Macroeconomics

Lecture 7: dynamic programming methods, part five

Chris Edmond

1st Semester 2019

This class

- A more flexible approach to practical dynamic programming
- Use techniques for function interpolation and approximation
- In particular, a method known as *collocation*
- Details implemented using CompEcon toolkit for Matlab

Main idea

- Bellman equation for the optimal growth model

$$v(k) = \max_{c \geq 0} \left[u(c) + \beta v(f(k) - c) \right]$$

where $k' = f(k) - c$ denotes the capital stock chosen for next period

- Suppose we can write

$$v(k) \approx \sum_{j=1}^{n} a_j \phi_j(k)$$

using n known *basis functions* $\phi_j(k)$ with *coefficients* a_j

Main idea

- Solve for n coefficients a_j by solving Bellman equation at n given *collocation nodes* k_i for $i = 1, ..., n$

- That is, find n coefficients a_j such that

$$\sum_{j=1}^{n} a_j\, \phi_j(k_i) = \max_{c\geq 0} \left[u(c)+\beta \sum_{j=1}^{n} a_j\, \phi_j(f(k_i)-c))\right], \qquad i = 1, ..., n$$

- This is a system of n nonlinear equations in n unknowns

Notation

- Let Φ_{ij} denote the elements of the *collocation matrix*

$$\Phi_{ij} = \phi_j(k_i), \qquad i, j = 1, ..., n$$

To form this we need the n basis functions and n nodes

- LHS of Bellman equation is then

$$\boldsymbol{\Phi a}$$

- RHS of Bellman equation is a vector-valued function $\boldsymbol{v}(\boldsymbol{a})$ with typical element

$$v_i(\boldsymbol{a}) = \max_{c \geq 0} \left[u(c) + \beta \sum_{j=1}^{n} a_j \, \phi_j(f(k_i) - c)) \right], \qquad i = 1, ..., n$$

Notice that in this formulation $v_i(\cdot)$ is a known function

Collocation equation

- In short, want to find vector $\boldsymbol{a}$ that solves *collocation equation*

$$\boldsymbol{\Phi}\boldsymbol{a} = \boldsymbol{v}(\boldsymbol{a})$$

- For example, can do *function iteration* on the coefficients by

$$\boldsymbol{a}^{l+1} = \boldsymbol{\Phi}^{-1}\boldsymbol{v}(\boldsymbol{a}^{l}), \qquad l = 0, 1, 2, \ldots$$

starting from some $\boldsymbol{a}^0$ and iterating until

$$\|\boldsymbol{a}^{l+1} - \boldsymbol{a}^{l}\| = \max_i \left[\, |\, a_i^{l+1} - a_i^{l} \,|\, \right] < \varepsilon$$

for some some pre-specified tolerance $\varepsilon > 0$

- Can often improve on this using *Newton's method* (or variants)

Aside on Newton's method

- Suppose we want to find a root x such that $f(x) = 0$ for some $f(\cdot)$
- Let x^0 be an initial guess and suppose $f(x^0) \neq 0$
- Consider first-order approximation of $f(x)$ around x^0

$$f(x) \approx f(x^0) + f'(x^0)(x - x^0)$$

- Then find x^1 such that this linear approximation is zero

$$x^1 = x^0 - f'(x^0)^{-1} f(x^0)$$

And

$$x^{l+1} = x^l - f'(x^l)^{-1} f(x^l), \qquad l = 0, 1, 2, \ldots$$

Aside on Newton's method

- Similar idea for vector-valued functions

$$\boldsymbol{f}(\boldsymbol{x}) \approx \boldsymbol{f}(\boldsymbol{x}^0) + \boldsymbol{f}'(\boldsymbol{x}^0)(\boldsymbol{x} - \boldsymbol{x}^0)$$

  where $\boldsymbol{f}'(\boldsymbol{x}^0)$ is the *Jacobian* of $\boldsymbol{f}(\boldsymbol{x})$ evaluated at $\boldsymbol{x}^0$ — i.e., the matrix of partial derivatives of the form

$$\frac{\partial f_i(\boldsymbol{x}^0)}{\partial x_j}, \qquad i, j = 1, ..., n$$

  where $f_i(\boldsymbol{x})$ denotes the typical element of the vector $\boldsymbol{f}(\boldsymbol{x})$

- Implies iterative scheme

$$\boldsymbol{x}^{l+1} = \boldsymbol{x}^l - \boldsymbol{f}'(\boldsymbol{x}^l)^{-1}\boldsymbol{f}(\boldsymbol{x}^l), \qquad l = 0, 1, 2, \ldots$$

Applying Newton's method to our problem

- For our problem we are trying to find $\boldsymbol{a}$ such that

$$\boldsymbol{f}(\boldsymbol{a}) \equiv \boldsymbol{\Phi}\boldsymbol{a} - \boldsymbol{v}(\boldsymbol{a}) = \boldsymbol{0}$$

- The Jacobian of $\boldsymbol{f}(\boldsymbol{a})$ is the matrix

$$\boldsymbol{f}'(\boldsymbol{a}) = \boldsymbol{\Phi} - \boldsymbol{v}'(\boldsymbol{a})$$

