Matlab Basics

Paul Schrimpf

January 14, 2009

Overview

- Goals
  - Matlab features
  - Program design
  - Numerical methods

Topics to be covered

- Essentials of Matlab
- Using Matlab's features to design good programs
  - Example: dynamic programming
- Optimization and integration
  - Example: maximum likelihood
- Object-oriented programming
  - Example: automatic differentiation

Matlab References

- `help function` or more detailed, `doc function`
- Matlab Primer
- MATLAB on Athena
- 10.34 Matlab tutorial
- Numerical Computing with Matlab
- Art of Matlab

Operators

Matrix Operators

```
a+b;
a-b;
a*b;
a^n;
A';
A \ b; % returns x s.t. A*x=b
A / b; % returns x s.s. x*A=b
```

Array Operators

```
a.*b;
a.^n;
a.\b; % these are
a./b; % equivalent
a & b;  % don't confuse with &&
a | b;  % don't confuse with ||
~a;
arrayfun(fn,a); % evaluate fn at each element of a (usually no
```

Avoid Matrix Inversion

```
K = 2000;   N = K+1;   b = ones(K,1);
x = randn(N,K); y = x*b + randn(N,1);
xx = x'*x;
xy = x'*y;

% from slow to fast ...
tic; bhat1 = (xx)^(-1)*xy; toc;
tic; bhat2 = inv(xx)*xy; toc;
tic; bhat3 = xx \ xy;    toc;
```

- \ is also more accurate, see *purpose of inv*
- Example: `funWithInv.m`

Array Functions

Arrays of Constants

```
eye(10); % 10 by 10 identity
zeros(3); % 3 by 3 of zeros
zeros(2,3); % 2 by 3 of zeros
ones(31,35,69);
1:5;   % [1 2 3 4 5]
```

Vector Functions

```
sum(a,2); % sum along 2nd dimension of x
max(a); % max along 1st dimension of a
any(a,2); % a(:,1) | a(:,2) | ...
all(b);   % b(1,:) & b(2,:) & ...
cumprod(a); % cumulative product
```

More Functions

- All standard mathematical functions – linear algebra, special functions, polynomials, etc
- Manipulating arrays – sort, permute, find, set operations
- Strings – regexp, findstr, etc
- Use the Matlab Function Reference

Flow Control

```
if (j==3)
  % ... some commands ...
elseif (j>4)
  % ... some other commands ...
else
  % ... some other commands ...
end
```

```
for j=lo:hi
  x(j) = sqrt(j);
end
```

```
epsilon = 1;
while (1-epsilon ~= 1)
  epsilon = epsilon*0.99
end
```

Warning – Arrays and Flow Control

```
A = [1 2 3]; B = A; C = [1 2 2];
if A==B
  fprintf('A==B\n');
end

if A==C  % what message will be printed?
  fprintf('A==C\n');
elseif A~=C
  fprintf('A~=C\n');
else
  fprintf('~(A==C) && ~(A~=C) !?\n');
end
```

Output

Array Subscripting

```
A = magic(4); % 4 by 4 magic matrix
A(2,3); % by subscript
A(5); % by linear index -- A(5) = A(1,2)
ind2sub(size(A),5); % convert linear index to subscripts
bigA = A>10; % logical 4 by 4 matrix
A(bigA); % vector of elements of A > 4, in order of linear ind
```

Array Subscripting

```
A = eye(2);
B = rand(3,2,2);
A(1,:)  % [1  0]
A(:,2)  % [0; 1]
try
  B(1,:,:)+A; % not allowed
catch
  squeeze(B(1,:,:))+A;
end
B(1); % = B(1,1,1)
A(3); % = A(1,2) – matrices stored columnwise
B(2,A==1); % [B(2,1,1) B(2,2,2)]
```

Structures

- Way of organizing related data
- Create a structure, `s`, with fields, `x`, `y`, and `name`

```
s.y = 1;
s.x = [1 1];
s.name = 'foo';
% or equivalenty
s2 = struct('y',1,'x',[1 1],'name','foo');
```

- Use the fields like normal variables
- Can create arrays of structures

```
for i=10:(-1):1
  s(i).y = rand();
  s(i).x = [i:i+2];
  s(i).name = sprintf('name%d',i);
end
```

Structures

- Structure array → normal array

```
% slow, explicit way
for i=1:length(s)
  X(:,i) = s(i).x;
end
% equivalent fast way
X = [s.x]; % rationale: s.x is a comma separated list
```

