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

$$v(x,y) = \max_{c,x'} u(c) + \beta E\left[v(x',y')|y\right] \quad \text{s.t.}$$

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

$$x' \geq 0$$

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 θ by making $\tilde{v}(x, a; \theta)$ approximately satisfy the Bellman equation
  - Collocation: θ solves Bellman equation exactly on some grid of points
  - Least squares: θ minimizes sum of squared errors

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

  - Galerkin: θ solves

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

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

Original Program

Link to code

- Good:
  - Readable – lots of commments
  - 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 $\longrightarrow$ CRRA
    - ★ approximation to $v()$ $\longrightarrow$ cubic spline
    - ★ solution method $\longrightarrow$ value function iteration and collocation
    - ★ y_t i.i.d.

buffer.m (1)

```
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  global sigma beta yl yh R ppv ppf s1 p1
6
7  % possible shocks are in s1 and probability in p1
8  s1  = [.5 1.5];
9  p1  = [.5;.5 ];
10 yl=min(s1); yh=max(s1);
11 % parameters
12 % make sure beta*(1+r)<1 !!!
13 sigma=.5; w = 1; ymean=s1*p1;
14 beta=.95;  Δ=1/beta-1
15 r=.02
16 R=1+r;
17 xgrid=100;
```

buffer.m (2)

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

% initial "guess"(es)
c       = min((r/1+r)*x + yl/(1+r),x);
v_mean  = 1/(1-beta)*(r/R*x + ymean).^(1-sigma)/(1-sigma);
v_aut   = (r/R*x).^(1-sigma)/(1-sigma) + beta/(1-beta)*(r/R*x*(
% use this guess...
v       = v_mean;
% ... in spline form (for interpolating between grid points)
ppv=spline(x,v);
```

buffer.m (3)

```
% 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 optimization is performed over c between 0 and x
        % 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
        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 tru
    end
```

buffer.m (4)

```
1     % update interation housekeeping (criterion and iteration
2     v_equiv     = ((1-sigma)*(1-beta)*v).^(1/(1-sigma));
3     v_new_equiv = ((1-sigma)*(1-beta)*v_new).^(1/(1-sigma));
4     [crit bla1]= max(abs( v_equiv - v_new_equiv ));
5     [crit_c bla2]= max(abs( (c - c_new)./c )); crit_c= 100*(c_
6     crit_percent= [v_new_equiv(bla1) - v_equiv(bla1)]*100/ymea
7     iter=iter+1;
8     disp([crit_percent , crit_c*100, iter]) ; % display iterat
9     disp([x(bla1) , x(bla2)])  ; %displays where the action is
10
11    v=v_new; c=c_new; ppv=spline(x,v); % updates information a
```

buffer.m (5)

```
1       % iterate on c(x) policy if its quite stable (using Howard
2       % make sure this does speed up convergence and not make co
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);
8               ppv=spline(x,v);
9           end
10      end
11
12  end
```

obj.m – Right side of Bellman equation

```
1 function f=obj(c,x)
2   global sigma beta yl yh p R ppv s1 p1;
3   u = c.^(1-sigma)/(1-sigma);
4   xPrime = R*(x-c)*ones(1,length(s1)) + ones(length(c),1)*s1;
5   fi= beta*[ppval(ppv,xPrime)*p1];
6   ff = u + fi;
7   f = -ff;
8 end
```

- We will focus on improving this part of the code

Better Way

```
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
```

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

```
function f=evalApprox(fn,x)
  f = ppval(fn,x);
end
```

Global Variables

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

