SM-2302: Software for Mathematicians

Lecture 2: Visualization & Programming

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

Semester I, 2025/26

Adapted from 6.057 IAP 2019 (`MIT OCW`)

User-defined Functions

- Functions look exactly like scripts, except they must start with a **function declaration**:

Must have the keyword: function

Function name should match m-file name

```
function  [x, y, z] = funcName ( in1, in2 )
```

if more than one output, use square brackets

inputs

- **No need for return:** MATLAB 'returns' the variables whose names match those in the function declaration.

- **Variable scope:** Any local variable created within the function but not returned disappears after the function stops running.

Overloading

- MATLAB functions are generally overloaded:
  - Can accept different numbers of input arguments
  - Can return different numbers of output arguments

- Example using `size()`:

`>> a = zeros(2, 4, 8)`	a 3D matrix (2×4×8)
`>> D = size(a)`	returns a vector: `[2 4 8]`
`>> [m, n] = size(a)`	returns first two dimensions: `2, 4`
`>> [x, y, z] = size(a)`	returns all three dimensions
`>> m2 = size(a, 2)`	returns size along 2nd dimension: `4`

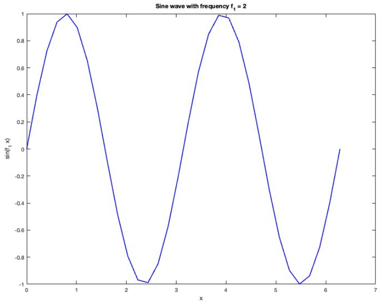
- You can overload your own functions:
  - Use special variables: `nargin`, `varargin` (for inputs)
  - And: `nargout`, `varargout` (for outputs)

Example 1 (Functions)

Goal: Write a function to plot a sine wave with a given frequency.

- Write a function with the following declaration: `>> function plotSin(f1)`
- Inside the function script, plot the sine wave $y = \sin(f_1 x)$ over the interval $x \in [0, 2\pi]$.
- Use 16 points per period for good sampling.

Expected output:



Relational operators

The operation of many branching constructs is controlled by **boolean values** (0 is false, 1 is true).

- **Rational operators**

`==`	equal to	`~=`	not equal
`>`	greater than	`<`	less than
`>=`	greater or equal	`<=`	less or equal

- **Logical operators**

`&`	and	`~`	logical not
`|`	or	`xor`	logical exclusive or
`all`	all true	`any`	any true

if/else/elseif

- `if/else/elseif` are basic flow control, which is common to all languages
- MATLAB syntax is slightly unique:

`if`

```
if cond
    commands
end
```

`else`

```
if cond
    commands1
else
    commands2
end
```

`elseif`

```
if cond
    commands1
elseif cond2
    commands2
else
    commands3
end
```

- Conditional statements evaluate boolean (true or false)
- No parentheses required - common blocks are between reserved words
- Too many `elseif`s? Consider using `switch`

for

- `for` loops are used for a known number of iterations:

```
for n = 1:100 % loop variable
    commands % command block
        ...
end
```

- Loop variable:
  - defined as a vector
  - becomes a scalar within the command block
  - doesn't need consecutive values
- Command block is anything between the `for` line and the `end`

while

- `while` is a more general for loop, where the number of iterations is not required:

```
while cond % conditional expression
    % some loop that is executed when cond is true
      ...
end
```

- Beware of infinite loops (use CTRL+C)
- Add `break` line to exit a loop

Example 2 (Conditionals)

Goal: Make your function flexible by checking the number of input arguments.

- Modify your existing function `plotSin(f1)` to accept two inputs:
  `function plotSin(f1,f2)`
- If the function is called with 1 input, plot the sine wave as before (using `f1` only). Otherwise, display the line:

  ```
  `Two inputs were given'
  ```

- Use the built-in variable `nargin` to check how many inputs were passed.

