General MATLAB

Bending MATLAB to your will

Beyond The Mouse
April 24, 2009
Celso Reyes

Outline

- Answer questions from handout
- Working from the command line
  - Structs
  - Cells
- Writing Scripts in the editor
- Creating and Juggling functions

from http://xkcd.com/251  April 22, 2009
Basic Data types are Double, char, and logical
ALL data are Arrays ( 1x1, 1xn, nxm, nmxmxp... )
Data Initialization
Accessing data: [ ], ( )
Indexing tricks: end, colon, and apostrophe
The MATLAB desktop environment

- Command window
- Workspace
- History
- Window
The MATLAB – nodesktop environment

Command prompt
MUCH Faster on slow machines
Best that most PC’s can hope for when SSH’ing into the SUN or LINUX networks.
All variables are created in the Workspace.

The history window keeps track of each line you’ve typed and can be used to repeat commands.

- recently used commands can be repeated through the use of up-arrows, and down-arrows

After the first few letters of a command have been written, the **TAB** key may be able to auto complete your line.

OKish for tinkering.
A struct is a special data type whose data is stored in fields that are accessible by name

- student.name = ‘joe’
- student.age = 25;

Equivalent to...

- student = struct(...

A cell is a container that can hold disparate types of data

- mycell(1) = {[1 5]}
- mycell(2,1) = {student}

Curly braces tell MATLAB to wrap this value inside a cell.
**Structs**

- structs may be nested
- all elements within an array of structs will have same fields.
- field names can be found with `fieldnames()` function.
- If values have same size, you can get all values from a field at once.

\[
\text{vols} = [\text{stereo}.\text{volume}]\]

*but only 1 level deep!*
items are put into cells by surrounding the item with curly braces. *e.g.*
mycell = \{item_1, item_2, \ldots, item_{max}\}

each cell element (a cell) can be retrieved with parenthesis. *e.g.*
mycell(index) = mycell_{index}

each cell value is accessed with curly braces. *e.g.*
mycell\{index\} = item_{index}

cells can provide multiple arguments to a function. *e.g.*
funkyfunction(mycell\{:\})
Cells vs String Arrays

**Character array**
- Each character is an element
- Each string must be the same length, but spaces can be used to pad them to the same length.
- Access each string via (row,:) 
- Access columns via (,:,col)

**Cell array**
- Each entire string is an element
- Each string can be any length
- Access string via {whichword}
The MATLAB Editor

- Debug Controls
- Cell Tools
- M-Lint Code Analyzer messages

Line 6: Terminate statement with semicolon to suppress output
Line 7: Parse error at ‘;’: usage might be invalid MATLAB syntax

Warnings found, click to go to the next message
No Warnings found
Variables used in scripts are created in the workspace. When the script finishes, these variables still exist.

When the script starts, variables may or may not already exist.

Sections of the script can be run independently.

- Each new section starts with `%%`
- Comments start with `%`
%% Script grabs mean of each GPS file in a dir

% directory containing preprocessed GPS files
files = dir('C:/data/2009/04');

%% Loop through each file, and get its mean
% we’re skipping the first two files ‘cause they are always ‘.’ and ‘.’
for n = 3 : numel(files)
    fileName = fullfile('C:/data/2009/04/', ... files(n).name); % one file per month
    load(fileName) % your variable is called “z”
    means(n-2) = mean(z);
end

N = 1;
figure out #files

N <= #files

No

Yes

Load Nth File (variable : Z)

means(N-2) = mean(Z)

N = N + 1

Done
%% Script grabs mean of each GPS file in a dir

% directory containing preprocessed GPS files
files = dir('C:/data/2009/04/');

%% Loop through each file, and get its mean
% we’re skipping the first two files ‘cause they are always ‘.’ and ‘.’
for n = 3 : numel(files)
    fileName = fullfile('C:/data/2009/04/', ... files(n).name); % one file per month
    load(fileName) % our variable is called “z”
    means(n-2) = mean(z);
end

Load Nth File (variable : Z)
N <= #files
means(N-2) = mean(Z)
Done

N = N + 1
No
Yes
N = 1;
figure out #files
14
%% Script grabs mean of each GPS file in a dir

% directory containing preprocessed GPS files
myDir = 'C:/data/2009/04';
files = dir(fullfile(myDir,'*.mat'));
means = [];
filenames = {};

%% Loop through each file, and get its mean
for n = 1 : numel(files) % one file per month
    fileName = fullfile(myDir,files(n).name);
gpsFileContents = load(fileName);
    means(n) = mean(gpsFileContents.z);
    filenames(n) = {fileName};
end
A function only knows about variables that are created within it, so there is no need to worry about pre-existing values.

The comments immediately below the function declaration are displayed when the user asks for HELP for a function.

The MATLAB command lookfor searches the first comment line.
This code has been moved from a script to a function.

Accepts the directory as an input
function [means dates] = get_gps_means(startday, endday)

%Figure out which files to grab, they're in directories like
%"C:/DATA/YYYY/MM" in files called gpsDD.mat

dates = fix(datenum(startday)) : fix (datenum(endday))
nDates = numel(dates);
[Y M D] = datevec(dates);
means = nan(1,nDates)

for n = 1 : nDates
    thisfile = {sprintf('C:/DATA/%04d/%02d/gps%02d.mat',...;
                  Y(n),M(n),D(n))}
    if (exist(thisfile, 'file'))
        tmp = load(thisfile);
        if any(strcmp(fieldnames(tmp), 'z'))
            means(n) = mean(tmp.z);
        else
            disp(['unable to load file ' thisfile]);
        end
    end
end
end

Now, any arbitrary range of dates can be processed.

