Matlab – Designing Programs

Paul Schrimpf

January 14, 2009
What makes a good program?

1. Correct
   ▶ Test everything

2. Maintainable
   ▶ Expect unexpected changes

3. Efficient
   ▶ But not at cost of 1 and 2
General Principles

- Plan before coding
- Break problem into small, independent tasks performed by separate functions
  - Write and test each function individually
  - Document how functions interact – inputs and outputs
  - Make functions general, avoid unnecessary assumptions
- Choose data types carefully
- Write nicely
  - Descriptive names
  - Comments
Efficiency

- Avoid loops
- Preallocate arrays
- Use built-in functions
- Be aware of memory layout
Simple income fluctuation problem

Based on a problem set from 453

$$v(x, y) = \max_{c, x'} u(c) + \beta E \left[ v(x', y') \mid y \right]$$

s.t.

$$x' = R(x - c) + y'$$

$$x' \geq 0$$

$$y$$ follows a discrete Markov process

- Want the policy function, $$c(x, y)$$
Solution Method - Projection and Value Function Iteration

- Approximate $v()$ by projecting it onto a finite dimensional space

$$v(x, a) \approx \tilde{v}(x, a; \theta)$$

  - e.g. splines, orthogonal polynomials, Fourier series, etc.

- Solve for $\theta$ by making $\tilde{v}(x, a; \theta)$ approximately satisfy the Bellman equation

  - Collocation: $\theta$ solves Bellman equation exactly on some grid of points
  - Least squares: $\theta$ minimizes sum of squared errors

$$\theta = \arg \min_{\theta} \int_{X} \left( \tilde{v}(x, a; \theta) - \max_{c, x'} u(c) + \beta E [\tilde{v}(x', y'; \theta)|y] \right)^2 dx$$

  - Galerkin: $\theta$ solves

$$0 = \int_{X} \left( \tilde{v}(x, a; \theta) - \max_{c, x'} u(c) + \beta E [\tilde{v}(x', y'; \theta)|y] \right) f_k(x) dx$$

For basis functions $\{f_k\}_{k=0}^K$
Original Program

- **Good:**
  - Readable – lots of comments
  - Efficient – fast enough

- **Bad:**
  - Not modular at all
  - Does not separate the essence of the problem from the particular assumptions made in this case
    - utility function $\rightarrow$ CRRA
    - approximation to $v()$ $\rightarrow$ cubic spline
    - solution method $\rightarrow$ value function iteration and collocation
    - $y_t$ i.i.d.
% this program solves an Income Fluctuation Problem
% requires obj.m function which is the RHS of the Bellman
% equation as a function of c=consumption and x=cash in hand

global sigma beta yl yh R ppv ppf s1 p1

% possible shocks are in s1 and probability in p1
s1 = [.5 1.5];
p1 = [.5; .5 ];
yl=min(s1); yh=max(s1);

% parameters
% make sure beta*(1+r)<1 !!!
sigma=.5; w = 1; ymean=s1*p1;
beta=.95; \Delta=1/beta-1
r=.02
R=1+r;
xgrid=100;
buffer.m (2)

1 % grid for x — excludes zero to avoid utility==--infinity — low ends
2 %    | note: make sure that xhigh is high enough so that assets come
3 %    | if you iterate and it seems like this is not the case adjust
4 %    | likewise, don't make it too large because it is wasteful        
5 epsilon=1e-3; xhigh= yh*5; xlow = yl;
6 x=(xlow : (xhigh - xlow)/(xgrid-1) :xhigh)';

7 % initial "guess"(es)
8 c         = min((r/1+r)*x + yl/(1+r),x);
9 v_mean    = 1/(1-beta)*r/R*x + ymean).^(1-sigma)/(1-sigma);
10 v_aut    = (r/R*x).^(1-sigma)/(1-sigma) + beta/(1-beta)*(r/R*x);  
11 % use this guess...
12 v        = v_mean;
13 % ... in spline form (for interpolating between grid points)
14 ppv=spline(x,v);
% iterate until convergence
iter=0; crit=1; c_new =c; v_new=v;
while crit >1e−10

