PageRank

From our example earlier, the first few results are:

Iteration	0	1	2	3
u	$\begin{pmatrix} 0.1667 \\ 0.1667 \\ 0.1667 \\ 0.1667 \\ 0.1667 \\ 0.1667 \end{pmatrix}$	$\begin{pmatrix} 0.0250 \\ 0.1667 \\ 0.1667 \\ 0.0250 \\ 0.4500 \\ 0.1667 \end{pmatrix}$	$\begin{pmatrix} 0.0250 \\ 0.1065 \\ 0.4075 \\ 0.0250 \\ 0.3296 \\ 0.1065 \end{pmatrix}$	$\begin{pmatrix} 0.0250 \\ 0.1746 \\ 0.3084 \\ 0.0250 \\ 0.3612 \\ 0.1059 \end{pmatrix}$