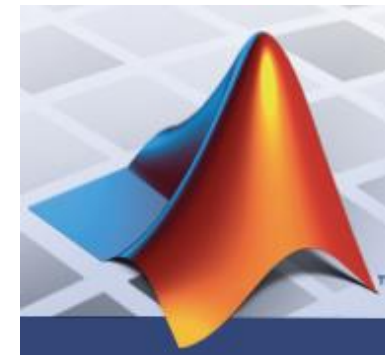
Introduction to Matlab workshop
November 2019

Structuring & debugging Matlab code



Danny Mitchell
(originally adapted from Rhodri Cusack)

Please download and unzip today's examples from:
http:/imaging.mrc-cbu.cam.ac.uk/methods/MatlabLecturesSchedule
and open them in the Matlab editor

Outline - goals

- Writing code that works (debugging)
- Writing clear code
- Writing fast code
- Writing code fast

In the following, red outlines highlight things that could be improved.
(Although this can be subjective!)

X

Why write clear code?

- Easier…
  - …to follow while writing and debugging
    - less likely to make mistakes
    - bugs are easier to spot
  - …to understand when you come back to it later
  - …to modify
  - …for *others* to understand and modify
    - Easier for them to help you
    - Less likely they will need your help

Making code more readable (example A)

```
1 -     a=3;
2 -     B=10;
3 -     c=[1:B];
4 -   for d=1:a
5 -     c=c(randperm(B));
6 -     fprintf('%d\n',d);
7 -   for e=1:B
8 -     fprintf('%d %d\n',e,c(e));
9 -     end;
10 -    end;
```

X

- What is this script for?
- Try running it
- How might it be improved?

- Structure your code
  - Leave spaces
  - Indent (Ctrl-I)
- Use meaningful variable names
- Add comments
- Be consistent
  - **`myVariable`**
  - **`MyVariable`**
  - **`myvariable`**
  - **`my_variable`**
  - **`MYVARIABLE`**
- Modularise (use sub functions)
- Cells to demarcate regions (%%)
- Ellipsis to continue line (**...**)

- Print helpful comments to the command line

Making code more readable (example C): Write this more clearly…

```
% Number of trials
num_trials=40;

% 4 conditions, 10 reps, change for randomized trial order
trialtype=[2    3    3    2    1    4    2    4    4    4    3    3    2    2    4    1   ...
    1    1    4    1    2    3    3    2    4    1    2    2    3    1    3    1    1    1   ...
    4    4    3    4    3    2];


fprintf('Experiment begins\n');
tic

% Main trial loop
for trialind=1:num_trials
    if (trialtype(trialind)==1)
        % Condition 1
        fprintf('Trial %d, this is an example of stimulus 1... AAAA\n',trialind);
        timerun(trialind)=toc;
        pause(0.1);
    elseif (trialtype(trialind)==2)
        % Condition 2
        fprintf('Trial %d, this is an example of stimulus 2... BBBB\n',trialind);
        timerun(trialind)=toc;
        pause(0.1);
    elseif (trialtype(trialind)==3)
        % Condition 3
        fprintf('Trial %d, this is an example of stimulus 3... CCCC\n',trialind);
        timerun(trialind)=toc;
        pause(0.1);
    elseif (trialtype(trialind)==4)
        % Condition 4
        fprintf('Trial %d, this is an example of stimulus 4... DDDD\n',trialind);
        timerun(trialind)=toc;
        pause(0.1);
    end;
end;
```

X

Repeated (especially non-independent) information is hard to change, and allows odd inconsistencies

Long manually entered lists are easy to get wrong, hard to check, and hard to change

Repeats of similar code are tedious to write, read and maintain. Easy way to introduce bugs!

Making code more readable (example D): Maybe you did better?

```
% These important parameters can be easily adjusted independently of
% everything else. They're at the top for convenience.
stimuli={'AAAA','BBBB','CCCC','DDDD'};
num_reps=10;

% Calculate how many conditions, trials
num_conds=length(stimuli);
num_trials=num_conds*num_reps;

% Reset random seed so that you get different trial orders every time (even
% when Matlab has just been started)
rand('state',sum(100*clock));

% num_reps trials of each condition, in randomized order
trialtype=kron(1:num_conds,ones(1,num_reps));
trialtype=trialtype(randperm(num_trials));

fprintf('Experiment begins\n');
tic

% Main trial loop
for trialind=1:num_trials
    fprintf('Trial %d, this is an example of stimulus %d... %s\n',trialind, ...
        trialtype(trialind),stimuli{trialtype(trialind)});
    timerun(trialind)=toc;
    pause(0.1);
end;
```