Plot options

- We can change the line colour, marker style and line style by adding a **string argument**:

```
>> plot(x, y,' k  .  - ');
```

color (black)

marker style

line style

- Or we can plot without connecting the dots, by omitting the line style argument:

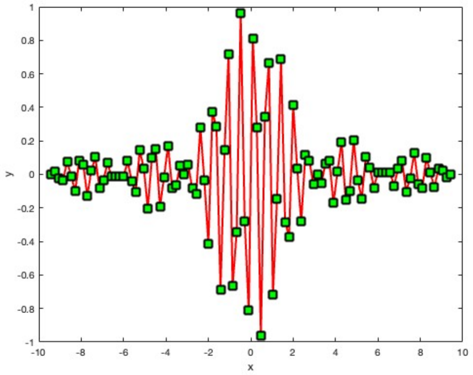
```
>> plot(x,y,'.')
```

- Refer to `help plot` for a full list of colours, markers and line styles

Line and marker options

Everything on a line can be customised:

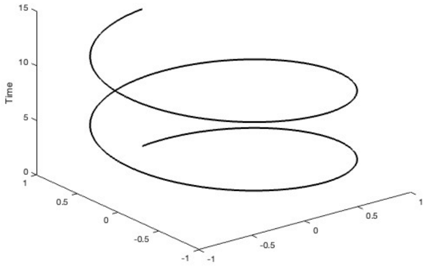
```
plot(x, y, 's-','LineWidth', 2, ...
'Color', [1 0 0], ... % RGB vector (red)
'MarkerEdgeColor', 'k', ... % black border
'MarkerFaceColor', 'g', ... % green fill
'MarkerSize',10)
```



3D line plots

We can plot in three dimensions just as easily as in two dimensions

```
time = 0:0.001:4*pi;
x = sin(time); y = cos(time); z = time;
plot3(x, y, z, 'k', 'LineWidth', 2);
zlabel('Time')
xlim([-1 1]); ylim([-1 1]); zlim([0 15]); % set limits on all 3 axes
```



Axis modes

Built-in axis modes (see `doc axis` for more modes)

`>> axis square`	makes the current axis look like a square box
`>> axis tight`	fits axes to data
`>> axis equal`	makes x and y scales the same
`>> axis xy`	puts the origin in the lower left corner (default for plots)
`>> axis ij`	puts the origin in the upper left corner (default for matrices/images)

Multiple plots in one figure

- Use `subplot` to have multiple axes in one figure:

`>> subplot(2,3,1)`	creates a figure with 2 rows and 3 columns of axes, and plots on the first axis each axis can have labels, a legend, and a title
`>> subplot(2,3,4:6)`	activates a range of axes and merges them into one.

- To close existing figures, use

`>> close([1 3])`	closes figures 1 and 3
`>> close all`	closes all figures (useful in scripts)

Saving figures

Figures can be saved in many formats. The common ones are

- **MATLAB figure (*.fig)** preserves all information
- **Bitmap file (*.bmp)** is an uncompressed image
- **EPS file (*.eps)** is a high-quality scaleable format
- **portable document format (*.pdf)** is a compressed image

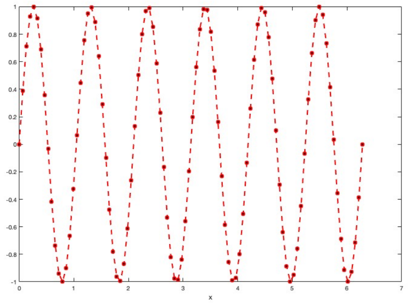
Example 3 (Advanced plotting)

Goal: Enhance your `plotSin` function with custom plotting features and subplot behaviour.

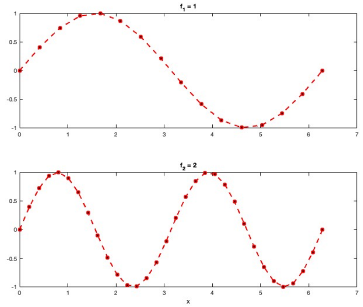
- Style the plot: using square markers connected by a dashed red line (`'r--s'`) with thickness of 2 as the line. Set the marker face colour to black, and use marker size 6.
- When called with 2 inputs: open a new figure and create a vertical layout with 2 plots. Plot each frequency in a separate subplot.
- MATLAB properties to use: `LineWidth`, `MarkerFaceColor`, `Marker`, `figure`, `subplot`

Expected outcome:

`plotSin(6)`



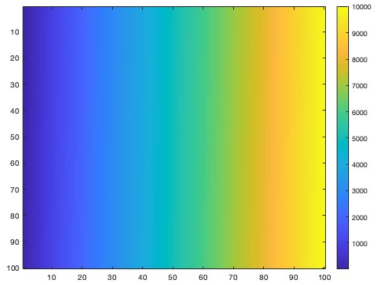
`plotSin(1,2)`



Visualising matrices

- Any matrix can be visualised as an image:

```
mat = reshape(1:10000, 100, 100);
imagesc(mat); % Scales values to span the entire colormap
colorbar
```

