Macroeconomics III
Lecture 3: Global approximation and Interpolation

Tiago Cavalcanti

FGV/EESP
São Paulo

Road Map: Root Finding

1. Global approximation;
2. Interpolation;
3. Hybrid root finding method (fsolve).

Reading: Judd (1998, ch 6)

The goal

- Suppose: $y = g(x)$ but $g(\cdot)$ is unknown. What is observed is a collection of points in R^2, $D = \{(x_0, y_0), ..., (x_n, y_n)\}$ where $y_i = g(x_i)$, $x_i \neq x_j \; \forall i \neq j$.
- Task: Find a function $\hat{g}(x)$ that approximates $g(x)$ as closely as possible.
- Broadly types of global approximations:
  - **Regression**: some information for $g(x)$ is known, giving us n facts to pin down $m < n$ free parameters
  - **Interpolation**: some information for $g(x)$ is known, giving us n facts to pin down n free parameters
- Comparison:

	info known	points x_i
Regression	$g(x_i)$ at $n > m$ points	given by data
Interpolation	$g(x_i)$ at n points	selected

Global approximation

Typically we approximate function $g : [a, b] \to R$ by:

$$\hat{g}(x) = \sum_{j=1}^{m} c_j \phi_j(x), \text{ where}$$

- m is the degree of interpolation.
- ϕ_j basis function;
- c_j basis coefficients.

Regression

- Regression analysis looks for $E[y/x] = \hat{g}(x)$ where $y = g(x) + \varepsilon$.
- It minimizes
$$\min_{\theta} \sum_{i=1}^{n} \left(g\left(\mathbf{x}_i; \theta\right) - y_i\right)^2$$
- The resulting approximate function $\hat{g}(\mathbf{x}_i)$ will be parametrised
- **Note**: One needs a large number of observations to generate a good fit.

Interporlation

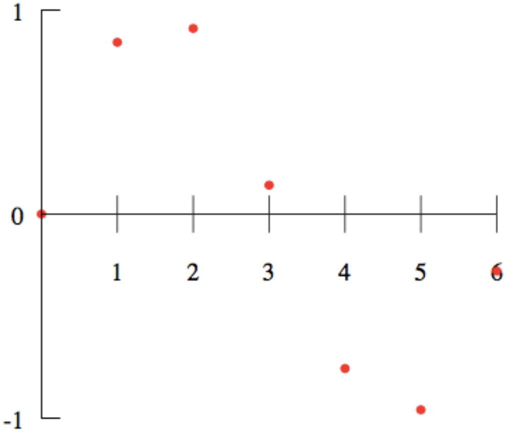
- Interpolation: Construct $\hat{g}$ so that $y_i = \hat{g}(x_i)$ such that the Interpolant and the underlying function must agree at a finite number of points. Additional restrictions may be imposed (first derivatives, smoothness, etc).

- There are many different types of interpolation. Below (main techniques) they are listed by popularity:

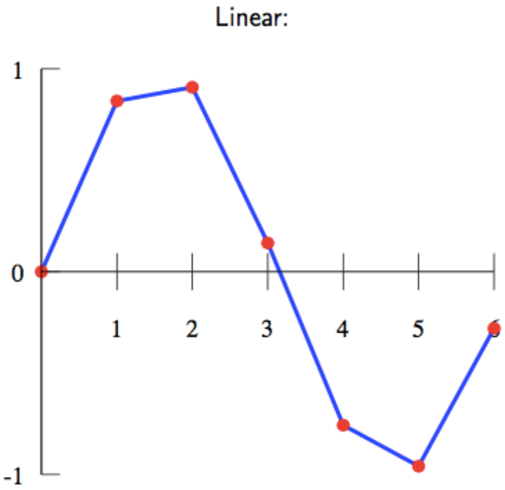
  1. Linear interpolation;
  2. Spline interpolation;
  3. Polynomial interpolation (including Chebyshev interpolation).