```
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

util() using Structures (1 of 2)

```
1  function u = util(c,parm)
2  % given c, return utility
3  % parm struc contains parameters that define the utility
4  % function
5  % parm has two fields: class = 'crra' or 'cara'
6  %                       riskAversion
7
8  % check input arguments - probably unecessary here, but
9  %                         instructive nonetheless
10   assert(nargin==2,'need 2 input arguments\n');
11   assert(isstruct(parm),'2nd argument not a struct');
12   assert(isfield(parm,'class') && isfield(parm,'riskAversion'),
13          '2nd arg malformed');
14   assert(numel(parm.riskAversion)==1,'parm.riskAversion should
```

util() using Structures (2 of 2)

```
1    switch(parm.class) % type of utility function
2     case {'crra','CRRA'}
3      if (parm.riskAversion==1)
4        u =log(c);
5      else
6        u = c.^(1-parm.riskAverion)/(1-parm.riskAverion);
7      end
8     case {'cara','CARA'}
9      u = -exp(-parm.riskAversion*c);
10    otherwise
11     error('%s is not a valid utility function class',parm.class
12   end % switch
13 end % function util()
```

Memory Basics

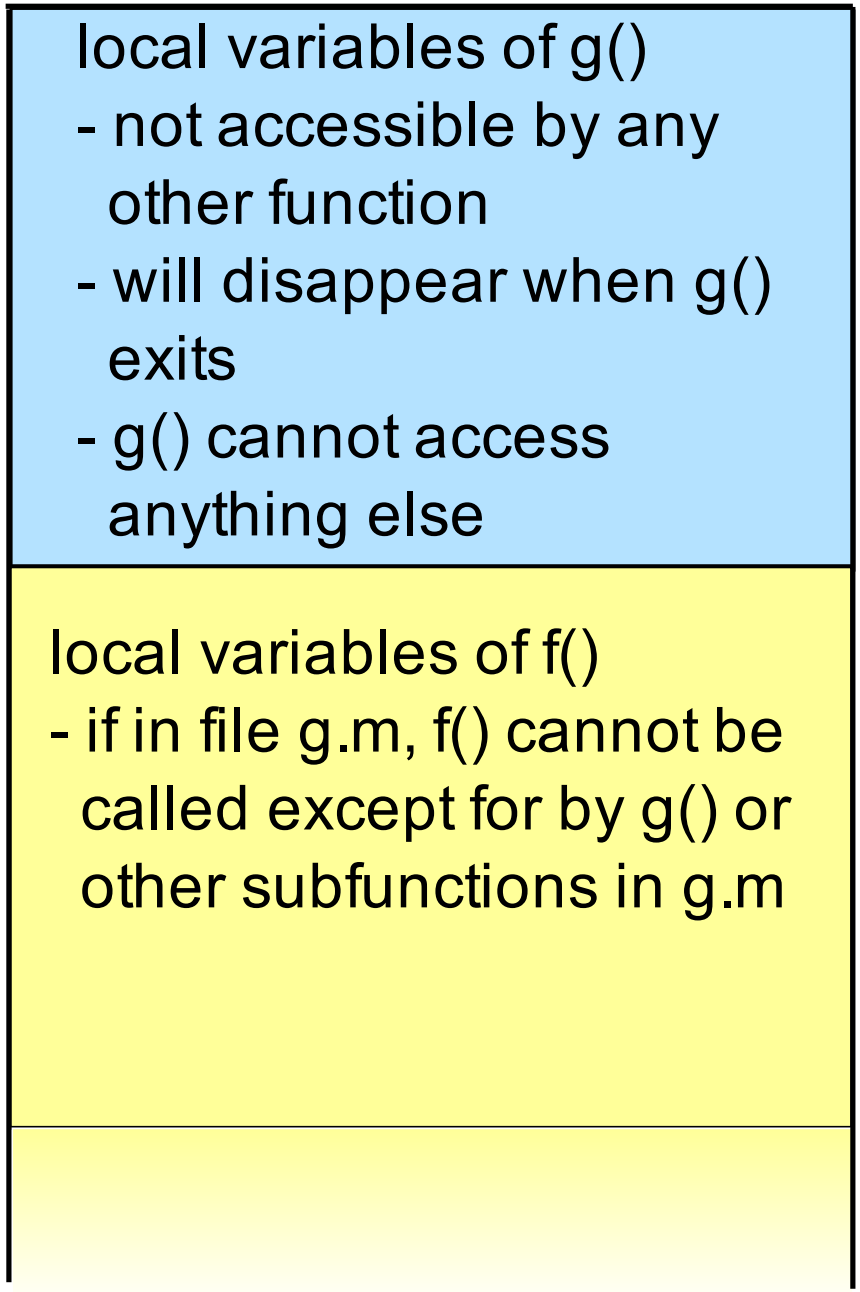
- A basic understanding of memory is useful for understanding function interactions, debugging, and global and persistent variables
- The stack and frames
- Global space
- Persistent space
- Nested Functions