- Test for equality

```
isequal(s1,s2); % works for any s1, s2
```

Structures

- Get a list of fields

```
f = fieldnames(s); % creates cell array containing names
                   % of s
```

- Dynamic field reference:

```
s.x      % a static reference to s.x
s.('x')  % dynamic reference to s.x
```

Structures

- Loop over fields

```
f = fields(s); % fields() equivalent to fieldnames()
for i=1:length(f)
  doSomething(s.(f{i})); % do something to each field
end
% equivalently,
for f=fields(s)' % for can loop over any array
  doSomething(s.(char(f)));
end
% most compact
structfun(@doSomething,s);
```

Cell Arrays

- Cell arrays are can have entries of arbitrary datatype

```
a = cell(3,2); % create 3 by 2 cell array
a{1,1} = 1;
a{3,1} = 'hello';
a{2,2} = randn(100,100);
```

- Useful for strings and avoiding squeeze()
- Using cell arrays with other datatypes can be tricky
  - indexing with () gives elements of cell arrays, which are themselves cells
  - indexing with {} converts elements of cell arrays to their underlying type, returns comma separated list if not singleton

```
a = {[1 2], 3}; % create 2 by 1 cell array
y = a{1}; % y is 1 by 2 numeric array
ycell =a(1); % is 1 by 1 cell array
x = y+1; % allowed
xcell = ycell+1; % not allowed
onetwothree = [a{1:2}]; % = [1 2 3]
```

Commenting

- Comments are anything after a `%` or a `...`
- Special comments:
  - First contiguous block of comments in an m-file are that file's help
    - `% See also FUNCTION` creates clickable link to help for function.m
    - Always include: a description of what the function does, what inputs are expected, and what kind of output will be produced
  - Code "cells" are delimited by `%% Cell title`
    - Matlab editor has special abilities for working with cells
    - `publish('file.m')` runs file.m and makes nice output

```
% publish all m-files in currect directory
files = dir('*.m');
cellfun(@(x) publish(x,struct('evalCode',false)), ...
        {files.name},'UniformOutput',false);
```

Debugging

- Nobody writes a program correctly the first time
- A debugger lets you pause your program at an arbitrary point and examine its state
- Debugging lingo:
  - breakpoint = a place where the debugger stops
  - stack = sequence of functions that lead to the current point; up the stack = to caller; down to the stack = to callee
  - step = execute one line of code; step in = execute next line of code, move down the stack if a new frame is added; step out = execute until current frame exits
  - continue = execute until the next breakpoint

Matlab Debugging

- Buttons at top of editor – set/clear break points, step, continue
- More under Debug menu or from the command line:
  - Set breakpoints

```
dbstop in mfile at 33 % set break point at line 33 of
dbstop in mfile at func % stop in func() in mfile
dbstop if error % enter debugger if error encountered
dbstop if warning
dbstop if naninf
```

  - `dbstack` prints the stack
  - `dbup` and `dbdown` move up and down the stack
  - `mlint file` analyzes file.m for potential errors and inefficiencies, messages also shown on right edge of editor

```
for i=1:10
  x(i) = i;  %#ok (tells mlint to ignore this line)
end
```

Profiling

- Display how much time each part of a program takes
- Use to identify bottlenecks
  - Try to eliminate them
- Could also be useful for debugging – shows exactly what lines were executed and how often

Matlab Profiler

- `profile on` makes the profiler start collecting information
- `profile viewer` shows the results
- Very nice and easy to use

Example: Diffs in Diffs Simulation

- From 382: recreate and extend simulations from Betrand, Duflo, and Mullanaithan (2004)
- Illustrates:
  - Importing data
  - Lots of subscripting
  - Use of structures
  - Random numbers
  - Comments and publishing

`Code`

Exercises

1. Take a simple program that you have written in another language and rewrite it in Matlab.
2. Taken from the art of Matlab blog: "Q: Suppose there is a multiple-choice quiz, and for each question, one of the responses scores 0 points, one scores 3 points, one scores 5 points, one scores 8 points, and one scores 10 points. If the quiz has 4 questions, and assuming that each taker answers all of the questions, then which totals per taker are not possible? For example, it would not be possible to finish the quiz with a total score of 2. If the quiz had 7 questions? Can you generalize the code so that the number of questions can be varied by varying a single assignment?"
3. Write a collection of Matlab functions for linear regression. You could include OLS, GLS, SUR, IV, 3SLS, etc.