Interpolation Examples

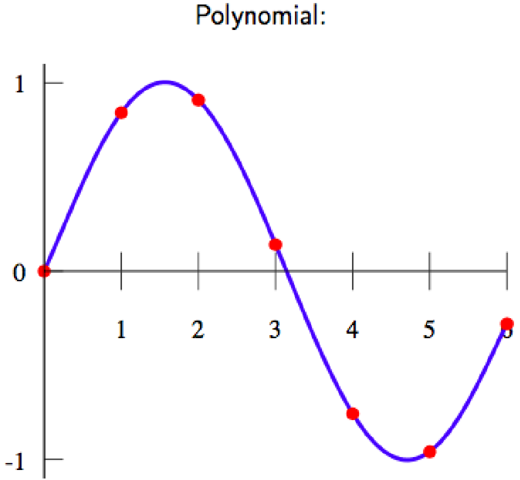
Raw data:



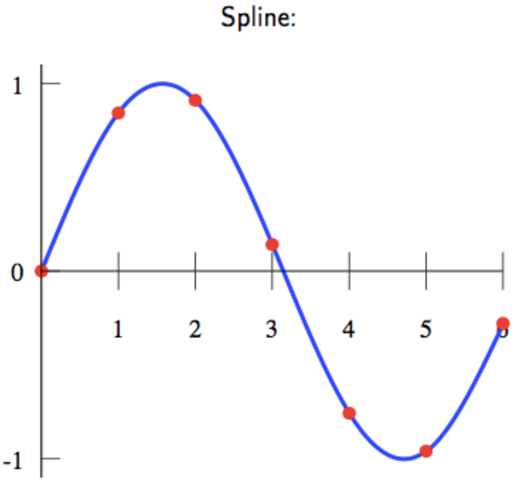
Interpolation Examples



Interpolation Examples



Interpolation Examples



- Spline and polynomial interpolations differ from each other (details below)

Piecewise Linear Interpolation

- Simplest way to interpolate: Connecting the points;
- Draw straight lines between successive data points;
- Piecewise-linear interpolant $\hat{g}$ is given by:

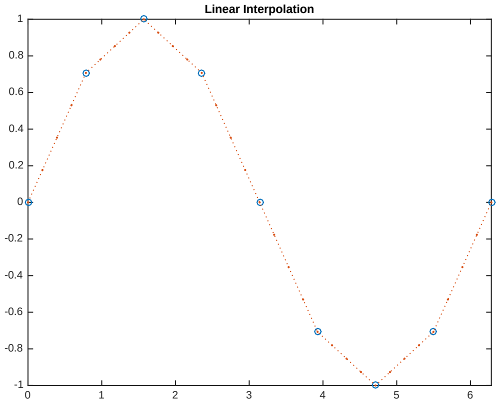
$$\hat{g}(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) \text{ for } x \in [x_i, x_{i+1}].$$

- Kindergaten procedure of "connecting the dots".

Matlab has the built-in function iterp1.

- vq = interp1(x,v,xq) returns interpolated values at specific query points using linear interpolation.
  - Vector x contains the sample points;
  - Vector v contains the corresponding values, $v(x)$.
  - Vector xq contains the coordinates of the query points.
- Example:

```
% Generates a linear approximation of a sine function
x = 0:pi/4:2*pi; % values for x
v = sin(x); % Corresponding values of y
xq = 0:pi/16:2*pi; % Coordinates of the query points.
                  % It can be the same as x or finer
vq = interp1(x,v,xq); % Linear Interpolation
plot(x,v,'o',xq,vq,':.') % plot the interpolation
%xlim([0 2*pi]);
%title('Linear Interpolation');
```


Linear Interpolation
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6

Linear Interpolation

While humble, linear interpolation has significant advantages which account for its popularity.