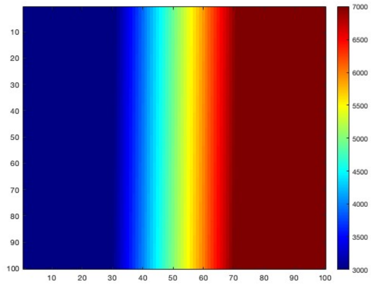


Universiti Brunei Darussalam

Visualising matrices

- Set limits for the color axis (analogous to `xlim, ylim`) and change the `colormap` (default is `parula`):

```
caxis([3000 7000])
colormap(jet) % other options are cool, gray, hot ...
```



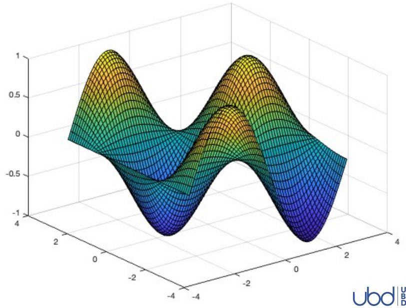
Surface plots

- It is more common to visualise surfaces in 3D. For example,

$$f(x, y) = \sin x \cos y, \quad x \in [-\pi, \pi],\ y \in [\pi, \pi]$$

- `surf` creates a surface by connecting vertices at points in space (x, y, z). See `help surf` for more options.
- The vertices can be denoted by matrices `X, Y, Z`, and created using `meshgrid`.

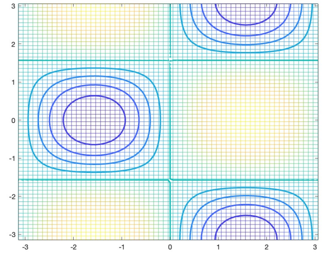
```
% make the x and y vectors
x = -pi:0.1:pi; y = -pi:0.1:pi;
[X,Y] = meshgrid(x,y); % create the matrices
Z = sin(X).*cos(Y); % function to evaluate
surf(X,Y,Z) % plot the surface
```



Contour

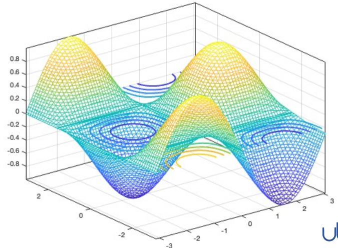
Make surfaces two-dimensional by using the `contour` command:

```
contour(X, Y, Z, 'LineWidth', 2)
hold on
mesh(X, Y, Z)
```



- `contour` takes the same arguments as `surf`
- the color indicates height
- linestyle and colormap properties can be modified

```
contour3(X, Y, Z, 50) % creates a 3D ...
    contour plot with contour levels 50
```



Example 4 (3D plots)

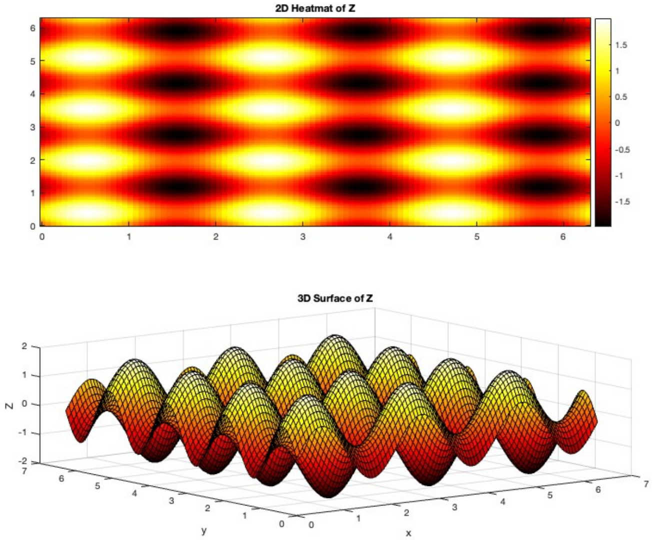
Goal: Extend `plotSin(f1, f2)` to create 2D and 3D visualizations of a combined sine function.