Both multiple arguments and return values are present.
function [means dates] = get_gps_means(startday, endday)
%Figure out which files to grab, they're in directories like
"C:/DATA/YYYY/MM" in files called gpsDD.mat

dates = fix(datenum(startday)) : fix (datenum(endday))
nDates = numel(dates);
means = nan(1,nDates);

for n=1:nDates
    thisfile = getfilename(date(n));
    means(n) = process(thisfile);
end

function means = process(filename)
%Load a file, and return the mean of its Z's
if (exist(thisfile,'file'))
    tmp = load(thisfile);
    if any(strcmp(fieldnames(tmp),'z'))
        means = mean(tmp.z);
    else
        disp(['unable to load file ' thisfile]);
        means = nan;
    end
end

function fn = getfilename(thisdate)
%Figure out which files to grab based on date
[Y M D] = datevec(thisdate);

thisfile = {sprintf(...
'C:/DATA/%04d/%02d/gps%02d.mat',...
Y,M,D)}
subfunctions

- Subfunctions are all written in the same file as, and are written after the primary function.

- Subfunctions are only accessible to the functions contained within that one file.

```matlab
function outStuff = primary(inStuff)
    % The primary function is first function in the % M-file. This function can be invoked from % outside the M-file.
    outStuff = subfunction (inStuff);
    outStuff = otherSub(outStuff);

function myStuff = subfunction (myStuff)
    % visible only to all functions within this file.
    myStuff = myStuff .* 2;

function outStuff = otherSub(inStuff)
    % visible only to all functions within this file.
    outStuff = subfunction (inStuff);
    outStuff = outStuff + 1;
```
SCOPE of a variable is the section of code that has access to it.
- A variable’s scope is usually limited to the function in which it was created. In subfunctions, goes out of scope.

LIFE of a variable is the entire time it exists, from creation to deletion.
- A variable can be out of scope, but still exist.
Follow this program to determine scope and lifetime of each of the variables...

```plaintext
Workspace: >> n = 2;
>> weird(n);

function q = weird(n)
q = n + zing(n+1);

function s = zing(n)
s = n * 2;
```
**Arguments** are the inputs to a function.

- Enclosed in parenthesis
- comma separated

Number of input arguments can be determined by using **nargin**
Return Types are the values that a function passes back to the main program

- Multiple return types are enclosed in square brackets.
- A program can find out how many variables it was called with by using nargout
A mask is an array of logical values that can overlay another array, allowing you to work with specific values within that array.

\[ \text{mask} = (P \leq 0) \]

\[ P(\text{mask}) = [3; -5; -20; 0] \]
Indexing can be done with either an array of logicals (the same size as the array you’re trying to get information from) or an array of doubles.

- **Logical** – The index array is a mask that tells MATLAB which elements to keep or throw away.
- **Double** – each number represents the position within an array of the element of interest.

```matlab
>> primes = [1 3 5 7 9]
>> [isPrime, loc] = ismember(3,primes)
isPrime → true and loc → 2
>> [isPrime, loc] = ismember(primes,3)
isPrime → [F T F F F F]
loc → [0 1 0 0 0]
find(isPrime) → 2
```
Vectorizing your code can make it run much faster.

% log of numbers from .01 to 10
x = .01;
for k = 1:1001
  y(k) = log10(x);
  x = x + .01;
end

% log of numbers from .01 to 10
x = .01:.01:10
y = log10(x);

% append "new" to all files in direct
files = dir;
for n = 1: numel (files)
  newfiles(n) = ... 
  {strcat (files(n).name, '.new')}
end

% append "new" to all files in direct
files = dir;
newfiles = strcat({files.name},' .new')
Start with a clear vision of what goes in and what goes out.

List the broad steps required to solve the problem.

Each broad step is a perfect candidate for a function.

%% Deal cards Example
% 1. Find out how many players and how many cards each.
% 2. Create a deck
% 3. Shuffle deck
% 4. Deal to each player
% 5. Determine Score
Putting it together: Poker Skeleton

Use your outline to create skeletal functions that serve as placeholders for yet-to-be-created functions.

```matlab
function poker(nplayers, ncards)
    % 1. Find out how # players and # cards each.
    % 2. Create a deck
    % 3. Shuffle deck
    % 4. Deal to each player
    % 5. Determine Score

    function deck= create_deck()
    disp('creating a deck!')
    deck = [];

    function deck= shuffle_deck(deck)
    disp('shuffle shuffle')

    function show_cards(cardlist)
    disp('showing cards');

    function [cards, deck] = deal_cards(ncards, deck)
    disp('dealing')
    cards = [];

    function score= get_score()
    disp('Score!');
    score = 1;
```
starting game
creating standard deck
shuffle shuffle...
dealing 5 cards
Player 1:
Ace of Spades
Queen of Diamonds
7 of Diamonds
8 of Spades
9 of Spades
* High Card : 14

dealing 5 cards
Player 2:
10 of Clubs
Queen of Spades
5 of Diamonds
2 of Spades
10 of Hearts
* Pair!

Winner is player # : 2
fin