CME 192: Introduction to MATLAB
Lecture 2

Stanford University

January 17, 2019

Outline

Review

Scripts & Functions

Control Flow

Debugging

Review

Lecture 1

- Variables
- Operators
- Built-in functions
- Arrays: vectors and matrices
- Strings
- Cell Arrays
- Using documentation

Outline

Review

Scripts & Functions

Control Flow

Debugging

Scripts

script.m

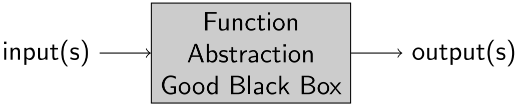
```
% most common sum of two die
% randi(<rand 1 to n>, <rows>, <cols>)
N = 10^5;
dice1 = randi(6, 1, N);
dice2 = randi(6, 1, N);

% result
mode(dice1 + dice2)
```

```
7
```

- a series of MATLAB commands
- equivalent to command line history
- good for quick prototyping
- `.m` extension
- no input arguments, no outputs
- use workspace variables (global)

What is a function?



- important concept
- allows for abstraction
- abstraction allows for complexity
- abstraction allows for code sharing

```
function [<out1>, <out2>, ...] = <name>(<arg1>, <arg2>, ...)
  <statement1>
  <statement2>
  <statement3>
   ...
end
```

Functions

`common_sum.m`

```
function s = common_sum(N)
  dice1 = randi(6, 1, N);
  dice2 = randi(6, 1, N);

  % suppressed
  s = mode(dice1 + dice2);
end
```

```
>> common_sum(10^5)
ans = 7
```

- takes inputs, produces outputs
- can be a program
- good for abstraction
- `.m` extension
- same name as the file
- doesn't create workspace variables (global)

Multiple Output Arguments Functions

`cplx2mag_ang.m`

```matlab
function [mag, ang] = cplx2mag_ang(c)
  mag = abs(c);

  % in computers always radians
  ang = angle(c);
end
```

```
>> [m, a] = cplx2mag_ang(4 + 2i)
m =  4.4721
a =  0.46365
>> % discard one output with ~
>> [~, a] = cplx2mag_ang(4 + 2i)
a =  0.46365
>> [m, ~] = cplx2mag_ang(4 + 2i)
m =  4.4721
```

- useful when computational effort can be combined
- typically better to use separate functions
- don't overuse, 4 output arguments is probably too many

Helper Functions

mag_ratio.m

```matlab
% magnitude ratio of two vectors
function r = mag_ratio(v1, v2)
  r = mag(v1) / mag(v2);
end

% magnitude of a 2D vector
function m = mag(v)
  m = sqrt(v(1)^2 + v(2)^2);
end
```

- place after the main function
- not available from command line
- easily reuse code
- form logical operations into functions
- improve readability

Anonymous Functions

```
% find for which x, f(x) = x * cos(x) = 4
f = @(x) x * cos(x) - 4;
% fzero(<fn>, <guess>), finds function zero
fzero(f, 3.0)
```

```
>> script
ans = 5.5224
```

```
% function generator
function fn = add_x(x)
  fn = @(z) z + x;
end
```

```
>> add2 = add_x(2);
>> add2(4)
ans = 6
```

Format

```
<fn_name> = @(<arg1>, <arg2>, ...) <output>
```

- necessary for functions operating on functions (ODE, zero finding, etc.)
- suited for simple applications
- useful for working in the command line

Commenting

```
% use % symbol for comments

%% for section headings

%{
 use %{ for blocks of comments
 close with
%}
```

- comment non-trivial code
- do not comment trivial code

Output Printing

```
x = 2;
x
disp(x);
% fprintf(<string>, <var1>, <var2>, ...)
% formats:
% integer - %d, real - %f, character - %c, string - %s
fprintf('x is %f \n', x);
fprintf('x is %d \n', x);
% \n is the newline character
fprintf('x is %d', x);
fprintf('x is %d', x);
```

```
x =
   2
2
x is 2.0000
x is 2
x is 2x is 2
```

Scripts & Functions

	Scripts	Functions
`.m` extensions	✓	✓
inputs, outputs		✓
workspace variables	✓	
same name as file	n/a	✓
can be a program	✓	✓
helper functions in same file		✓
preferred		✓

Outline

Review

Scripts & Functions

Control Flow

Debugging

Relational and Logical Operators

Relational operators

A == B	equality, element wise
A ~= B	inequality, element wise
A > B	greater than, element wise
A < B	less than, element wise
A >= B	greater equal than, element wise
A <= B	less equal than, element wise