- If two inputs are passed, evaluate:

$$Z = \sin(f_1 X) + \sin(f_2 Y)$$

  using a 2D grid of values (create matrices `X` and `Y` using `meshgrid`).
- Top subplot: 2D heatmap of `Z`. Use `imagesc(x,y,Z)`, apply `colormap hot`, add `colorbar`. Then set axis to `xy`.
- Bottom subplot: 3D surface plot of `Z` (Use `surf`)

Expected outcome: `plotSin(3,4)` generates this figure



Specialised plotting functions

Functions	Example
`polar` makes polar plots	`>> theta = 0:0.01:2*pi;` `>> polar(theta, cos(theta*2));`
`bar` makes bar graphs	`>> bar(1:10, rand(1,10));`
`quiver` adds velocity vectors to a plot	`>> [X, Y] = meshgrid(1:10, 1:10);` `>> quiver(X, Y, rand(10), rand(10));`
`stairs` plots piecewise constant functions	`>> stairs(1:10, rand(1,10));`
`fill` draws and fills a polygon with specified vertices	`>> fill([0 1 0.5], [0 0 1], 'r');`

see `help` on these functions for syntax.

find

- `find` is a very important function that
  - returns indices of nonzero values
  - can simplify code and help avoid loops
- Basic syntax: `index = finc(cond)` for example,

```
x = rand(1,100);
inds = find(x<0.4 & x<0.6);
```

- Here, `inds` contain indices at which `x` has values between 0.4 and 0.6:

`x>0.4`	returns a vector with 1 for true, and 0 for false
`x<0.6`	returns a vector with 1 for true, and 0 for false
`&`	combines the two vectors using logical `and` operator
`find`	returns the indices of 1s

Avoiding loops

Consider the linear space: `x = sin(linspace(0, 10*pi, N));`
How many of the entries are positive?

Using a loop & conditional if/else:

```
count = 0;
for n = 1:length(x)
    if x(n)>0
        count = count + 1;
    end
end
```

Without loop:

```
count = length(find(x>0));
```

length(x)	loop time	find time
100	0.01	0
10,000	0.1	0
100,000	0.22	0
1,000,000	1.5	0.04

Avoid loops, and use built-in functions for more efficient codes.

Vectorization

- Another way to avoid loops is by using **vectorization**, which is more efficient for MATLAB
- Vectorized codes use indexing and matrix operations to avoid loops

For instance, to add every two consecutive terms:

```
% slow and complicated
a = rand(1,100);
b = zeros(1,100);
for n = 1:100
    if n==1
        b(n)=a(n);
    else
        b(n) = a(n-1)+a(n);
    end
end
```

```
% efficient and cleaner
a = rand(1,100);
b = [0 a(1:end-1)] + a;
```

Preallocation

- Avoid variables growing within a loop, as memory reallocation is a time-consuming process.
- Preallocate the required memory by initialising the array with its default value. For example,

```
a = zeros(1, 100);
for n = 1:100
    x = linspace(0, pi, 1000);
    y = sin(n*x).*exp(-x);
    res = trapz(x, y);  % Numerical integration
    a(n) = res;
end
```

- Variable a is only assigned values, no memory is allocated here.

Debugging

To use the debugger in MATLAB,

1. Set breakpoints – click the red dot next to a line number in the Editor
2. Run the script, then MATLAB pauses at the breakpoint
3. Step through the code using the debugging buttons:
    - `Step`: Execute current line and pause at the next
    - `Step In`: Dive into a called function
    - `Step Out`: Finish function and return to caller
    - `Continue`: Resume until next breakpoint
4. Use the command window or workspace to inspect variables
5. Stop debugging by clicking `Stop` button or type `dbquit` and `dbclear all`

Debugging measures

- When debugging functions, use the `disp` command to print messages instead of removing semicolons, which is easier. For example:
  ```
  >> disp(['loop iteration ' num2str(n)]);
  ```
- It can be helpful to determine the execution time of your code by using the `tic/toc` function. For example,

```
A = zeros(1000); A(1,3)=10; A(21,5)=pi;
B = sparse(A); % squeezes out any zero elements from full matrix A
C = rand(1000,1);

tic; A\C; toc;  % slow
tic; B\C; toc;  % much faster!
```