  % compute for each point on the grid for x the optimal c
  for i=1:xgrid

    % perform the minimization — obj.m is a function for the
    % the optimization is performed over c between 0 and x(i)
    % current cash available as a parameter, so that obj is
    % the optimizer
    if i>1; c_low=c(i−1) ;else; c_low=0;end; %incorporate the
    c_low=0;
    [c_new(i) bla]=fminbnd('obj', c_low , x(i) , [],x(i));
    v_new(i) = −bla; % note: obj is the negative of the true
  end
% update iteration housekeeping (criterion and iteration number)
v_equiv = ((1-sigma)*(1-beta)*v).^(1/(1-sigma));
v_new_equiv = ((1-sigma)*(1-beta)*v_new).^(1/(1-sigma));
[crit bla1]= max(abs( v_equiv − v_new_equiv ));
[crit_c bla2]= max(abs( (c − c_new)./c ));
crit_c= 100*(c_new(bla2)-c(bla2))/c(bla2);
crit_percent= [v_new_equiv(bla1) − v_equiv(bla1)]*100/ymean;
iter=iter+1;
disp([crit_percent , crit_c*100, iter]);
% display iteration statistic
disp([x(bla1) , x(bla2)]); % displays where the action is

v=v_new; c=c_new; ppv=spline(x,v); % updates information and
% iterate on c(x) policy if its quite stable (using Howard iteration technique)
% make sure this does speed up convergence and not make convergence problematic
% removing this part if necessary
if abs(crit_c*100) < 1e-4
    disp('howard iteration k=100')
    for k=1:100;
        v=-obj(c,x);
       ppv=spline(x,v);
    end
end
end
end
function f = obj(c, x)

    global sigma beta yl yh p R ppv s1 p1;
    u = c.^(1-sigma)/(1-sigma);
    xPrime = R*(x-c)*ones(1,length(s1)) + ones(length(c),1)*s1;
    fi = beta*[ppvval(ppv, xPrime)*p1];
    ff = u + fi;
    f = -ff;
end

We will focus on improving this part of the code.
Better Way

```matlab
1 function f=obj(c,x)
2   global beta R valFn s1 p1;
3   xPrime = R*(x-c)*ones(1,length(s1)) + ones(length(c),1)*s1;
4   f = -(util(c) + beta*evalApprox(valFn,xPrime)*p1);
5 end

1 function u=util(c)
2   global sigma;
3   u = c.^(1-sigma)/(1-sigma);
4 end

1 function f=evalApprox(fn,x)
2   f = ppval(fn,x);
3 end
```
Global Variables

- Often abused – should only be used for constants
- Obfuscates interdependence between parts of a program

```plaintext
1    sigma = 2;
2    u1 = util(1);
3    sigma = 3;
4    u2 = util(1);
```
Alternatives

1. Pass everything as function argument
   - Very explicit
   - Cumbersome
   - Does not make different roles of variables clear

2. Pass arguments in a structure

3. Make the function maintain an internal state – persistent variables
function u = util(c,parm)
% given c, return utility
% parm struc contains parameters that define the utility
% function
% parm has two fields: class = 'crra' or 'cara'
% riskAversion

% check input arguments — probably unnecessary here, but instructive nonetheless
assert(nargin==2,'need 2 input arguments\n');
assert(isstruct(parm),'2nd argument not a struct');
assert(isfield(parm,'class') && isfield(parm,'riskAversion'),
    '2nd arg malformed');
assert(numel(parm.riskAversion)==1,'parm.riskAversion should
switch(parm.class) % type of utility function
  case {'crra','CRRA'}
    if (parm.riskAversion==1)
      u = log(c);
    else
      u = c.^(1-parm.riskAversion)/(1-parm.riskAversion);
    end
  case {'cara','CARA'}
    u = -exp(-parm.riskAversion*c);
  otherwise
    error('%s is not a valid utility function class',parm.class);
  end % switch
end % function util()
Memory Basics

- A basic understanding of memory is useful for understanding function interactions, debugging, and global and persistent variables
- The stack and frames
- Global space
- Persistent space
- Nested Functions
Example Code

```matlab
function g = g(x)
    a = x^2;
    g = a;
end;
```

```matlab
function f = f(x)
    a = exp(x);
    f = g(a);
end;
```

Call stack

- **Frame for function g()**
  - Local variables of g()
    - Not accessible by any other function
    - Will disappear when g() exits
    - g() cannot access anything else