- Implies iterative scheme

$$\boldsymbol{a}^{l+1} = \boldsymbol{a}^{l} - [\boldsymbol{\Phi} - \boldsymbol{v}'(\boldsymbol{a}^{l})]^{-1}\,[\boldsymbol{\Phi}\boldsymbol{a}^{l} - \boldsymbol{v}(\boldsymbol{a}^{l})], \qquad l = 0, 1, 2, \ldots$$

Applying Newton's method to our problem

- The Jacobian of $\boldsymbol{v}(\boldsymbol{a})$ has elements

$$\frac{\partial v_i(\boldsymbol{a})}{\partial a_j} = \beta\, \phi_j(f(k_i) - c(k_i\,;\,\boldsymbol{a}))$$

- This uses the Envelope theorem — i.e, we can ignore the indirect effects of a_j that come through the optimal policy

$$c(k_i\,;\,\boldsymbol{a}) = \operatorname*{argmax}_{c \geq 0} \left[u(c) + \beta \sum_{j=1}^{n} a_j\, \phi_j(f(k_i) - c)) \right]$$

- To implement this, we need to choose a *basis-node* scheme

- Common choices include *Chebychev polynomials* and piecewise polynomial *splines*

Chebychev polynomials

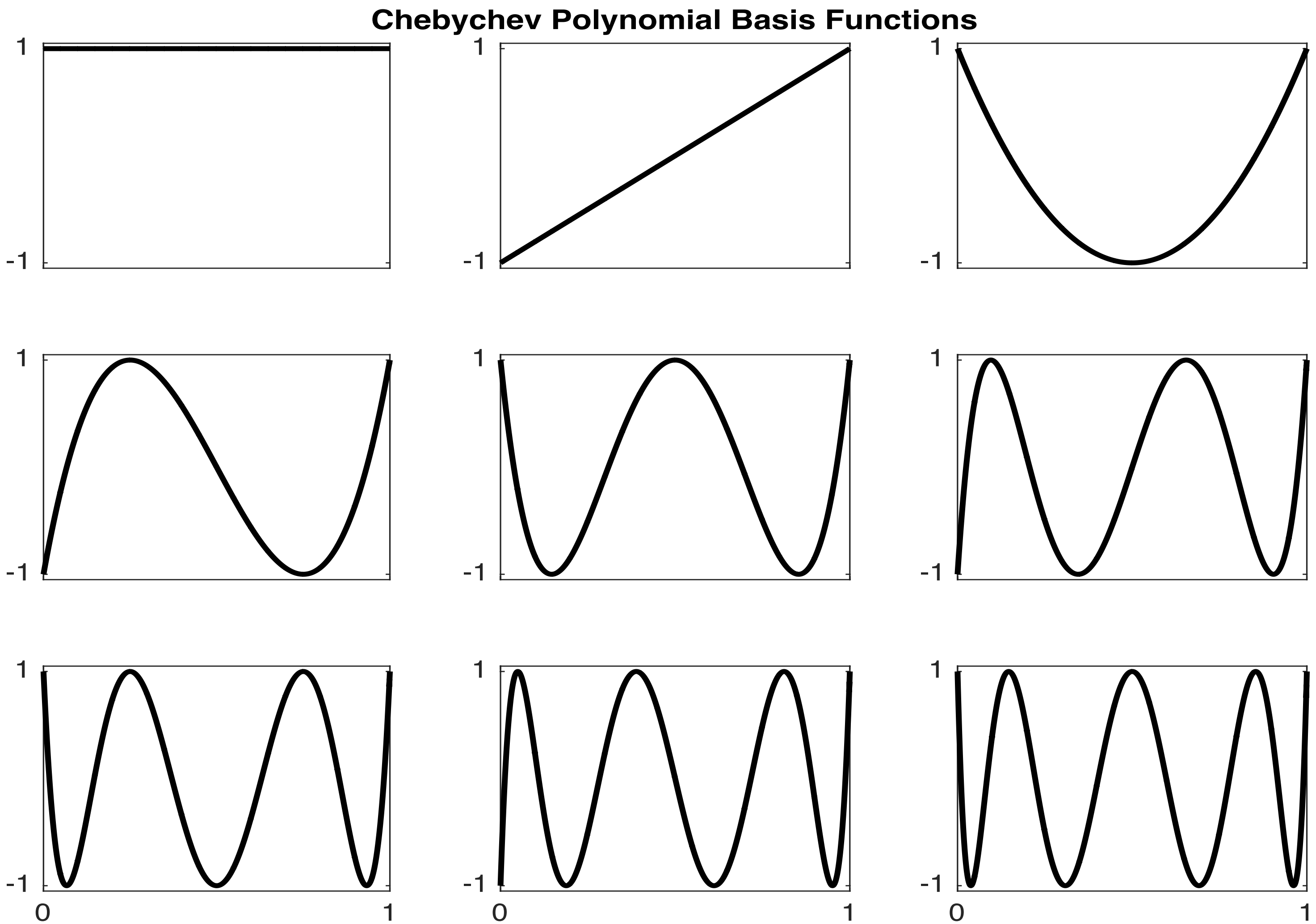
- For some $x \in [a, b]$ the jth basis function is

$$\phi_j(x) = P_{j-1}(z), \qquad z = 2\frac{x-a}{b-a} - 1$$

where the polynomials on $z \in [-1 + 1]$ are given by

$$\begin{aligned}
P_0(z) &= 1 \\
P_1(z) &= z \\
P_2(z) &= 2z^2 - 1 \\
&\vdots \\
P_j(z) &= 2zP_{j-1}(z) - P_{j-2}(z), \qquad j = 2, 3, \ldots
\end{aligned}$$

