SM-2302: Software for Mathematicians

Lecture 4: Advanced Methods

Dr Huda Ramli
Mathematical Sciences, FOS, UBD
`huda.ramli@ubd.edu.bn`

Semester I, 2025/26

Adapted from 6.057 IAP 2019 (`MIT OCW`)

Outline

Lecture 4: Advanced Methods

- Probability and Statistics
- Data Structures
- Images
- File I/O

Statistics

In data analysis, we can compute the statistics using built-in functions like `mean`, `median`, `mode`

Suppose we have some random data: `scores = 100 * rand(1,100);`

`hist(scores,5:10:95);`	makes a histogram with bins centered at $5, 15, 25, \ldots 95$
`hist(scores,20);`	makes a histogram with 20 bins
`N=histc(scores,0:10:100);`	returns the number of occurrences between the specified bin edges 0 to < 10, 10 to $< 20, \ldots 90$ to < 100
`bar(0:10:100,N,'r');`	you can plot these manually

ubd Universiti Brunei Darussalam

Random numbers

Many probabilistic processes rely on random numbers, and MATLAB contains the common distributions built in.

`>> rand`	draws from the uniform distribution from 0 to 1
`>> randn`	draws from the standard normal distribution (Gaussian)
`>> random`	can give random numbers from a wider range of distributions see `help random`

You can also seed the random number generators:

```
rand('state',0); rand(1);
rand(1);
rand('state',0); rand(1); % same random number
```

Changing mean & variance

We can alter the given distributions:

`>> y = rand(1,100)*10 + 5;` draws 100 uniformly distributed numbers between 5 and 15.

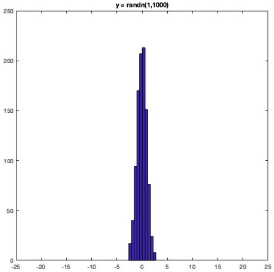
- `rand(1,100)` generates 100 random numbers uniformly distributed in $[0, 1)$
- `*10` scales them to $[0, 10)$
- `+5` shifts the interval to $[5, 15)$

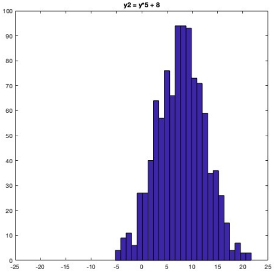
`>> y = floor(rand(1,100)*10 + 6);` gives 100 uniformly distributed *integers* between 6 and 15

- `floor` or `ceil` are more appropriate here than `round`
- you can also use `randi([6,15],1,100)`

Universiti Brunei Darussalam

Changing mean & variance






```
>> y=randn(1,1000);
```

generates 1,000 random values from a standard normal distribution, i.e. $y \sim \mathcal{N}(0,1)$

```
>> y2=y*5+8;
```

This scales the std to 5, and shifts all values upward by 8, i.e. $y_2 \sim \mathcal{N}(8,5^2)$

Example 1 (Simulating 1D Brownian Motion)

Goal: Simulate random motion of a particle in 1D and analyze its behaviour.

- Create a script called `brwn.m`
- Initialize a vector `pos` of length 10001 to store positions over time.
- Simulate particle movement, by looping from 2 to 10001.
- At each step, generate a random number:
    - If < 0.5, move left: `-1`.
    - If ≥ 0.5, move right: `+1`
    - Update the position based on the previous one
- Plot a 50-bin histogram of visited positions.

ubd Universiti Brunei Darussalam

Outline

Lecture 4: Advanced Methods

- Probability and Statistics
- Data Structures
- Images
- File I/O

Advanced data structures

Previously, we have used 2D matrices:

- Can have `n`-dimensions (e.g. RGB images)
- Every element must be the same type (e.g. integers, doubles, characters)
- Matrices are space-efficient and convenient for calculations
- Large matrices with many zeros can be made `sparse`

Sometimes, more advanced data structures are more appropriate:

- **Cell arrays** are similar to arrays, but their elements don't have to be of the same data type.
- **Structs** can bundle variable names and values into a single structure, similar to object-oriented programming in MATLAB.

Cells: organization

A **cell array** is like a matrix, but each element can store any type: numbers, strings, arrays, etc.

3×3 Matrix

1.2	-3	5.5
-2.4	15	-10
7.8	-1.1	4

3×3 Cell Array

'John'	32	{2,4}
'Mary'	27	1
'Leo'	18	[]

- **Why use cells?**
  - You can group heterogeneous data together without padding or multiple arrays.
  - Matrices can only store one type of data (e.g., only numbers).

Cells: initialization

- To initialize a cell, specify the size:
  `>> a = cell(3,10);` `a` is a cell with 3 rows and 10 columns

- or it can be done manually using curly braces `{}`
  `>> c = {'hello world', [1 5 4 3], rand(3,2)};`
  `c` is a cell with 1 row and 3 columns

- To access a cell element, use curly braces `{}`

```
>> a{1,1} = [1 3 4 -10];
>> a{2,1} = 'hello world 2';
>> a{1,2} = c{3};
```

Example 2 (Cells)

Goal: Generate simple random sentences using a cell array.

- Create a script called `sentGen`.
- Define a 2×3 `cell array`:
  - Row 1: three **names** (e.g., `'Alice'`, `'Bob'`, `'Carl'`)
  - Row 2: three **adjectives** (e.g., `'brilliant'`, `'kind'`, `'funny'`)
- Use `randi(3)` to pick:
  - a random name index (1 to 3)
  - a random adjective index (1 to 3)
- Display a sentence of the form: `'[name] is [adjective].'`
- Run the script multiple times to get different combinations.