- **Frame for function f()**
  - Local variables of f()
    - If in file g.m, f() cannot be called except for by g() or other subfunctions in g.m
Example Code

function g = g(x)
global e;
a = x.^e;
g = a;
e = e+x;
end;

function f = f(x)
global e c;
e = 3;
f = g(x) + g(g(x));
end;
Example Code

function g = g(x)
    persistent e;
    if isempty(e)
        e = 0;
    end
    e = e+1;
    g = x.^e;
end;

function f = f(x)
    persistent c;
    if isempty(c)
        c = 1;
    end
    c = c*g(c);
    f = g(x)+c;
end;
Example Code

```matlab
function f = f(x)
    x = x + 1;
    f = g(x);
end

function g = g(y)
    a = rand();
    g = y/a+sqrt(x);
end

function y = h(z)
    y = z.*z;
end
```
function u = util(c)
%given a real array c, return utility
%given the string 'get', returns a structure representing
%the parameterization of the utility function
%given a structure, sets the parameterization of the utility function

persistent parm; % utility function parameters

if (isstruct(c))
    parm = c;
    fprintf(['util(): set class=''%s''\n' ...
        '    riskAversion=%g\n'], ...
        parm.class,parm.riskAversion);
    u = [];
    return; % exit now
end
if isempty(parm)
  \% set defaults
  warning('parm not set, using default values\n');
  parm.class = 'crra';
  parm.riskAversion = 1;
end

if ischar(c)
  if ~strcmpi(c,'get')
    warning(['unrecognized character argument, ''%s''\n' ...
             'Returning parm anyway'],c);
  end
  u = parm;
else
else
    switch(parm.class) % type of utility function
    case {'crra','CRRA'}
        if (parm.riskAversion==1)
            u = log(c);
        else
            u = c.^(1-parm.riskAversion)/(1-parm.riskAversion);
        end
    case {'cara','CARA'}
        u = -exp(-parm.riskAversion*c);
    otherwise
        error('%s is not a valid utility function class',parm.class);
    end % switch(parm.class)
end % if ischar(c) ... else ...
end % function util(c)
Another Approach

function f=obj(c,x)
global beta R valFn s1 p1;
xPrime = R*(x−c)*ones(1,length(s1)) + ones(length(c),1)*s1;
f = −(util(c) + beta*evalApprox(valFn,xPrime)*p1);
end

- \( \text{obj}(c,x) \) computes the expected value of consuming \( c \) today
  - Problem: \( \text{obj}(c,x) \) depends on lots of nuisance parameters
  - Problem: generality increases the number of nuisance parameters and complicates the code
- Solution: stop trying to pass \( \text{obj}() \) parameters to functions and give it functions instead
Function Handles

- Assign a function to a variable:

```matlab
U = @util;  % now U(c) and util(c) are equivalent commands
```

- Define an inline function:

```matlab
U = @(c) log(c);  % now U(c) = log(c)
```

- Function handles are just like normal variables. They can be passed as function arguments and be part of structures and cell arrays.
obj() with Function Handles

function f=obj(c,x,pm)

% return −E(u(c)+ beta *v(x')|x)
% c = vector, consumption today
% x = current state
% pm = structure with u(), beta, v(), s, p;

xp = pm.R*(x−c)*ones(1,length(pm.s)) + …
ones(length(c),1)*pm.s;

f = −(pm.U(c) + pm.beta*pm.V(xPrime)*pm.p);
end
Call to obj() with Function Handles

```
1  % constructing parm
2  parm.beta = 0.95;
3  parm.R = 1.02;
4  parm.U = @(c) c.^(1-sigma)/(1-sigma);
5  parm.V = @(x) ppval(valFn,x);
6  % note that subsequent changes to sigma and valFn will not change
7  % parm.U and parm.V
8
9  % maximize the value
10 [c ev] = fminbnd(@(c) obj(c,x,parm), cLo, cHi);
```
buffer.m with Function Handles

1 % this program solves an Income Fluctuation Problem
2 % requires obj.m function which is the RHS of the Bellman
3 % equation as a function of c=consumption and x=cash in hand
4
5 % possible shocks are in s1 and probability in p1
6 s1  = [.5 1.5];
7 p1  = [.5;.5 ];
8 yl=min(s1); yh=max(s1);
9
10 % parameters
11 % make sure beta*(1+r)<1 !!!
12 sigma=.5; w = 1; ymean=s1*p1;
13 beta=.95; Δ=1/beta−1
14 r=.02
15 R=1+r;
16 xgrid=100;
buffer.m with Function Handles