Chebychev polynomials



Linear splines

- For some $x \in [a, b]$ with n evenly-spaced breakpoints t_j

$$\phi_j(x) = \begin{cases} 1 - \dfrac{|x - t_j|}{h} & \text{if } |x - t_j| < h \\ 0 & \text{otherwise} \end{cases}$$

where the breakpoints are

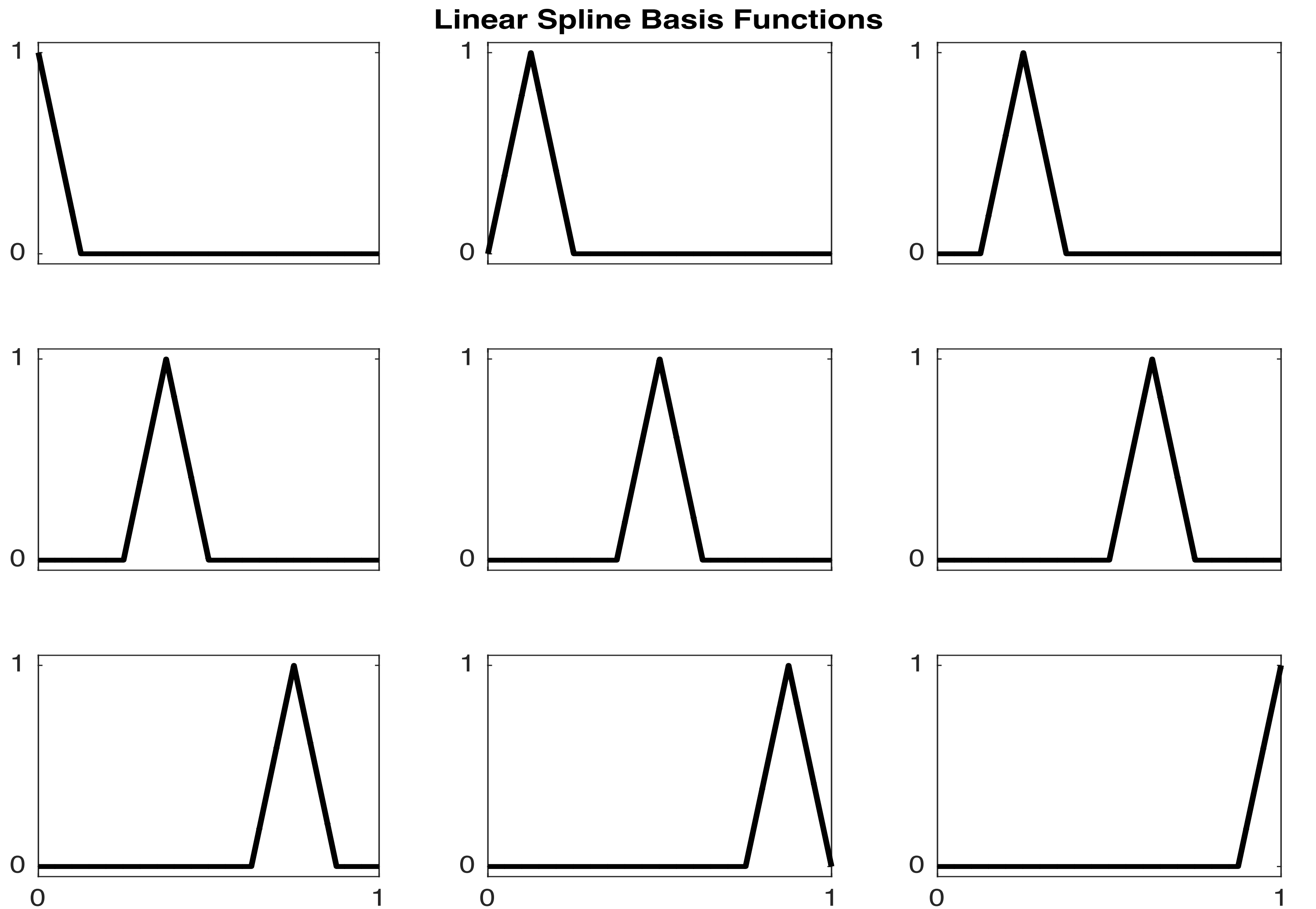
$$t_j = a + (j - 1)h$$

and where h is the distance between any pair of breakpoints

$$h = \frac{b - 1}{n - 1}$$

- In practice choose interpolation nodes $x_1, \ldots, x_n$ to coincide with breakpoints $t_1, \ldots, t_n$ so that $\phi_j(x_i) = 1$ if $i = j$ and zero otherwise