- **Advantages**:
  - Very fast.
  - Extremely simple.
  - Preserves weak concavity and monotonicity.
  - Easily generalizes to multi-dimensional case.
- **Disadvantages**
  - Does not preserve differentiability (not a smooth function)

Spline interpolation

- The biggest deficiency of linear interpolation is that it produces a non-differentiable interpolant.
- Splines are a different way to interpolate that can produce an interpolant that is continuously differentiable up to a given order.
- In particular, they take piecewise-polynomials within each interval $[x_{i-1}, x_i]$.

Splines

- **Definition**: A **spline** of order n is a piece-wise polynomial real function $s : [a, b] \to \mathbb{R}$, with $s \in C^{n-2}$ when there is a grid of nodes $a = x_0 < x_1 < ... < x_m = b$ such that $s(x)$ is a polynomial of degree $n - 1$ on each subinterval $[x_{j-1}, x_j], j = 1, ..., m$.

- **Example**: A spline of order 2 gives the standard piece-wise linear interpolation, i.e. given grid points and information about the function at these points, fit straight lines that connect the dots.

This is different from polynomial interpolation which uses one high-order polynomial for the entire domain $[a, b]$.

Spline interpolation

Spline of different orders go by different names:

- **Linear spline/interpolant (order 2):** linear interpolant (a spline of order 2 since it is C^0 globally and uses polynomials of degree 1 (linear) in each interval);
- **Quadratic spline (order 3):** each interval is a parabola and the entire spline is C^1.
- **Cubic spline (order 4):** each interval is a cubic and the entire spline is C^2 - most popular.

Cubic Splines

- Let's focus on cubic splines (others follow similar procedure).
- On each interval $[x_{i-1}, x_i]$ the spline is a cubic:

$$s(x) = a_i + b_i x + c_i x^2 + d_i x^3, \; x \in [x_{i-1}, x_i].$$

- For each interval i there is a separate set of coefficients (a_i, b_i, c_i, d_i).
- In total there are $n + 1$ data points, n intervals and $4n$ unknown coefficients.
- Task: How do we determine the coefficients?

Conditions

- **Condition 1:** $s(x_i) = g(x_i) = y_i$,

$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3,\ i = 1, ..., n$$

- **Condition 2:** The polynomial pieces must connect

$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3,\ i = 0, ..., n-1$$

- **Condition 3:** First and second derivative have to agree at x_i

$$\begin{aligned} b_i + 2c_i x_i + 3d_i x_i^2 &= b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2,\ i = 1, ..., n-1 \\ 2c_i + 6d_i x_i &= 2c_i + 6d_i x_{i+1},\ i = 1, ..., n-1 \end{aligned}$$

Conditions

- **Condition 1:** $s(x_i) = g(x_i) = y_i$,

$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3,\ i = 1, ..., n$$

- **Condition 2:** The polynomial pieces must connect

$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3,\ i = 0, ..., n-1$$

- **Condition 3:** First and second derivative have to agree at x_i

$$\begin{aligned} b_i + 2c_i x_i + 3d_i x_i^2 &= b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2,\ i = 1, ..., n-1 \\ 2c_i + 6d_i x_i &= 2c_i + 6d_i x_{i+1},\ i = 1, ..., n-1 \end{aligned}$$

Conditions

- **Condition 1:** $s(x_i) = g(x_i) = y_i$,

$$y_i = a_i + b_i x_i + c_i x_i^2 + d_i x_i^3,\ i = 1, ..., n$$

- **Condition 2:** The polynomial pieces must connect

$$y_i = a_{i+1} + b_{i+1} x_i + c_{i+1} x_i^2 + d_{i+1} x_i^3,\ i = 0, ..., n-1$$

- **Condition 3:** First and second derivative have to agree at x_i

$$\begin{aligned} b_i + 2c_i x_i + 3d_i x_i^2 &= b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2,\ i = 1, ..., n-1 \\ 2c_i + 6d_i x_i &= 2c_i + 6d_i x_{i+1},\ i = 1, ..., n-1 \end{aligned}$$