Example Code

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

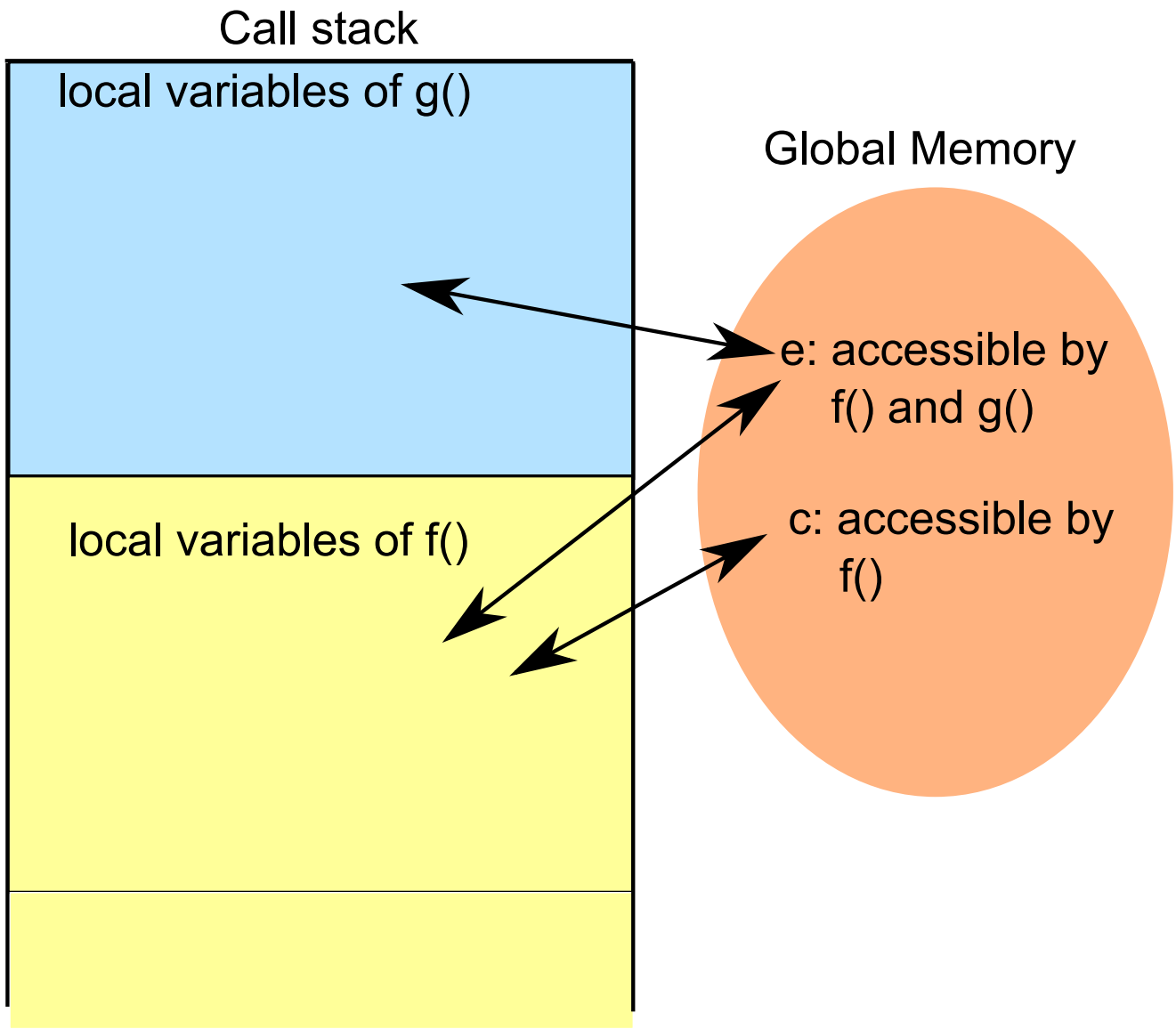
```
function f = f(x)
  a = exp(x);
  f = g(a);
end;
```