Occasionally syntax changes between versions:

```
rng('shuffle');
```

There are often many ways to skin a cat:

```
trialtype=repmat(1:num_conds,[1 num_reps]);
```

Structuring code into functions

- As scripts become larger, they can become cumbersome
  - Hard to find one section (though "code-folding" might help)
  - Hard to run just one part (although "cell execution" might help)
  - Hard to reuse one part – need to find it, remember what variables it needs, what variables are in the workspace, etc.
- Functions are usually better:
  - Encapsulate job, with well defined input and output interface
  - Flexible, e.g. rerun same code on different inputs
  - No need to worry about what might or might not be in the workspace
  - Easier for Mlint to spot more potential problems
  - Usually faster
- Another good chance to make things clear and tidy:
  - Choose descriptive function names
  - Structure them into directories
    - make sure they are on the path; check correct version is being used with:

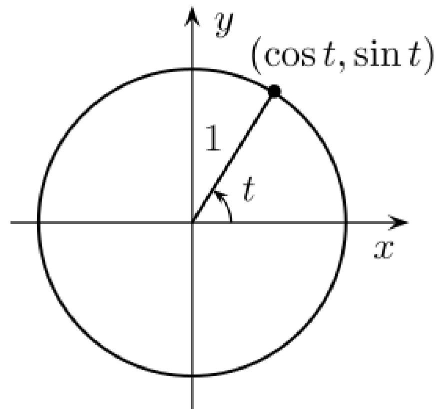
    ```
    which <functionname> -all
    ```
  - Document each function: What does it do? What are input and output variables? Who wrote it and when? Was it based on previous code? Keywords for `lookfor` in first comment line

Tiny examples (Example E)

1. Circumference of unit circle is

   $C = 2 \pi r$

   $= 6.28\ldots$



Will the example script always give the right answer?

2. The second part of the script prints a statement.

   Is it correct? What if it's run for a 2nd time? 3rd time...

Keeping the workspace tidy, and optimizing memory

- Good:
  - **Initialise variables prior to use** (largest first)
  - Use the appropriate data type (Consider sparse variables)
  - Group related variables into structures?

    (good for organisation; not for memory use; example below)
  - Prefer functions over scripts
- Less good:
  - `Clear` unwanted variables (but be careful!)
  - `Pack` (command-line only)
- Last resorts?
  - Launch without Java Virtual Machine? (matlab.exe –nojvm)
  - Save to file (slow).
    - If loading/saving via .mat file, use structure, and specify required variables.

E.g. `load handel` versus `wav=load('handel');`

or `wav=load('handel','y');`

Google

Google Search I'm Feeling Lu

Advertising Programmes Business Solutions About G

© 2011 - Privacy

Integrated development environments (IDEs)

Integrated design/debugging environments

- Typically consist of:
  - Source code editor
  - Compiler/interpreter
  - Debugger
- Makes coding much easier
  - The cycle of “tweak code - rerun – examine output” is a major factor in how long a program takes to write.
  - “Debuggers” let you run code line-by-line and examine what is happening
  - “M-Lint” will…
    - spot errors,
    - potential errors,
    - opportunities for optimization,
    - suggests improvements and fix things automatically!
    - Calculate McCabe complexity metric :
      - `checkcode filename.m -cyc` (…aim for < 10)
    - Shut up if you tell it to: `%#ok`, or set in preferences
  - useful to fix own code, or to work out what another program (e.g. SPM) does.

K>> Using the debugger

- Breakpoints
  - Adding breakpoints manually
  - **Triggering debug mode on an error** (`dbstop if error`)
  - conditional breakpoints (`dbstop in MFILE at LINENO if EXPRESSION`)
- Moving in the stack `(dbstack)`
  - `dbup, dbdown`
- Progressing through the code…
  - `dbcont` (F5)
  - `dbstep` (F10)
  - `dbstep in` (F11)
  - `dbstep out` (shift+F11)
  - `dbquit` (shift+F5)
- Catching an error without the debugger
  -
```
try……
      %… to run some code that you think might fail…
catch anError……
      %… in which case jump to here.
end
%… otherwise, continue from here…
```

Be careful!
Record the exception

K>> Practice using the debugger

- Example G. Try to get it to run to the end!
- Some tips:
  - Read the error message!
  - Are brackets of the right type and do they match up?
  - Are matrices of the right dimension? Should they be transposed?
  - Have variables been misspelled, or deleted when still needed?
  - Do functions have the correct inputs and outputs?...
  - ...use `help <functionnname>` if unsure what a function does
  - Are multiplication/divisions matrix-wise or element-wise?
  - etc...