Hint: To access cell contents using curly braces: `C{row, col}`.

Structs

- **Structs** allow you to name and bundle the relevant variables.
- To initialise an empty struct:

```
>> s = struct;
```

  - `size(s)` is `1x1`
  - Initialization is optional but recommended when using large structs

- To add fields:

```
>> s.name = 'Leo';
>> s.age = 18;
>> s.childAge = [];
```

  - Fields can be anything: matrix, cell and even struct
  - Useful for keeping variables together

- For more information, see `help struct`

Struct arrays

To initialize a struct array, provide field and value pairs:

```
>> ppl = struct('name', {'John', 'Mary', ...
    'Leo'}, 'age', {32, 27, 18}, ...
    'childAge', {[2;4], 1, []});
```

- `size(s)` is 1x3
- every cell must have the same size

`>> person=ppl(2);`	`person` is now a struct with fields name, age, children
	the values of the fields are the second index into each cell
`>> ppl(3) = s;`	adds struct `s` (field must match)
`>> person.name`	returns `'Mary'`
`>> ppl(1).age`	returns 32

ppl	`pp(1)`	`pp(2)`	`pp(3)`
name:	`'John'`	`'Mary'`	`'Leo'`
age:	32	27	18
childAge:	`[2; 4]`	`1`	`[]`

Struct: access

- To access `1x1` struct fields, give name of the field

  ```
  >> stu = s.name;
  >> a = s.age;
  ```

  `1x1` structs are useful when passing many variables to a function. Put them all in a struct, and pass the struct as an argument.

- To access `nx1` struct arrays, use indices

`>> person=ppl(2);`	`person` is now a struct with fields name, age, child's age
`>> pName=pp(2).name`	pName is `'Mary'`
`>> a=ppl.age`	`a` is a `1x3` vector of the ages; this may not always work, the vectors must be able to be concatenated

Example 3 (Sructs)

Goal: Transform a `cell array` into a `struct array`.

- Modify the `sentGen` script from the previous exercise.
- Create a `struct array` with two fields:
  - `name` - holds names from the first row of the cell array
  - `adj` - holds adjectives from the second row
- Use the existing cell array! Do **not** create a new array manually.
- Update the sentence display logic to use the struct, use output formatting:

```
>> fprintf('%s is %s.\n', name, adj);
```

- Run the script multiple times to describe each person.

Bonus: Include additional fields like `age` and `childAge` like in the previous slide.

Outline

Lecture 4: Advanced Methods

- Probability and Statistics
- Data Structures
- Images
- File I/O

Handles

- To initialize a cell, specify the size:

```
>> L = plot(1:10,rand(1,10));      gets the handle for the plotted line
>> A = gca;                        gets the handle for the current axis
>> F = gcf;                        gets the handle for the current figure
```

- To see the current property values, use `get`:

```
>> get(L);
>> yVals = get(L, 'Ydata');
```

- TO change the properties,

```
>> set(A, 'FontName', 'Arial', 'XScale', 'log');
>> set(L, 'LineWidth', 1.5, 'Marker', '*');
```

- Everything you see in a figure can be completely customized using handles.

ubd Universiti Brunei Darussalam

Reading/Writing Images

- Images can be imported as a matrix of pixel values

```
>> im = imread('myPic.jpg');
>> imshow(im);
```

- MATLAB supports almost all image formats such as `.jpeg`, `.tiff`, `.gif`, `.bmp`, `.png`
- To write an image, specify the RGB matrix (0 to 1 doubles, or 0 to 255 uint8)

```
img = rand(100,100,3);  % random RGB values between 0 and 1
imwrite(img, 'rand_img.png');
```

see `help imwrite` for more options.

Outline

Lecture 4: Advanced Methods

- Probability and Statistics
- Data Structures
- Images
- File I/O

Importing Data

- MATLAB provides robust support for reading and writing data.
- Use modern functions for better compatibility and performance.
- For structured tabular data, use `readtable`:

```
data = readtable('data.csv');
head(data)
```

  - Detects headers and delimiter automatically.
  - Supports CSV, TSV, TXT, XLSX, etc.

ubd Universiti Brunei Darussalam

Reading Excel files

Use `readtable`, `readmatrix`, or `readcell`:

```
T = readtable('file.xlsx');
M = readmatrix('file.xlsx');
C = readcell('file.xlsx');
```

- `readtable` keeps variable names.
- `readmatrix` returns numeric array (headers skipped).
- `readcell` returns everything (strings + numbers).

Writing Excel files

Use `writetable`, `writematrix`, or `writecell`:

```
writetable(T, 'output.xlsx');
writematrix(M, 'matrix.xlsx');
writecell(C, 'mixed.xlsx');
```

- These replace the deprecated `xlswrite`.
- Use sheet and range options as needed:

```
writetable(T, 'output.xlsx', 'Sheet', 'Sheet2', 'Range', 'B2');
```


ubd Universiti Brunei Darussalam

Reading any text file

- Use `fopen`, `fread`, `fgets`, `fscanf`, `textscan` for low-level control:

```
fid = fopen('raw.txt', 'r'); % Returns a handle to a file
lines = textscan(fid, '%s', 'Delimiter', '\n');
fclose(fid);
```

- `textscan` allows formatted reading, great for custom log files.