Defensive coding in MATLAB

Jeremy Badler, Ph.D.

9th January, 2020

This is not really about defensive coding

- “Defensive coding” (officially) = protecting your programs from incompetent or malicious user input
- BUT we still want to minimize bugs and write code others can understand (= standards and style)
- For stimulus generation and data analysis, there are additional considerations like measurement units and parameter logging
- MATLAB is more forgiving than standard programming languages, but this sometimes makes bugs harder to spot

What might badly-written code look like?

```
++) {         // Order of camel with index 0.
=0;l<5;l++) {         // Order of camel with index 1.
 (l != k) {
     for (int m=0;m<5;m++) {         // Order of camel with index 2.
             if (m != l) {
                     if (m != k) {
                             for (int n=0;n<5;n++) {         // Order of camel with index 3.
                                     if (n != m) {
                                             if (n != l) {
                                                     if (n != k) {
                                                             for (int o=0;o<5;o++) {         // Order of camel with index 4.
                                                                     if (o != n) {
                                                                             if (o != m) {
                                                                                     if (o != l) {
                                                                                             if (o != k) {
                                                                                                     for (int v=1;v<4;v++) {
                                                                                                             for (int w=1;w<4;w++) {
                                                                                                                     for (int x=1;x<4;
                                                                                                                             for (int
                                                                                                                                     f
```

General concepts

- Divide programs into logically-sound chunks, using functions if necessary
- Make variable and function names consistent and informative
- Try not to repeat code
- Save all parameters
- Timestamp outputs to avoid overwriting
- Document what the code DOESN'T say
- Conventions in this presentation:

  blue = code, green = comments, red = strings

Variables

- Make names informative and include units
  - timeMS, targetSizePix, frameRateHz, screenResolutionPixPerDeg
- Captalization conventions
  - variables start with lower case: fixationDurationSec
  - functions have the first letter capitalized: ComputeDistribution()
  - constants are in all caps: DEBUG_MODE = 1;
- Prefixes **n** for number and **i** for counter
  - **for** iTrial = 1:nTrial,
- CamelCase vs Underscore_method
  - Be consistent: I prefer CamelCase with underscores for constants
- Avoid overwriting MATLAB functions (length, which, log, …)
- Define constants only ONCE, at the top of the code

Comments

- Header comments are for users, code comments for programmers
- Comment liberally, but make them informative

```
x = x + 1;  % add one to x (this is not a useful comment)
x = x + 1;  % add one to compensate for MATLAB indexing (better)
```

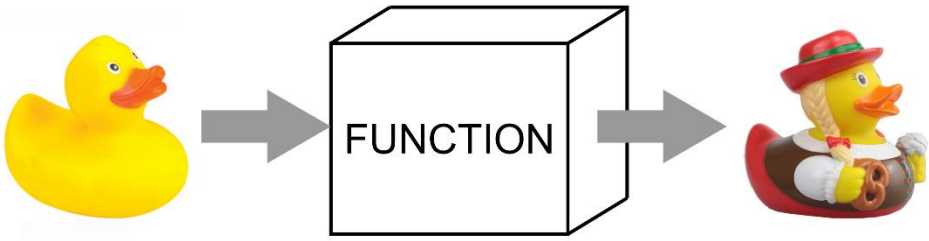
- Uses double comments to mark section boundaries

```
%% initialize user parameters
%% real-time display loop
```

- Comment loop ends

```
for iTrial = 1:nTrial,
    for iFrame = 1:nFrame,
    <3 pages of draw code>
    end % for iTrial = 1:nTrial,
end % for iFrame = 1:nFrame,
```

Comments and functions



- Documenting function example

```
function [dotXdeg, dotYdeg, colorList, nDots] = ...
        GenerateDotFieldSq(squareSideDeg, dotDiamDeg, …
                                dotDensityPerDeg2, blackProb, debugFlag)

% generate probabilistic dot field, with checks to eliminate overlaps

% NOTE: overlap avoidance will silently fail if too many iterations
needed (currently >1E6)

% INPUT squareSideDeg = size of dot field (etc.)

% OUTPUT dotXdeg = horizontal positions of all dots (etc.)
```

Functions and structures

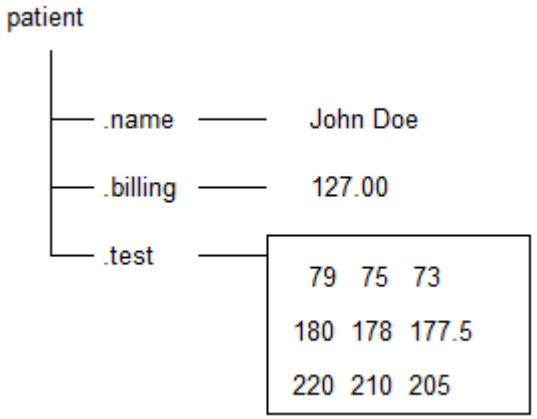
- Consider structures if you have many parameters

```
patient = []; % initialize as empty (not struct[]!)
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79, 75, 73; 180, 178, 177.5; 220, 201, 205];
```