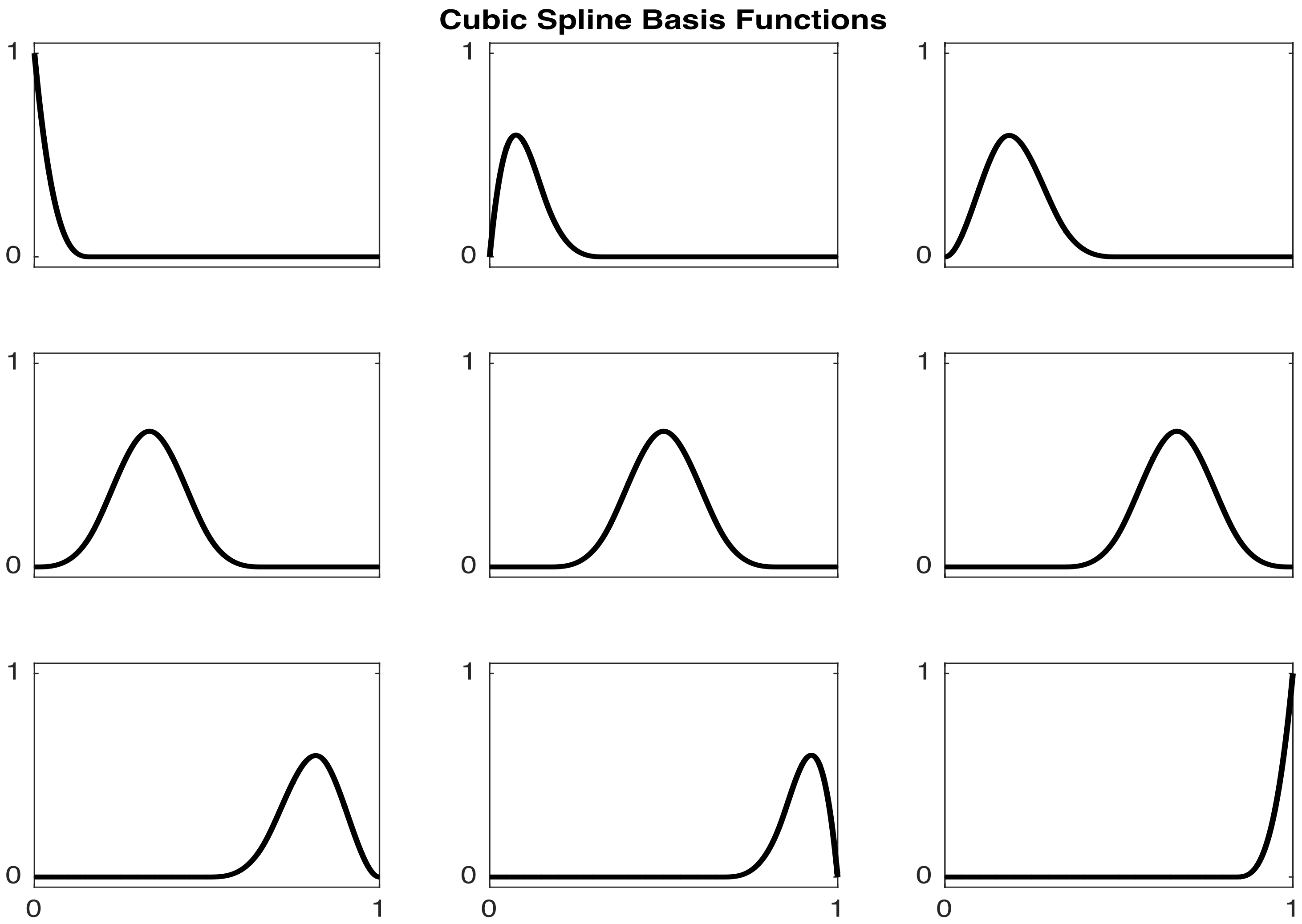
Linear splines



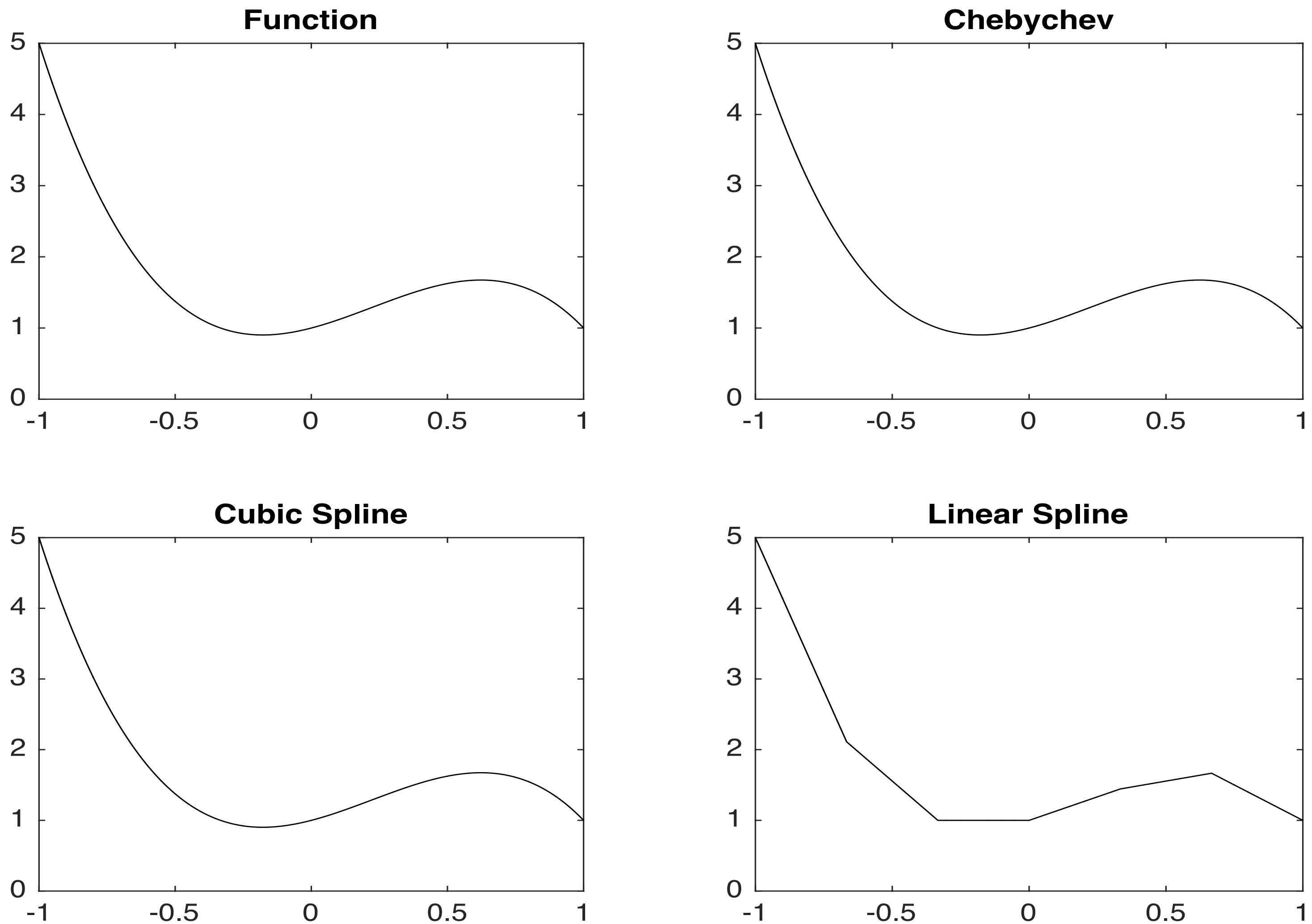
Cubic splines

- Linear splines are a series of line segments spliced together to form a continuous function
- Cubic splines are a series of cubic polynomials spliced together to form a twice continuously differentiable function
- Cubic splines a natural basis for approximating smooth functions

