Matlab – Miscellaneous Topics

Paul Schrimpf

May 31, 2007

This lecture will cover three unrelated topics:

1. Using the symbolic math toolbox
   - Help compute derivatives for minimizations
   - Automatic log-linearization and higher order approximations
2. Debugging and profiling
3. Automating the creation of quality output

Symbolic Math Toolbox

- Symbolicaly compute derivatives, integrals, solutions to equations, etc.
- Perform variable precision arithmetic
- Uses computational kernel of Maple

Perturbation Methods

- A model that relates an endogenous variable, x, to some parameters, ϵ

$$f(x(\epsilon), \epsilon) = 0$$

  want to solve for $x(\epsilon)$

- Suppose $x(0)$ is known
- Approximate $x(\epsilon)$ using a Taylor series and the implicit function theorem
  - We know: $\frac{d^n}{d\epsilon^n}\left(f(x(\epsilon), \epsilon)\right) = 0$ for all n
  - Use this to solve for $x^n(0)$, e.g.

$$\begin{aligned} 0 =& f_x(x(\epsilon), \epsilon)x'(\epsilon) + f_\epsilon(x(\epsilon), \epsilon) \\ x'(0) =& - f_\epsilon(x(0), 0) f_x(x(0), 0)^{-1} \end{aligned}$$

  - This becomes tedious, messy for high n $\longrightarrow$ automate it with the symbolic math toolbox

Log-Linearization

- Log-linearization is just a first order perturbation method
- We will generate an arbitrary order approximation to the neoclassical growth model
- Based on Schmitt-Grohé and Uribe (2004)
  - They compute a second order approximation, we generalize their approach

Model

- Model: CRRA, no leisure, Cobb-Douglas production

$$0 = E_t \left[\begin{array}{c} c_t^{-\gamma} - \beta c_{t+1}^{-\gamma} \alpha e^{a_{t+1}} k_{t+1}^{\alpha-1} + (1-\delta) \\ c_t + k_{t+1} - e^{a_t} k_t^{\alpha} - (1-\delta) k_t \\ a_{t+1} - \rho a_t \end{array} \right]$$

- We want the policy functions:

$$c_t = g(k_t, a_t, \sigma)$$
$$\left[\begin{array}{c} k_{t+1} \\ a_{t+1} \end{array} \right] = h(k_t, a_t, \sigma) + \left[\begin{array}{c} 0 \\ \sigma \epsilon_{t+1} \end{array} \right]$$

- Expand around non-stochastic steady-state, $(c, k, a, \sigma) = (\bar{c}, \bar{k}, \bar{a}, 0)$

perturb.m (1)

```
% compute n-order approximation for neoclassical growth model
% based on http://www.econ.duke.edu/¬uribe/2nd_order/neoclassi
clear;
%Declare parameters as symbols
syms SIG DELTA ALFA BETA RHO;
% equivalent command would be: SIG = sym('SIG'); ...

%Declare symbolic variables
syms c cp k kp a ap;
%Write equations that define the equilibrium
f = [c + kp - (1-DELTA) * k - a * k^ALFA; ...
  c^(-SIG) - BETA * cp^(-SIG) * (ap * ALFA * kp^(ALFA-1) + 1 -
  log(ap) - RHO * log(a)];
% redefine in terms of controls, y and states, x
x = [k a]; y = c; xp = [kp ap]; yp = cp;
% Make f a function of the logarithm of the state and control v
f = subs(f, [x,y,xp,yp], exp([x,y,xp,yp]));
```

perturb.m (2)

```
% define variables for steady state values
syms as cs ks;
xs = [ks as];
ys = cs;

% set parameter values
BETA=0.95; %discount rate
DELTA=1; %depreciation rate
ALFA=0.3; %capital share
RHO=0; %persistence of technology shock
SIG=2; %intertemporal elasticity of substitution
```

perturb.m (3)

```
% we need a lot of symbolic variables to be the Taylor
% series coeficients of g() and h(), we put these in arrays
% G and H
nX = length(x);
nY = length(y);
% initialize G and H
n = 2;
G = symArray([nY (n+1)*ones(1,nX+1)],'g');
H = symArray([nX (n+1)*ones(1,nX+1)],'h');
H(1:length(xs)) = xs; % steady state x=g(0)
G(1:length(ys)) = ys; % steady state y=h(0)
```