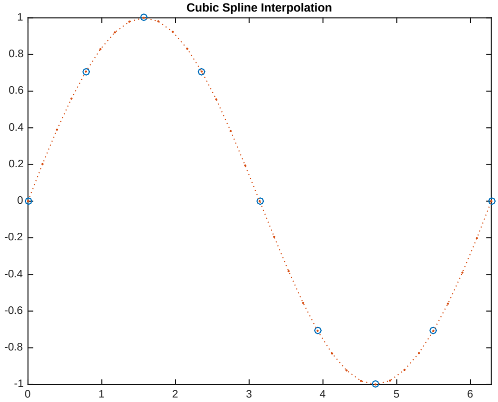
Conditions

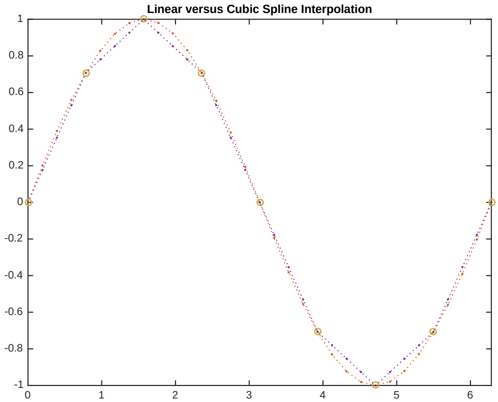
- There are $4n-2$ linear equations in $4n$ *unknowns* a, b, c, d.
- Two more conditions are needed:
  - It is common to use $s'(x_0) = 0 = s'(x_n)$ (choice should depend on problem)

Matlab has the built-in function spline.

- yy = spline(x,Y,xx). Cubic spline interpolation to find yy, the values of the underlying function Y at the values of the interpolant xx. For the interpolation, the independent variable is assumed to be the final dimension of Y with the breakpoints defined by x.
- Example:

```
% Generates a linear approximation of a sine function
x = 0:pi/4:2*pi; % values for x
v = sin(x); % Corresponding values of y
xq = 0:pi/16:2*pi; % Coordinates of the query points.
                  % It can be the same as x or finer
vq = spline(x,v,xq); % Linear Interpolation
plot(x,v,'o',xq,vq,':.') % plot the interpolation
xlim([0 2*pi]);
title('Cubic Spline Interpolation');
```


Cubic Spline Interpolation
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6


Linear versus Cubic Spline Interpolation
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6

Polynomial Interpolation

- Suppose we want to approximate a function $f : [a, b] \to \mathbb{R}$, where $f \in C[a, b]$ (continuous functions on $[a, b]$).
- The (vector) space of continuous functions on $[a, b]$ is spanned by all monomials, i.e. a basis for $C[a, b]$ is

$$\{1,\ x,\ x^2,\ \ldots, x^i, \ldots\}$$

- We can then generate approximations

$$\hat{f}(x) = \sum_{i=0}^{n} \theta_i x^i$$

- Or more generally

$$\hat{f}(x) = \sum_{i=0}^{n} \theta_i \phi_i(x)$$

  where $\{\phi_1(x), \phi_2(x), \ldots, \phi_n(x)\}$ is a subset from a family of polynomials that form a basis of $C[a, b]$.

Least Squares Method

- For an unknown function $f : [a, b] \to \mathbb{R}$, where $f \in C\,[a, b]$, determine polynomials $\phi_j\,(x)$ and then find coefficients $\theta = \{\theta_j\}_{j=1}^{n}$ such that:

$$\min \theta_i \sum_{i=1}^{n} \left(f(x_i) - \sum_{j=1}^{m} \theta_i \phi_j(x_i) \right)^2, \; m < n.$$

- Solution: $\theta = (\Phi' \Phi)^{-1} \Phi' y$.

- How to choose $\phi_j(x)$?

Which polynomials?