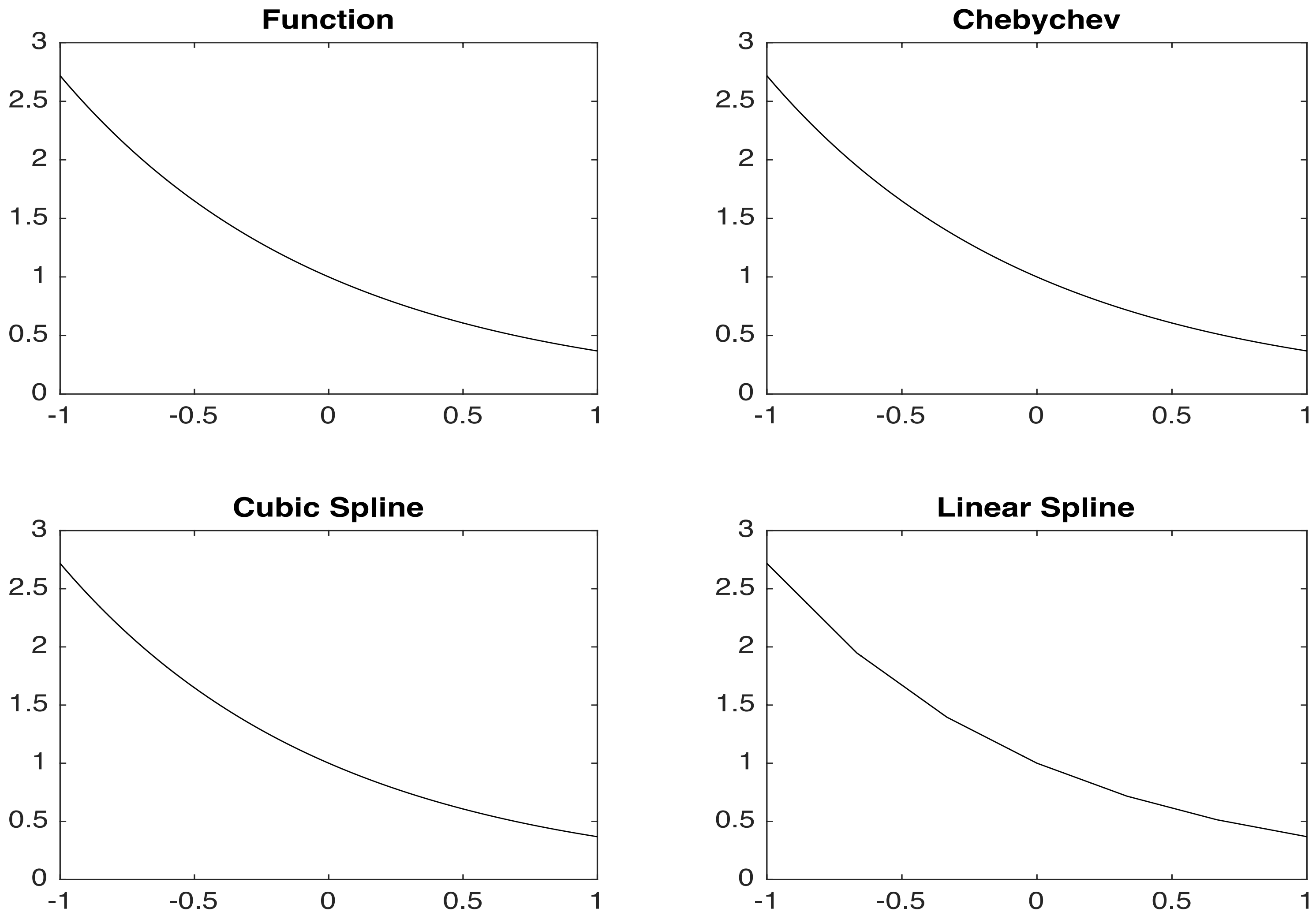
Cubic splines



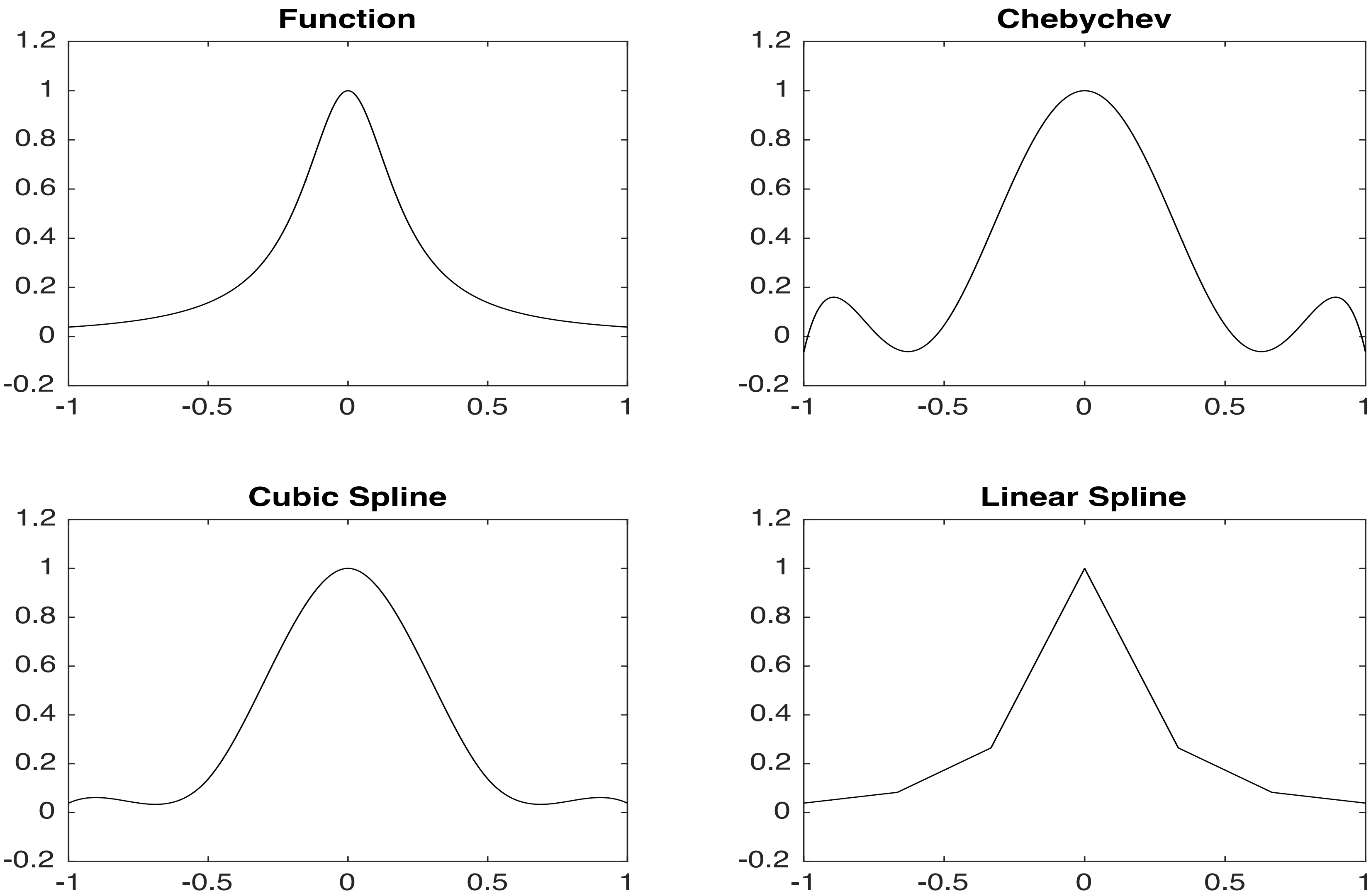
Approximating $f(x) = 1 + x + 2x^2 - 3x^3$