symArray.m

```
function A = symArray(d,prefix);
 % returns a symbolic array of size d
 % the symbolic variable names used are prefix%d
 % make sure you don't use these elsewhere
  v = 0;
  n = prod(d);
  sub = cell(length(d),1);
  for i=1:n
    [sub{:}] = ind2sub(d,i); % returns n-tuple subscript corres
                             % linear index i
    ind = sprintf(',%d',cell2mat(sub));
    ind = ind(2:length(ind));
    eval(sprintf('A(%s) = sym(''%s%d'');',ind,prefix,v));
    v=v+1;
  end
end % function symArray()
```

perturb.m (5)

```
% construct g, h, and g(h)
syms g h gh s e;
[g args cg]= multiTaylor(G,n);
% make g function of deviation from expansion point
g = subs(g,args,[x-xs,s]);

[h args ch]= multiTaylor(H,n);
h = transpose(h)+[0,s*e];
gh = subs(g,x,h);
gh = subs(gh,args,[x-xs,s]);
h = subs(h,args,[x-xs,s]);
T = [x,g,h,gh];  % T(x,s) = x, y, xp, yp
```

multiTaylor.m (1)

```
function [f x c] = multiTaylor(F,n)
  % given derivative matrix F, construct symbolic taylor series
  % that takes symbolic arguments 'x'
  % c is a vector of all symbolic coefficients used

  nOut = size(F,1); % dimension of f()
  nIn = ndims(F)-1; % dimension of args
                   % so, f: R^nIn -> R^nOut
  n=n+1;
  % construct arguments
  for i=1:nIn
     x(i) = sym(sprintf('x%d',i));
  end
```

multiTaylor.m (2)

- $f : \Re^k \rightarrow \Re^m$, write Taylor expansion as

$$f(x+h) \approx \sum_{|\alpha| \leq n} \frac{D^\alpha f(x)}{\alpha!} h^\alpha$$

where α is an k-tuple of integers, $|\alpha| = \sum |\alpha_i|$, $\alpha! = \prod \alpha_i!$, $h^\alpha =$`h.^alpha`

```
aold = zeros(nIn^(n-2),nIn);
a = ones(nIn^(n-1),nIn);
% a will be all n-tuples of positive integers such that sum(

% initialize f to zeros order expansion
ind = sprintf(',%d',a(1,:));
ind = ind(2:length(ind));
eval(sprintf('f = F(:,%s);',ind));
eval(sprintf('c = F(:,%s);',ind));
%f = F(:,a(1,:));
```

multiTaylor.m (4)

```
  % build taylor series
  for d=2:n
    aold(1:nIn^(d-2),:) = a(1:nIn^(d-2),:);
    j=1;
    for o=1:nIn^(d-2)
        for i=1:nIn
            a(j,:) = aold(o,:);
            a(j,i) = aold(o,i)+1;
            j = j+1;
        end
    end
    assert(j==nIn^(d-1)+1);
    for j=1:nIn^(d-1)
      ind = sprintf(',%d',a(j,:));
      ind = ind(2:length(ind));
      eval(sprintf('f = f+F(:,%s)*prod(x.^(a(j,:)-1)./(factoria
      eval(sprintf('c = [c; F(:,%s)];',ind));
    end
  end
end % function multiTaylor()
```

perturb.m (6)

```
% now we want to compose f(T(x,s)), differentiate n times, set
% resulting equations to zero and solve for unkown Taylor seri
% coefficients
FT = subs(subs(f),[x,y,xp,yp],T);
eqn = [];
dFT = FT;
for d = 1:n
  dFT = jacobian(dFT,[x,s]);
  eqn = [eqn; reshape(dFT,prod(size(dFT)),1)];
end
for i=1:length(eqn)
  % could do all at once, but this command is slow because
  % eqn has very complicated expressions
  fprintf('working on eqn(%d) ... ',i);
  eqn(i) = subs(eqn(i),[x,s,e],[xs,0,0]);
  fprintf('finished\n');
end
```