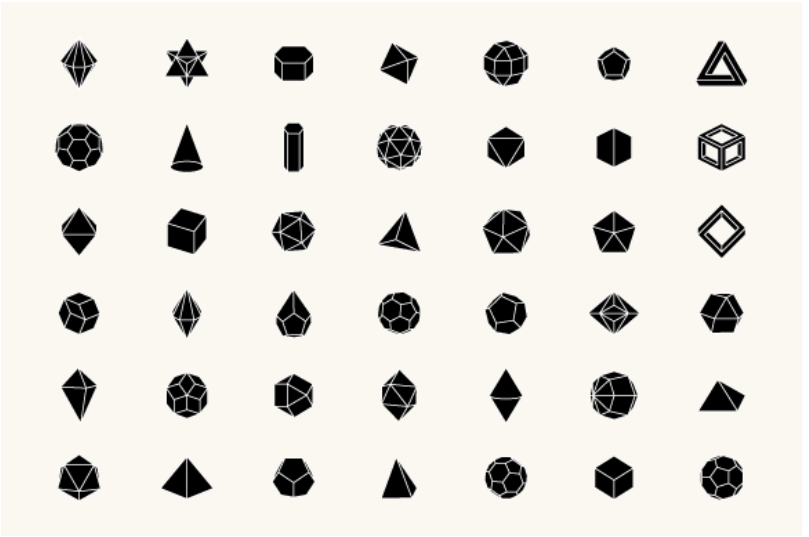
function updatedPatient = …

```
AddNewDataToPatient (newData, patient)
```

function drawSingleFrame(stimulusParameters)



Parameters



- Define parameters up top, in structure

```
params.dotDiamDeg = 0.26; % base dot size
params.dotDensityPerDeg2 = 2.0; % #dots per sq deg
<set up PTB screen window, etc>
params.derived.screenCtrPix = screenCtrPix; % substructure
params.derived.programName = mfilename; % returns name of m-file
<run single trial>
save (saveFileName, 'params', 'data', '-append'); % save after every trial
```

Randomizing

- Balanced, independent or joint?
  - Balanced assures all conditions are presented equally often, but can cause issues with predictability
  - Joint balanced assures parameter combinations are equally represented
  - True independent will create unequal group sizes that inconvenience statistical analysis
- Balanced independent:
  - Total trials nTrials multiple of nParamA * nParamB * nParamC *……
  - Use randperm() to index the matrix

```
randIdx = randperm(nTrials);   % shuffle
paramA_TrialIdxList = rem(randIdx-1, nParamA) + 1;  % vector of index values into ParamA
```

Randomizing continued

- Balanced joint:
  - Parameter A (n=2)
  - Parameter B (n=2)
  - Parameter C (n=3)

1	2	1	2	1	2	1	2	1	2	1	2
1	1	2	2	1	1	2	2	1	1	2	2
1	1	1	1	2	2	2	2	3	3	3	3

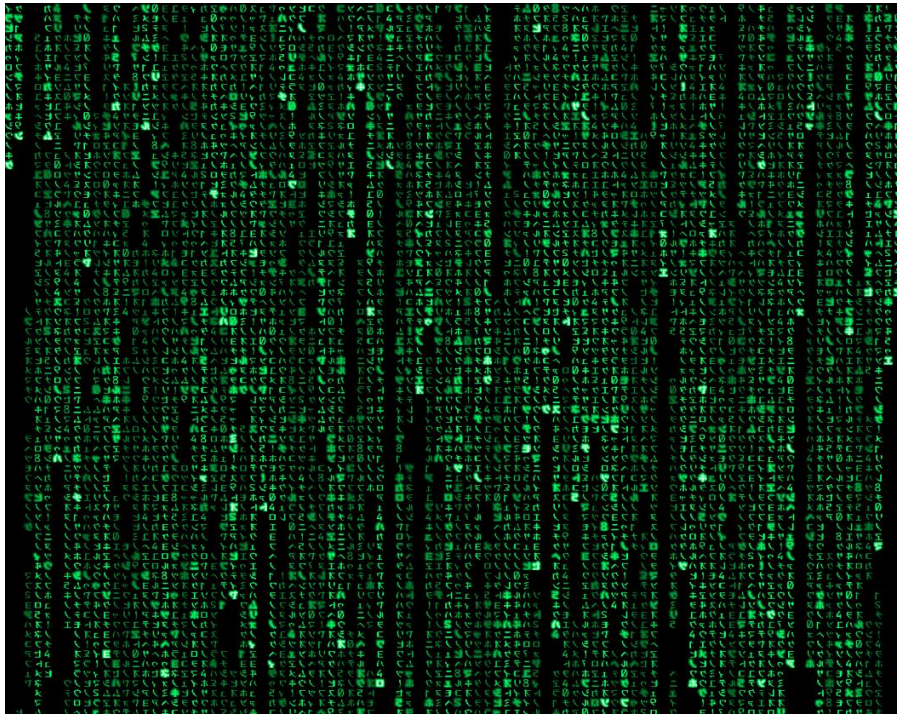
  - Use repmat() to create and randperm() to index the matrix
- True independent:
  - “sampling with replacement”

```
randIdx = randi(nParams, nTrials, 1);  % directly indexes your parameter
```