Call stack



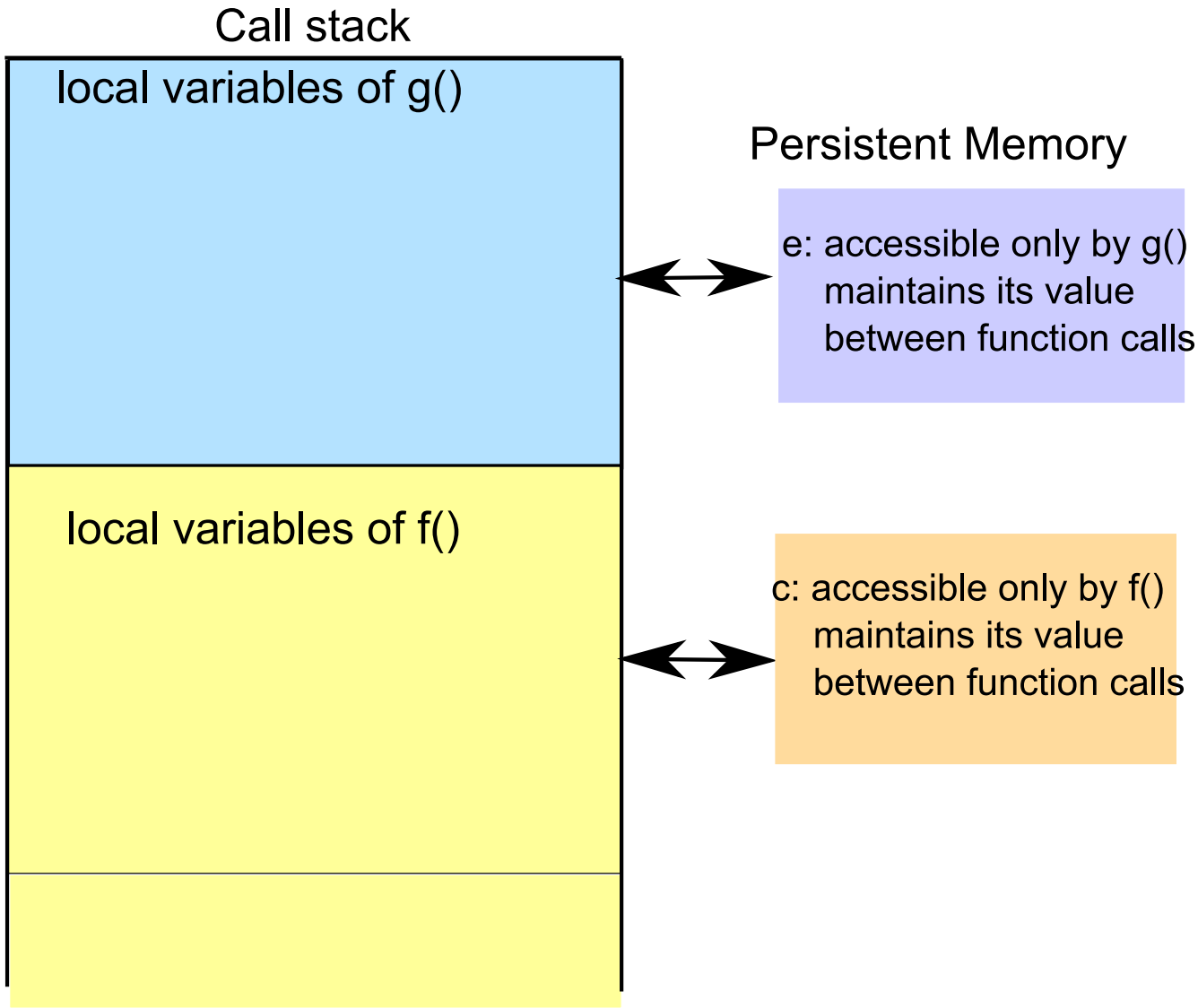
Example Code

```
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

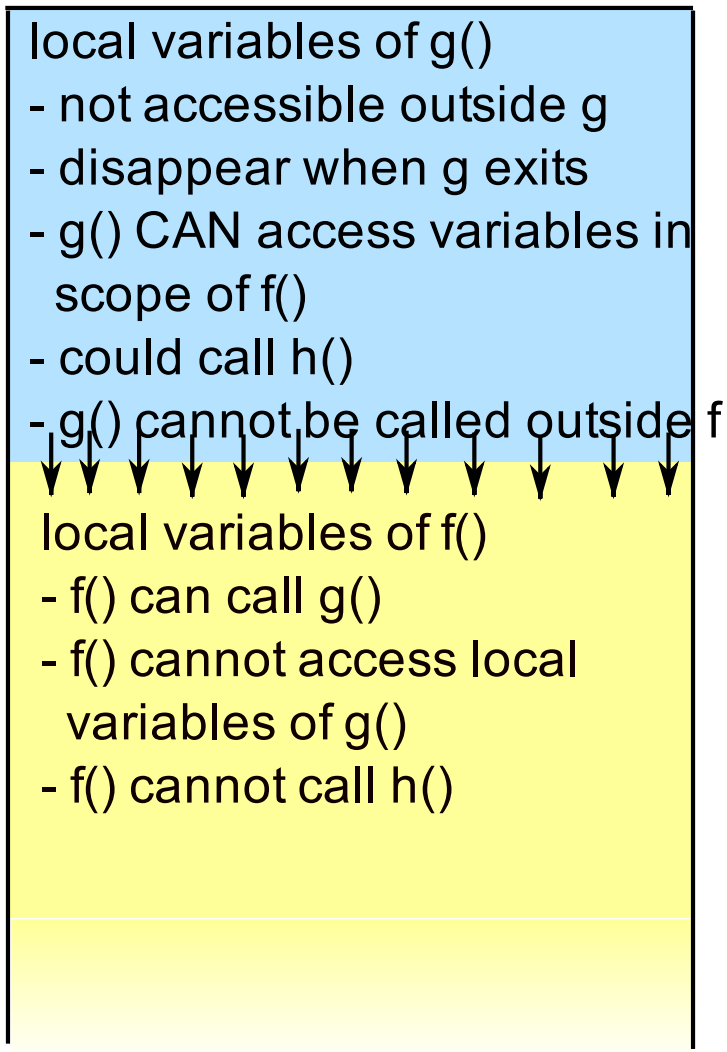
```
function g = g(x)
  persistent e;
  if isempty(e)
    e = 0;
  end
  e = e+1;
  g = x.^e;
end;

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



Example Code

```
function f = f(x)
  x = x + 1;
  f = g(x);
  function g = g(y)
    a = rand();
    g = y/a+sqrt(x);
    function y = h(z)
      y = z.*z;
    end
  end
end
```