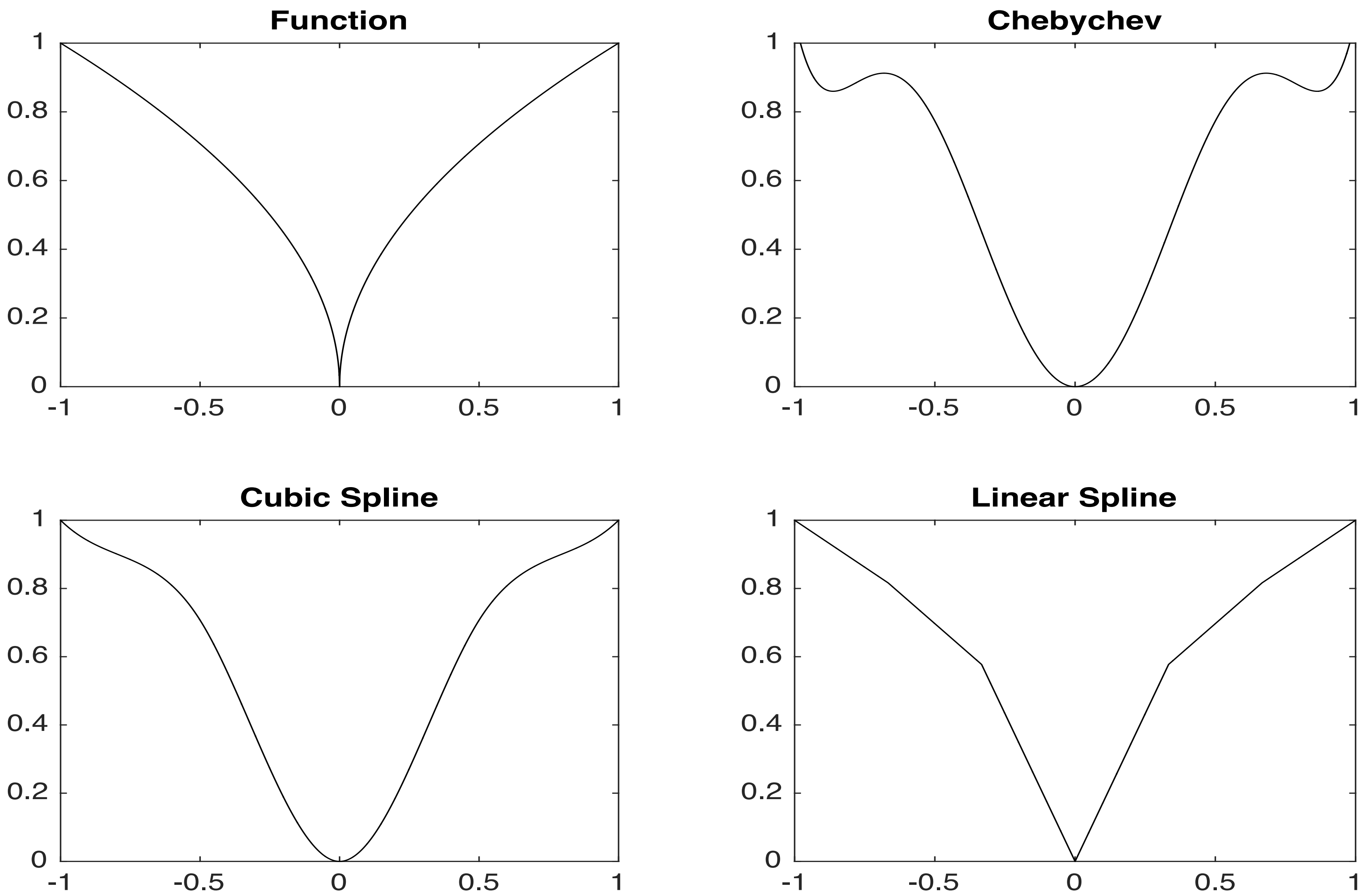
Approximating $f(x) = \exp(-x)$



Approximating $f(x) = 1/(1 + 25x^2)$



Approximating $f(x) = \sqrt{|x|}$



CompEcon toolkit

http://www4.ncsu.edu/~pfackler/compecon/toolbox.html

- Contains many computational tools, including for collocation

- For example

```
fspace = fundef({'spli', breaks})
```

- This creates a function structure using piecewise polynomial splines for basis functions on the breakpoints in `breaks`