- Refresh the randomizer and save the seed!

```
randState = rng('shuffle');  % refresh random generator based on current time
```

Data logging



- Initialize a matrix of responses
  - E.g., response latency, response key
  - Should be in register with trial parameters!
  - For variable-length data:
    - Allocate extra space now and trim at end
- Write data at end of each trial (after all gfx)
  - Save every trial in case of crash
  - For large data:
    - You can sometimes just save the random seed (e.g., 800x800x3 noise mask texture)
    - Consider separate matrices:

```
eval(sprintf('trialDataMatrix%.3d = currentTrialDataMatrix;', iTrial));  % e.g. trialDataMatrix006
```

  - Convention: longest dimension as rows (trial number or sample time)

Optimization

- In the PTB draw loop, all operations need to be completed within one frame (~16 ms)
- Modern computers are fast and forgiving, but certain operations still take a lot of time:
  - Initializing new variables
  - First-time function calls
  - Concatenating arrays and matrices
  - Creating textures
  - Writing to disk
- Move as much as possible out of the draw loop!

Optimization example

```
randState = rng;
    % save current state of random number generator
xyPosPix = stepSizePix * cumsum(randn(nFrames, 2));
    % generate positions for 2-D random walk
noiseCoreTex = Screen('MakeTexture', winPtr, noiseCore);
    % create an offscreen texture using a previously created matrix
t0 = GetSecs; t1=t0; t2=t0; t3=t0;
    % initialize the timer variables we will use for consistency checks
Priority(MaxPriority(winPtr));
    % set maximum execution priority
```

Optimization example continued

```
%% now we are ready to start the display loop
vbl=Screen('Flip', winPtr); % do initial flip to synchronize
t0 = vbl; % save starting time
for iFrame = 1:nFrame,
    Screen('DrawDots', winPtr, xyPosPix(iFrame,:), dotSizePix);
    Screen('DrawingFinished', winPtr);
    <handle user input, etc. here if necessary>
    vbl=Screen('Flip', winPtr, vbl + (waitFrames-0.5)*ifi);
end % for iFrame = 1:nFrame,
t1 = GetSecs - t0; % elapsed time of all frames
```

Protecting from users

- Check inputs

```
if isempty(userInput), <execute some contingency>; end
if userInput < minAllowed || userInput > maxAllowed,
    <execute some contingency>; end
if numel(userInputString) > maxAllowed, userInputString =
    userInputString(1:maxAllowed); end
```

- Function argument counts

```
function MyFun(criticalInput1, criticalInput2, optionalInput3)
% remember to put your help/instruction text here!
if nargin < 3, input3 = <some default value>; end
if nargin < 2, help (mfilename); return; end % displays help & aborts execution
```

Protecting from everyone

- Sometimes code crashes
  - Because of users, programmers or computers having a bad day
- This is annoying in Psychtoolbox because of screen windows, etc.

```
try
        < do complicated psychtoolbox stuff>
catch
        Priority(0);            % ramp down the priority if it was elevated
        ShowCursor;             % restore the cursor if it was hidden
        ListenChar(0);          % stop character checking and reenable keyboard echos
        sca; fclose all;        % close any open windows, textures, files
        rethrow(lasterror); % display the error that caused the crash
end  % could also save data here, but you don't need to because you save every trial, right?
```

Miscellaneous tips

- Indent loops and conditionals!
- Avoid if possible break and continue in loops, as they make it difficult to check flow control
  - Consider using a while loop: while iTrials <= nTrials && ~abortCondition,
- Use parentheses for mathematical expressions
- Break long lines with ellipsis (…)
- Don’t be afraid to use spaces between operators for readability
- Use a leading zero if necessary when writing decimals
  - x = 0.5;
- saveFilename = sprintf('%s_%s_%s', mfilename, subjCode, datestr(now, 30)); % generate a safe file name

Acknowledgements

- Unakafova, V. A. (2017). Best practices for scientific computing and MATLAB programming style guidelines. 10.13140/RG.2.2.32109.18408.
- Wilson, G., et al. ”Best practices for scientific computing.” PLoS Biol 12.1 (2014): e1001745. (as cited in Unakafova)
- Johnson, R., ”Matlab programming style guidelines.” USA Datatool. Version 1 (2002) 2.1 updated version at http://www.datatool.com/downloads/MatlabStyle2%20book.pdf (as cited in Unakafova)
- Giovanni Fusco, Smith-Kettlewell Eye Research Institute, San Francisco

Good Luck!