Call stack



Persistent `util()` (1 of 3)

```
1  function u = util(c)
2  %given a real array c, return utility
3  %given the string 'get', returns a structure representing
4  % the parameterization of the utility function
5  %given a structure, sets the parameterization of the utility fu
6
7    persistent parm; % utility function parameters
8
9    if (isstruct(c))
10     parm = c;
11     fprintf(['util(): set class=''%s''\n' ...
12              '            riskAverion=%g\n'], ...
13             parm.class,parm.riskAversion);
14     u = [];
15     return; % exit now
16   end
```

Persistent `util()` (2 of 3)

```
1    if isempty(parm)
2      % set defaults
3      warning('parm not set, using default values\n');
4      parm.class = 'crra';
5      parm.riskAversion = 1;
6    end
7
8    if ischar(c)
9      if ¬strcmpi(c,'get')
10       warning(['unrecognized character argument, ''%s''\n' ...
11                'Returning parm anyway'],c);
12     end
13     u = parm;
14   else
```

Persistent `util()` (3 of 3)

```
1     else
2       switch(parm.class) % type of utility function
3        case {'crra','CRRA'}
4         if (parm.riskAversion==1)
5           u =log(c);
6         else
7           u = c.^(1-parm.riskAverion)/(1-parm.riskAverion);
8         end
9        case {'cara','CARA'}
10        u = -exp(-parm.riskAversion*c);
11       otherwise
12        error('%s is not a valid utility function class',parm.cl
13      end % switch(parm.class)
14    end % if ischar(c) ... else ...
15  end % function util(c)
```

Another Approach

```
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
```

- `obj(c,x)` computes the expected value of consuming c today
  - Problem: `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 `obj()` parameters to functions and give it functions instead

Function Handles

- Assign a function to a variable:

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

- Define an inline function:

```
1          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

```
1  function f=obj(c,x,pm)
2  % return -E(u(c)+ beta *v(x')|x)
3  % c = vector, consumption today
4  % x = current state
5  % pm = structure with u(), beta, v(), s, p;
6
7    xP = pm.R*(x-c)*ones(1,length(pm.s)) + ...
8         ones(length(c),1)*pm.s;
9    f = -(pm.U(c) + pm.beta*pm.V(xPrime)*pm.p);
10 end
```

Call to obj() with Function Handles

```
% constructing parm
parm.beta = 0.95;
parm.R = 1.02;
parm.U = @(c) c.^(1-sigma)/(1-sigma);
parm.V = @(x) ppval(valFn,x);
% note that subsequent changes to sigma and valFn will not cha
% parm.U and parm.V

% maximize the value
[c ev] = fminbnd(@(c) obj(c,x,parm), cLo, cHi);
```

buffer.m with Function Handles

```
% 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

% 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;  Δ=1/beta-1
r=.02
R=1+r;
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
```

buffer.m with Function Handles

```
1  epsilon=1e-3; xhigh= yh*5; xlow = yl;
2  x=(xlow : (xhigh - xlow)/(xgrid-1) :xhigh)';
3
4  % initial "guess"
5  c       = min((r/1+r)*x + yl/(1+r),x);
6  v_mean  = 1/(1-beta)*(r/R*x + ymean).^(1-sigma)/(1-sigma);
7  % use this guess...
8  v       = v_mean;
9  % ... in spline form (for interpolating between grid points)
10 ppv=spline(x,v);
11 parm.V = @(x) ppval(ppv,x);
```

buffer.m with Function Handles

```
% 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 optimization is performed over c between 0 and x
        % 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
        c_low=0;
        [c_new(i) bla]=fminbnd(@(c) obj(c,x(i),parm), c_low , x
        v_new(i) = -bla; % note: obj is the negative of the tru
    end
```

buffer.m with Function Handles