perturb.m (7)

```
% solve for steady state
fs = subs(f,[x,y,xp,yp],[x,y,x,y]);
[as cs ks] = solve(fs(1),fs(2),fs(3),a,c,k);
as = 0;
cs = sym('log(exp(as)*exp(ks*ALFA) - DELTA*exp(ks))');
cs = subs(cs);
xs = subs([ks as]);
ys = subs(cs);
```

perturb.m (8)

```
% now need to ask to solve eqn for the unkown coefficients
% there doesn't seem to be an elegant way, so use eval ...
cmd = '';
for i=1:numel(eqn)
   cmd = sprintf('%s,subs(''q=0'',q,eqn(%d))',cmd,i);
end
coeffs = [cg; ch;];% cgh];
unknown = [];
for i=1:numel(coeffs)
    try
        % this will throw an error if coeffs(i) is unknown
        subs(coeffs(i));
    catch
        % add unknown coeff to list of things we're solving fo
        cmd= sprintf('%s,coeffs(%d)',cmd,i);
        unknown = [unknown; coeffs(i)];
    end
end
cmd = ['soln=solve(' cmd(2:length(cmd)) ');'];
% solve will take a very long time with exact eqn
```

perturb.m (8)

```
% print the solution(s)
for i=1:length(f)
    try
      fprintf('%s = %s\n',f{i},char(vpa(soln.(f{i}),4)));
    end
end

% could do more, like choose the stable solution,
% check for range of validy of the solution, maybe
% create some graphs, etc
```

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

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

Creating Output

- Just like your program should be easy to modify, your final output should be easy to modify
- Good goal: a single command runs your program, creates tables and graphs, and inserts them into your paper
- I do it with LaTeX
- Could probably also use Excel

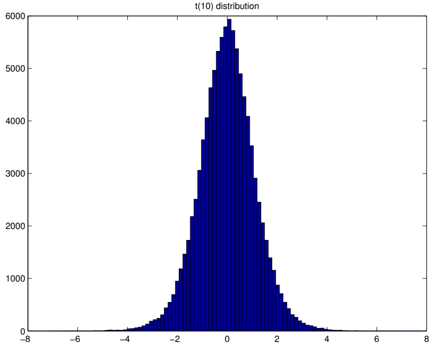

Table: A Random Matrix

	col 1	col 2	col 3	col 4	col 5
row 1	0.236	0.454	0.0552	0.581	0.296
row 2	0.881	0.162	0.204	0.676	0.702

LaTeX Code

```
\centering{\includegraphics[height=0.5\pageheight]{figs/randhi

\centering{
 \begin{table} \caption{A Random Matrix}
   \input{tables/rand.tex}
 \end{table}
}
```

Matlab Code

```
clear;
M = 2;
N = 5;
% create a table
x = rand(M,N);
out = fopen('tables/rand.tex','w');
fprintf(out,'\\begin{tabular}{');
for c=1:size(x,2);
  fprintf(out,'c');
end
fprintf(out,'}\n');
% print column headings
for c=1:size(x,2);
  fprintf(out,' & col %d',c);
end
fprintf(out,' \\\\ \\hline \n');
```

Matlab Code

```
% print the rows
for r=1:size(x,1);
  fprintf(out,'row %d',r);
  for c=1:size(x,2)
    fprintf(out,'& %.3g',x(r,c));
  end
  fprintf(out,' \\\\ \n');
end
fprintf(out,'\\hline\\end{tabular}');
fclose(out);

% create a histogram
figure;
hist(random('t',10,100000,1),100);
title('t(10) distribution');
print -depsc2 figs/randhist.eps;
```

Exercises

1. The perturbation code is not nearly as general as it could be. Make it so that it can solve any model of the form $f(x(\epsilon), \epsilon) = 0$. In particular, your code should be able to solve the income fluctuation problem from lecture 1. Compare the solution to the one obtained in lecture 1.
2. (hard) In the previous lecture we saw that derivatives can really help for optimization. Pick an often optimized class of functions and write a program using the symbolic toolbox that automatically computes derivatives.
3. Pick any program and profile it. Try to use the results to improve the performance of the program.
4. (boring) Make one of the programs we've covered produce nicer output.