- A natural family is the **monomial basis** $\{1, x, x^2, ..., x^m\}$
- **monomial basis** is not in general good (most of the times they are not) because for large numbers x, consecutive powers are close to each other and they may generate unreliable results, due to multicollinearity.
- Families of orthogonal polynomials do a good job at avoiding this problem (Legendre Polynomials, Chebyshev Polynomials, Laguerre Polynomials, Hermite Polynomials)

- **Definition**: The family of functions $\{\phi_1(x), \phi_2(x), \ldots\}$ is **orthogonal** with respect to weight $\omega(x)$ if

$$\int_a^b \omega(x)\phi_i(x)\,\phi_j(x) = 0, \text{ for } i \neq j$$

Chebyshev polynomials

- **Definition**: The nth **Chebyshev polynomial** on $[-1, 1]$ is defined recursively:

  $T_0(x) = 1,\ T_1(x) = x,\ T_2(x) = 2x^2 - 1, T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x).$

- The weight function is:

$$\omega(x) = (1 - x^2)^{-\frac{1}{2}}.$$

- One can also define $T_n(x) = \cos(n \arccos(x))$.
- Indeed:

$$\int_{-1}^{1} \frac{T_i\,(x)\,T_j\,(x)}{\sqrt{1 - x^2}} dx = \begin{cases} 0 & \text{if } i \neq j \\ \pi & \text{if } i = j = 0 \\ \frac{\pi}{2} & \text{if } i = j \geq 1 \end{cases}$$

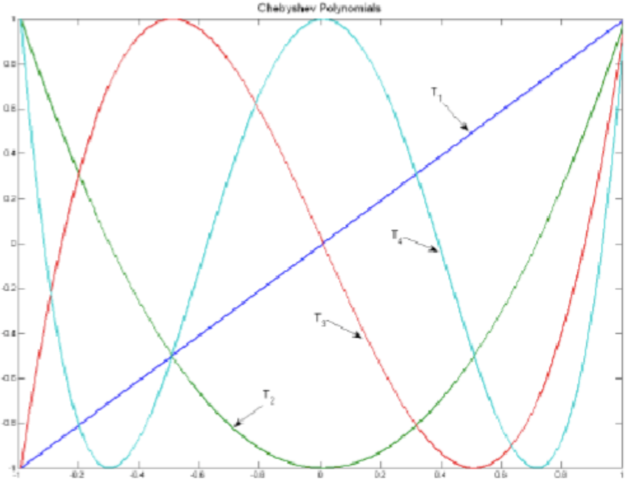
Chebyshev polynomials

- Some useful properties of Chebyshev polynomials, for constructing the function approximations
- **Range**: $T_n(-1) = -1$, $T_n(1) = 1$ and $T_n(z) \in [-1, 1]$.
- **Extrema**: $T_n(x)$ has $n+1$ extrema, equal to -1 or 1.
- **Roots**: $T_n(x)$ has n distinct roots in $[-1, 1]$, given by

$$x_i = -\cos\left(\frac{(2i-1)\pi}{2n}\right)$$

- **Discrete orthogonality**: If $\{x_i\}_{i=1}^{n}$ are the roots of a Chebyshev polynomial of order n, then

$$\sum_{k=1}^{n} T_i(x_k)\, T_j(x_k) = \begin{cases} 0 & \text{if } i \neq j \\ n & \text{if } i = j = 0 \\ \frac{n}{2} & \text{if } i = j \geq 1 \end{cases}$$


Chebyshev Polynomials
T_1
T_2
T_3
T_4

Chebyshev polynomials

- We usually want to approximate functions in a general interval $[a, b]$ instead of $[-1, 1]$, we can do the following transformation:

$$\begin{aligned} z &= h(x) = \frac{2(x-a)}{b-a} - 1, x \in [a, b], \\ x &= \frac{(z+1)(b-a)}{2} + a, z \in [-1, 1] \end{aligned}$$