```
1       % update interation housekeeping (criterion and iteration
2       v_equiv     = ((1-sigma)*(1-beta)*v).^(1/(1-sigma));
3       v_new_equiv = ((1-sigma)*(1-beta)*v_new).^(1/(1-sigma));
4       [crit bla1]= max(abs( v_equiv - v_new_equiv ));
5       [crit_c bla2]= max(abs( (c - c_new)./c )); crit_c= 100*(c_n
6       crit_percent= [v_new_equiv(bla1) - v_equiv(bla1)]*100/ymean
7       iter=iter+1;
8       disp([crit_percent , crit_c*100, iter]) ; % display iterati
9       disp([x(bla1) , x(bla2)])  ; %displays where the action is
10
11      v=v_new; c=c_new; ppv=spline(x,v); % updates information at
12                                         % iteration
13      parm.V = @(x) ppval(ppv,x);
```

buffer.m with Function Handles

```
1
2       % iterate on c(x) policy if its quite stable (using Howard
3       % make sure this does speed up convergence and not make co
4       % removing this part if necessary
5       if abs(crit_c*100) < 1e-4
6           disp('howard iteration k=100')
7           for k=1:100;
8               v=-obj(c,x,parm);
9               ppv=spline(x,v);
10              parm.V = @(x) ppval(ppv,x);
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 errror off the grid. This is a good measure of the quality of the approximation.
3. Change the method of choosing the "best" $\tilde{v}()$. 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?

main.m (1)

```
% 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;   % columns of tran equal -> iid
             .5 .5];
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', 'spl
parm.approx.order =20; % order of chebyshev approx or number o
parm.approx.xlow = min(parm.states)/10;     % min assets  in g
parm.approx.xhigh = max(parm.states)*20; % max assets in grid,
```

main.m (2)

```
% convergence criteria
TOL = 1.0e-4;   % tolerance
MAXITER = 1e6;  % maximum number of iterations

% solve for the value and policy functions
[vfunc cfunc] = solveValue(parm,TOL,MAXITER);
checkEulerEqn(parm,cfunc,vfunc);
```

solveValue.m (1)

```
1  function [vfunc cfunc] = solveValue(parm,TOL,MAXITER);
2  % compute the value function by iterating on the bellman equat
3
4    % initialize approximation function struct
5    for s = 1:numel(parm.states)
6      vfunc(s) = newApproxFn(parm.approx);
7    end
8
9    % compute an initial guess
10   for s=1:numel(parm.states)
11     c(:,s) = vfunc(s).grid;
12     val(:,s) = parm.U(vfunc(s).grid);
13     vfunc(s) = updateApproxFn(val(:,s),vfunc(s));
14   end
```

solveValue.m (2)

```
1   while ((critV > TOL || critC > TOL || iter<10) && iter≤MAXIT
2     for s=1:length(parm.states)
3       % compute for each point on the grid for x the optimal c
4       for i=1:length(vfunc(s).grid)
5         [c_new(i,s) v_new(i,s)]= ...
6             fminbnd(@(c) obj(c,vfunc(s).grid(i), s,vfunc(s), pa
7                                                 0,vfunc(s).grid(i));
8       end
9     end
10    v_new = -v_new; % b/c obj = -val
11    critV = max(max(abs((val-v_new)./val)));
12    critC = max(max(abs((c-c_new)./c)));
13    for s=1:length(parm.states)
14      val(:,s) = v_new(:,s);
15      vfunc(s) = updateApproxFn(val(:,s),vfunc(s));
16    end
17    c = c_new;
18    iter = iter+1;
19    if(mod(iter,20)==0)
20      fprintf('%4d: critC = %.4g, critV = %.4g \n',iter,critC,c
```

solveValue.m (3)

```
1     if(iter≤MAXITER)
2       fprintf('Convergence, crit = %g, iter=%d\n',critV,iter)
3     else
4       fprintf('Maximum iterations reached\n');
5     end
6
7     cfunc = vfunc; % use same settings as vfunc
8     for s=1:length(parm.states)
9       cfunc(s) = updateApproxFn(c_new(:,s),cfunc(s));
10    end
11  end % function solveValue()
```

obj.m

```
function [f df]=obj(c,x,s,vfunc,pm)
% return -E(u(c)+ beta *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
```

checkEulerEqn.m

```
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.orde
       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
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)