Writing fast code: Vectorization & Pre-allocation (example I)

- Matlab has been designed for maths
- For some things (e.g., matrix maths) it is very fast:

```
>> clear z; tic; ind=1:50000; z=sin(ind); toc;
Elapsed time is 0.006758 seconds.
fx >>
```

- Some things, e.g. loops, can be slow:

```
>> clear z; tic; for ind=1:50000; z(ind)=sin(ind); end; toc
Elapsed time is 11.848116 seconds.
fx >>
```

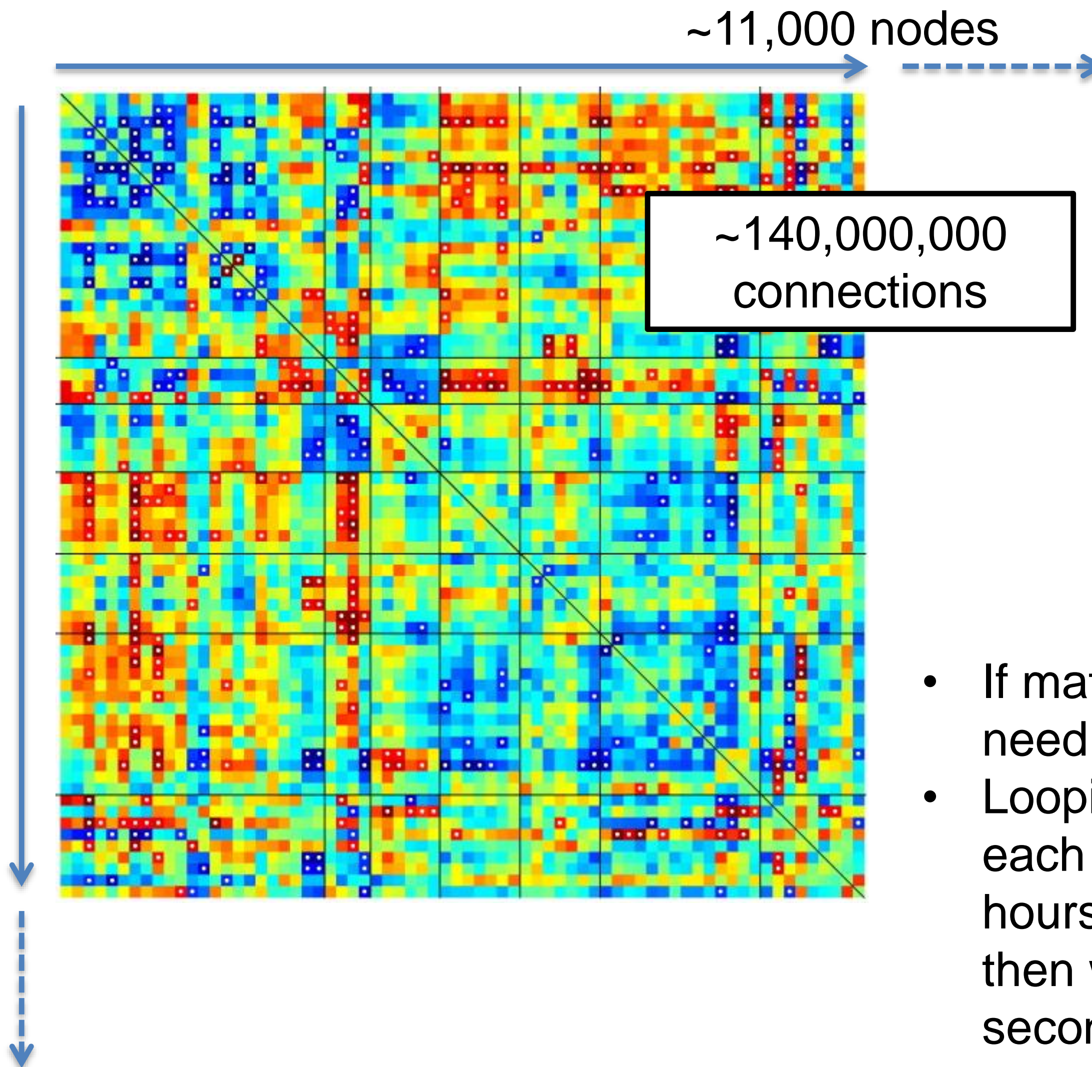
> - 1700 times slower!
> - Some of this can be recovered by pre-allocation – see demo