- By default `'spli'` uses cubic splines

- Subtle differences in syntax, depending on basis family

CompEcon toolkit

- Grid of evaluation nodes

```
grid = funnode(fspace)
```

- Collocation matrix

```
Phi = funbas(fspace)
```

- Interpolation coefficients for function $y = f(x)$ on `fspace`

```
a = funfitxy(fspace,x,y)
```

Collocation example

Uses Matlab files in "*collocation_example.zip*" in LMS

```
%%%%% economic parameters

alpha = 1/3;         %% capital's share in production function
beta  = 0.95;        %% time discount factor
delta = 0.05;        %% depreciation rate
sigma = 1;           %% CRRA (=1/IES)
rho   = (1/beta)-1; %% implied rate of time preference

kstar = (alpha/(rho+delta))^(1/(1-alpha)); %% steady state
kbar  = (1/delta)^(1/(1-alpha));
```

Parameter structure

```
%%%%% put in a structure to pass to other functions

parameters.alpha = alpha;
parameters.beta  = beta;
parameters.delta = delta;
parameters.sigma = sigma;
```

Breakpoints for splines

```
%%%%% set up grid of capital stock

n          = 99;          %% number of breakpoints for k grid
kmin       = tol;         %% effectively zero
kmax       = kbar;        %% effective upper bound

curv       = 0.5;         %% (curv = 0 log-spaced, curv = 1 linear)
breaks     = nodeunif(n, kmin.^curv, kmax.^curv).^(1/curv);
```

Breakpoints not evenly spaced. Will be $n + 2$ collocation nodes

Function space for approximations

```
%%%%% setup state space using CompEcon tools

fspace   = fundef({'spli', breaks}); %% function space structure


grid     = funnode(fspace); % nodes where we solve the problem
Phi      = funbas(fspace);  % matrix of collocation basis vectors
                            % Phi_{ij} = phi_j(k_i)

k        = grid;            % grid has n+2 elements
```

Initial guess at collocation coefficients

```
%%%%% initial guess at collocation coefficients "a"

c     = alpha*beta*k.^alpha; % guess for consumption policy
v     = log(c)/(1-beta);     % guess for value function

a     = Phi\v;               % implied collocation coefficients
```

Solve Bellman equation by collocation

```
%%%%% solve Bellman equation

for i=1:max_iter;

%%%%% optimal consumption given coefficients "a"

c = solve_brent('rhs_bellman',k,parameters,a,fspace,cmin,cmax,tol)

%%%%% maximized rhs of Bellman equation

v = rhs_bellman(c,k,parameters,a,fspace); %% v(a)
```

Numerical routine `solve_brent` does the maximization

RHS of the Bellman equation

```
function y = rhs_bellman(c,s,parameters,a,fspace)

beta  = parameters.beta;
sigma = parameters.sigma;

u  = utility(c,sigma);

Ev = expected_value(c,s,parameters,a,fspace);

y  = u+beta*Ev;
```

Utility function

```
function u = utility(c,sigma)

if sigma==1,

    u = log(c);

else

    u = (1/(1-sigma))*(c.^(1-sigma) - 1);

end
```

Evaluating $v(k') = \sum_j a_j \phi_j(k')$

```
function v = value(c,k,parameters,a,fspace)

alpha = parameters.alpha;
delta = parameters.delta;

kprime = (k.^alpha) + (1-delta)*k-c;

v = funeval(a,fspace,kprime);
```

Basis function $\phi_j(\cdot)$ evaluated at some k' not nodes k_i

Updating coefficients

```
%%%%% updated collocation coefficients

if do_newton == 1,

%%%%% implied by optimal consumption
kprime    = (k.^alpha) + (1-delta)*k-c;

%%%%% Jacobian matrix of v(a)
Jacobian = beta*funbas(fspace,kprime);

%%%%% Newton's method
anew      = a - (Phi-Jacobian)\(Phi*a-v);

else

anew      = Phi\v;

end
```

Check if converged

```
%%%%% check if converged

error = norm(anew-a,inf);

fprintf('%4i %6.2e \n',[i, error]);

if error<tol, break, end;

%%%%% if not converged, update and try again

a = anew;

end
```

Interpolation on finer grid

```
c_coeff = funfitxy(fspace,s,c);        %% for c(k)
g_coeff = funfitxy(fspace,s,kprime);   %% for g(k)=k'

%%%%% interpolate on finer grid

%%%%% finer grid
kfine = nodeunif(N, kmin.^curv, kmax.^curv).^(1/curv);

%%%%% interpolated value function
vfine = funeval(a,fspace,kfine);

%%%%% interpolation coefficients for c(k)
ac    = funfitxy(fspace,k,c);
```

With solution in hand, can interpolate as needed

Next class

- Stochastic dynamic programming