Logical operators

A & B	and, both true, element wise
A | B	or, either or both true, element wise
~A	not, true if false, false if true, element wise
xor(A, B)	exclusive or, only true if only one true, element wise

Other

any(A)	is at least one element true
all(A)	are all elements true
stcmp(A, B)	compare two strings

Conditional `if` statements

`my_abs.m`

```
% absolute value
function x = my_abs(x)
  if x >= 0.0
    x = x;
  else
    x = -x;
  end
end
```

Format

```
if <condition>
  <statement(s)>
elseif <other condition>
  <statement(s)>
else
  <statement(s)>
end
```

Conditional `switch` statements

pos_neg_abs.m

```
% absolute value
function x = pos_neg_abs(x, pos_neg)
  switch pos_neg
    case 'pos'
      x = my_abs(x);
    case 'neg'
      x = -my_abs(x);
    otherwise
      error('Function accepts only ''neg'' or ''pos''');
  end
end
```

Format

```
switch <cond_var>
  case <condition1>
    <statement(s)>
  case <condition2>
    <statement(s)>
  otherwise
    <statement(s)>
end
```

```
>> pos_neg_abs(-17, 'pos')
ans = 17
>> pos_neg_abs(5, 'neg')
ans = -5
```

While Loop

play_dice.m

```
% play dice, lose on 1
function play_dice(x)
  last_throw = 6;
  while last_throw ~= 1
    last_throw = randi(6);
  end
end
```

Format

```
while <condition>
  <statement(s)>
end
```

For Loop

count_positive.m

```
% count positive elements in an array
function c = count_positive(v)
  c = 0;
  for i = 1:length(v)
    if v(i) >= 0.0
      c = c + 1;
    end
  end
  % or
  c = 0;
  for el = v
    if el >= 0.0
      c = c + 1;
    end
  end
end
```

Format

```
for <it> = <start>:<end>
  <statement(s)>
end
```

or

```
for <it> = <container>
  <statement(s)>
end
```

Nested For Loop

all_negative.m

```
% return matrix with all negative elements
function M = all_negative(M)
  for i = 1:size(M, 1)
    for j = 1:size(M, 2)
      M(i, j) = -abs(M(i, j));
    end
  end
end
```

break and continue

infinite.m

```
function infinite()
  r = 6;
  while 1 % finishes
    r = randi(10^3);
    if r == 1
      break
    end
  end

  while 1 % never finishes
    r = randi(10^3);
    if r == 1
      continue
    end
    if r == 1
      break
    end
  end
end
```

- break to break out of the loop
- continue to jump to the next iteration
- Ctrl+C interrupts execution

Outline

Review

Scripts & Functions

Control Flow

Debugging

Reading Erros

remember_el_wise.m

```
1 function remember_el_wise()
2   A = rand(4, 4);
3   v = ones(1, 4);
4
5   A2 = A^2;
6   res = A2 * v;
7
8   disp(res);
9 end
```

```
>> remember_el_wise
error: remember_el_wise: operator *:
nonconformant arguments (op1 is 4x4, op2 is 1x4)
error: called from
    remember_el_wise at line 6 column 7
```

- MATLAB executes line by line → pay attention to line number
- MATLAB has descriptive errors → try to read the error message

Log (print) execution

mand.m

```
% iterate Mandelbrot function
function z = mand(c)
  f = @(z) z^2 + c;
  z = 1e10;
  for i = 1:5
    z = f(z) % unsuppressed
  end
end
```

```
>> mand(4 + 4i)
z =  1.0000e+20 + 4.0000e+00i
z =  1.0000e+40 + 8.0000e+20i
z =  1.0000e+80 + 1.6000e+61i
z =  1.0000e+160 + 3.2000e+141i
z = Inf - NaNi
ans = Inf - NaNi
```

- good for quick debugging
- gives insight into execution
- practically all other debugging tactics are better

Breakpoints

- Universal programming concept
- Stops program execution, allows to examine state of memory (variables)
- Available in virtually any programming language
- MATLAB (and Octave) GUI programs have good support for them, click line number
- Program execution resumed manually

Reason Through Execution

- Read the code line by line
- See if you can spot easy errors
- Reconcile program behavior with text on the screen

Rubber Duck Debugging

- Useful when all other techniques have failed
- Sit down and explain your program to a rubber duck (or a friend)