1    parm.R = R;
2    parm.beta = beta;
3    parm.p = p1;
4    parm.s = s1;
5    parm.U = @(c) c.^(1-sigma)/(1-sigma);
6    parm.V = @(x) x; % updated below
epsilon = 1e−3; xhigh = yh * 5; xlow = yl;
x = (xlow : (xhigh − xlow)/(xgrid−1) : xhigh)';

% initial "guess"
c = min((r/(1+r)*x + yl/(1+r)), x);
v_mean = 1/(1−beta)*(r/R*x + ymean).^(1−sigma)/(1−sigma);
% use this guess...
v = v_mean;
% ... in spline form (for interpolating between grid points)
ppv = spline(x, v);
parm.V = @(x) ppval(ppv, x);
buffer.m with Function Handles

1 % iterate until convergence
2 iter=0; crit=1; c_new =c; v_new=v;
3 while crit >1e−10

5 % compute for each point on the grid for x the optimal c
6 for i=1:xgrid

8 % perform the minimization — obj.m is a function for the
9 % the optimization is performed over c between 0 and x(i)
10 % current cash available as a parameter, so that obj is
11 % the optimizer
12 if i>1; c_low=c(i−1) ;else; c_low=0;end; %incorporate that
13 c_low=0;
14 [c_new(i) bla]=fminbnd(@(c) obj(c,x(i),parm), c_low , x(i));
15 v_new(i) = −bla; % note: obj is the negative of the true
16 end
buffer.m with Function Handles

```matlab
% update iteration housekeeping (criterion and iteration number)
v_equiv = ((1-sigma)*(1-beta)*v).^(1/(1-sigma));
v_new equiv = ((1-sigma)*(1-beta)*v_new).^(1/(1-sigma));
[crit bla1]= max(abs( v_equiv – v_new equiv ));
[crit c bla2]= max(abs( (c – c_new)./c )); crit c= 100*(c_new(bla2)– c(bla2))/c(bla2);
crit percent= [v_new equiv(bla1) – v_equiv(bla1)]*100/ymean;
iter=iter+1;
disp([crit_percent , crit c*100, iter]); % display iteration statistic
disp([x(bla1) , x(bla2)]); % displays where the action is

v=v_new; c=c_new; ppv=spline(x,v); % updates information and
% iteration
parm.V = @(x) ppval(ppv,x);
```
buffer.m with Function Handles

1 % iterate on c(x) policy if its quite stable (using Howard iteration technique)
2 % make sure this does speed up convergence and not make convergence problematics
3 % removing this part if necessary
4 if abs(crit_c*100) < 1e-4
5   disp('howard iteration k=100')
6     for k=1:100;
7       v=-obj(c,x,parm);
8       ppv=spline(x,v);
9       parm.V = @(x) ppval(ppv,x);
10      end
11   end
12 end
13
14 end
Exercises

There are many ways to further extend this program. In order from easiest to hardest:

1. Most of the lecture focused on how to make the utility function more general. Similarly, you might want to try to generalize the value function approximation method. Modify the program so that by changing a single parameter at the beginning, a user can choose a different approximation method. Functions for linear interpolation (interp1()) and cubic Hermite polynomial interpolation are included in Matlab, so it should be easy to use either of those. Alternatively, you can try a series approximation method. The folder of code named v4-pset, contains a version of the program that uses Chebyshev polynomials.

2. A good way to test the program would be to see whether the solution satisfies the Euler equation. Write a function that computes and reports the error in the Euler equation. There should be no error at the points included in the collocation grid. There will be some error off the grid. This is a good measure of the quality of the approximation.

3. Change the method of choosing the “best” $\tilde{v}(\cdot)$. The program currently does collocation. Try to implement least squares or the Galerkin method.

4. Right now we’re using the Bellman equation to solve for the value function. An alternative approach would be to use the Euler equation to solve for the policy function. Do it.