- BUT…
  - Sometimes loops are necessary, or more intuitive.
  - If you are going to use a slow loop, try to:
    - pre-allocate variables,
    - indicate progress (e.g. **`fprintf; drawnow`**, perhaps every n^{th} iteration)
    - Put slowest steps (e.g. file access) outside loop if possible
- Surprising: matrix orientation can also make a small difference! (generally better to operate down columns)

Vectorization: a real life case (Example J)

- Write out a connectivity matrix to 'Pajek' format:

```
*Vertices 6
1 "Node 1"
2 "Node 2"
3 "Node 3"
4 "Node 4"
5 "Node 5"
6 "Node 6"
*Arcs 8
1 2 2.0
2 3 2.0
3 1 2.0
1 4 1.0
4 5 2.0
5 6 2.0
6 4 2.0
4 1 1.0
```



- If matrix is symmetrical, only need to save half of it
- Looping through each cell to add each line to output file takes hours; preparing output variable then writing to file in one go takes seconds

Vectorization exercises (Example K)

```
% All of these would be quicker and more
%  elegant if vectorized
% Do not use them in this form

% Exercise - vectorize them!

% Get even numbers 2-20
for ind=1:10
    myevennumbers(ind)=ind*2;
end;

% Generate 10 random numbers between 1 & 20
for ind=1:10
    myrand(ind)=ceil(20*rand(1));
end;

% Find the square of the numbers from 1-10
for ind=1:10
    mysquares(ind)=ind^2;
end;

% Multiply each element in "a" by each
%  element in "b"
a=[1 2 3 4];
b=[1 2 1 3];
for ind=1:length(a)
    c(ind)=a(ind)*b(ind);
end;
```

```
% Given the two dimensional matrix x, subtract
%  the mean of each row
x=[1 2 3 4; 2 3 4 5; 7 6 5 4; -1 -2 -3 -4];
for row=1:size(x,1)
    rowtotal=0;
    for col=1:size(x,2)
        rowtotal=rowtotal+x(row,col);
    end;
    rowmean=rowtotal/size(x,2);
    for col=1:size(x,2)
        x(row,col)=x(row,col)-rowmean;
    end;
end;

% Find the index of the first occurence of "chicken"
x={'picasso','rembrandt','gaugin','chicken','monet'};
for ind=1:length(x)
    if (strcmp(x{ind},'chicken'))
        break;
    end;
end;
if (ind<=length(x))
    fprintf('The first chicken is at position %d\n',ind);
else
    fprintf('Not found.\n');
end;

% Find the first artist to get 5 points or more
clear x
x(1).name='picasso';
x(1).quality=3;
x(2).name='monet';
x(2).quality=3;
x(3).name='gaugin';
x(3).quality=5;
for ind=1:length(x)
    if (x(ind).quality>=5)
        fprintf('The artist is %s\n',x(ind).name);
    end;
end;
```

Writing code fast

- Optimization can become addictive – know when to stop!
- Matlab Profiler will tell you which parts of your code take the most time (don't need to optimise everything, just bottlenecks)
- A rare case… run-once jobs (less need to optimise?)
  - A. Spend 1 hour writing the code, and 10 hours running it
  - B. Spend 10 hours writing the code, and 1 hour running it
- Functions can be reused
- Debug efficiently
- Practise!

Topics that would need their own lecture:

Parallelization – is it worth it?

Various methods with pros & cons:

- Submitting directly to CBU cluster using qsub
- `parfor, spmd`
- GPU
- Initial overhead; limitations on syntax & memory
- Complex if dependencies
- Challenges of debugging

Version control – not specific to Matlab

- Prime example is **Git**
- A "repository" of the code keeps track of all changes
    - Any previous version may be accessed
- Can synchronise local copy with someone else's
- Particularly useful for large projects with many files, or collaborations
- If not using version control:
    - consider naming files with date, not e.g. FinalAnaysis_newer_version2b.m"
    - highlight changes with comments

Practice makes better.

Ever tried? Ever failed? No matter.
Try again. Fail again. Fail better.

Samuel Beckett, 1983

Good luck!