At the end, imagine that someone asks you to try another variation on the part of the program you just modified. How difficult is the change?
% this program solves an Income Fluctuation Problem

clear functions; clear variables;
addpath approx -end; % add approx directory to path

% parameters and settings
parm.states = [.5 1.7]; % possible states
parm.tran = [.5 .5; .5 .5]; % columns of tran equal -> iid
ra = 3;
parm.U = @(c) c.^(1-ra)/(1-ra);
parm.dU = @(c) c.^(-ra); % derivative of U
parm.w = 1;
parm.beta = 0.97;
parm.R = 1.02;

% solution method
parm.approx.method = 'spline'; % options are 'chebyshev', 'spline'
parm.approx.order = 20; % order of chebyshev approx or number of points for spline
parm.approx.xlow = min(parm.states)/10; % min assets in grid
parm.approx.xhigh = max(parm.states)*20; % max assets in grid,
main.m (2)

1 % convergence criteria
2 TOL = 1.0e-4; % tolerance
3 MAXITER = 1e6; % maximum number of iterations
4
5 % solve for the value and policy functions
6 [vfunc cfunc] = solveValue(parm,TOL,MAXITER);
7 checkEulerEqn(parm,cfunc,vfunc);
function [vfunc cfunc] = solveValue(parm,TOL,MAXITER);
% compute the value function by iterating on the bellman equation

% initialize approximation function struct
for s = 1:numel(parm.states)
    vfunc(s) = newApproxFn(parm.approx);
end

% compute an initial guess
for s=1:numel(parm.states)
    c(:,s) = vfunc(s).grid;
    val(:,s) = parm.U(vfunc(s).grid);
    vfunc(s) = updateApproxFn(val(:,s),vfunc(s));
end
while ((critV > TOL || critC > TOL || iter<10) && iter≤MAXITER)
    for s=1:length(parm.states)
        % compute for each point on the grid for x the optimal c
        for i=1:length(vfunc(s).grid)
            [c_new(i,s) v_new(i,s)] = ...
                fminbnd(@(c) obj(c,vfunc(s).grid(i), s,vfunc(s), parm),
                  0,vfunc(s).grid(i));
        end
    end
    v_new = −v_new; % b/c obj = −val
    critV = max(max(abs((val−v_new)./val)));
    critC = max(max(abs((c−c_new)./c)));
    for s=1:length(parm.states)
        val(:,s) = v_new(:,s);
        vfunc(s) = updateApproxFn(val(:,s),vfunc(s));
    end
    c = c_new;
    iter = iter+1;
    if (mod(iter,20)==0)
        fprintf(' %4d: critC = %.4g, critV = %.4g
',iter,critC,critV)
    end
end
if (iter ≤ MAXITER)
    fprintf('Convergence, crit = %g, iter=%d\n',critV,iter)
else
    fprintf('Maximum iterations reached\n');
end

cfunc = vfunc; % use same settings as vfunc
for s=1:length(parm.states)
    cfunc(s) = updateApproxFn(c_new(:,s),cfunc(s));
end
end % function solveValue()
function [f df]=obj(c,x,s,vfunc,pm)
% return \(-E(u(c) + \beta \cdot v(x'))|x\)
% c = vector, consumption today
% x = current assets
% s = current state index
% V = value function
% pm = structure with U(), beta, s, p;

xP = pm.R*(x-c)*ones(1,length(pm.states)) + ...
    pm.w*ones(length(c),1)*pm.states;
V = evalApproxFn(vfunc,xP);
f = -pm.U(c) + pm.beta*V*pm.tran(:,s);

% return gradient / Euler Equation
if (nargout>1)
    df = -(pm.dU(c) - pm.R*pm.beta ...
        *diffApproxFn(vfunc,xP)*pm.tran(:,s));
end
end
function checkEulerEqn(parm, cfunc, vfunc)
% computes the error in the euler equation

x = (parm.approx.xlow: ...
    (parm.approx.xhigh–parm.approx.xlow)/(5*parm.approx.order):
    parm.approx.xhigh)';
for s=1:length(parm.states)
    for i=1:length(x)
        c(i,s)=evalApproxFn(cfunc(s),x(i));
        [v ee(i,s)] = obj(c(i,s),x(i),s,vfunc(s),parm);
    end
end

for s=1:length(parm.states)
    fprintf('state %d, quintiles(ee) = %g %g %g %g\n', ...
        s,quantile(ee(:,s),[0.2,0.4,0.6,0.8]));
    fprintf('max|ee(:,s)| = %g, argmax = %g\n',m,x(i));
end
approxFn

- `approx/` directory contains functions for dealing with approximation – `newApproxFn`, `evalApproxFn`, `diffApproxFn`, `updateApproxFn`
- Idea: `approxFn` is a new datatype that we modify and use only through these functions
- Object-oriented programming formalizes this idea
  - Key terms: class, method, inheritance
  - Languages: Java, C++, and C#
- Matlab has facilities for object-oriented programming (but `approxFn` does not use them)