- The **general Chebyshev polynomials** order n are defined as $\tilde{T}_n : [a, b] \to \mathbb{R}$ with

$$\tilde{T}_n(x) = T_n(h(x)) = T_n\left(\frac{2(x-a)}{b-a} - 1\right)$$

- The weight function for which general Chebyshev polynomials become orthogonal is

$$\omega(x) = \frac{1}{\sqrt{1 - \left(\frac{2x-a-b}{b-a}\right)^2}}$$

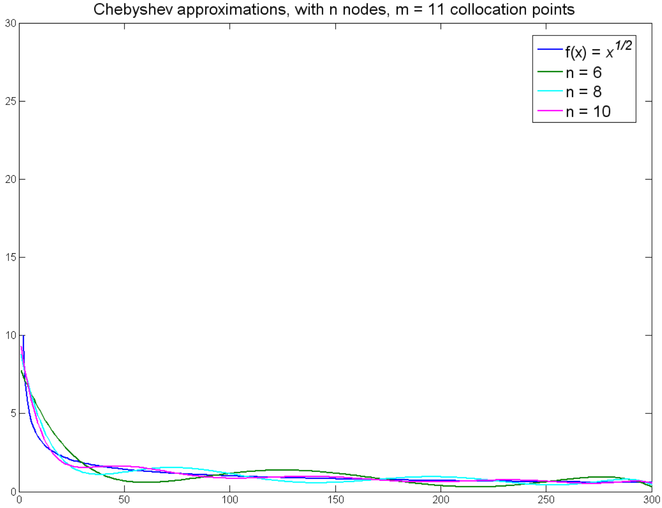
Which coefficients?

- Recall we are looking for an approximation of a function $g : [a, b] \to \mathbb{R}$ with Chebyshev polynomials of order up to n:

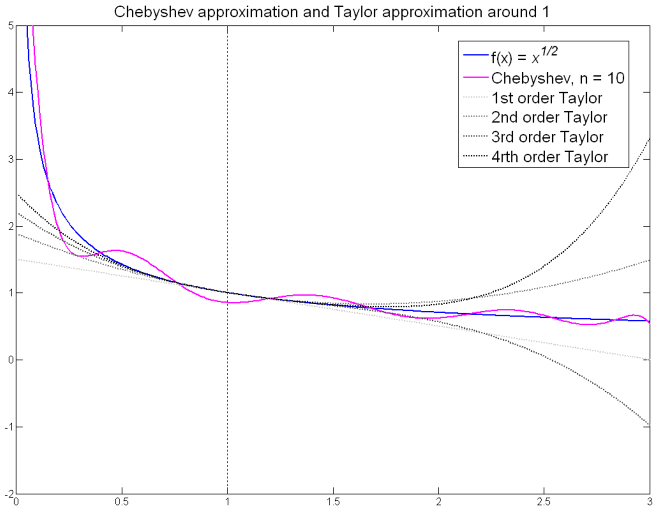
$$\hat{g}(x) = \sum_{j=0}^{m} \theta_j \tilde{T}_j(x)$$

- The problem is how to choose $n + 1$ coefficients $\theta = \{\theta_j\}_{j=0}^{n}$.
- We calculate appropriate coefficients $\{\theta_j\}_{j=0}^{n}$ using known information (or 'guessed' information) about the function g at well chosen **nodes** (or **collocation points**, or **grid points**) in $[a, b]$.
- If # nodes = $n = m + 1$, then the method is **interpolation**
- If # nodes = $n > m + 1$, then the method is **regression**

Example: Approximation of $f\left(x^{-1/2}\right)$.



Comparison:



Chebyshev approximation and Taylor approximation around 1
f(x) = x^{1/2}
Chebyshev, n = 10
1st order Taylor
2nd order Taylor
3rd order Taylor
4rth order Taylor
5
4
3
2
1
0
-1
-2
0
0.5
1
1.5
2
2.5
3

Powells Hybrid Method to Find the Root - fsolve

- Note that Newton's method applied to $f(x) = 0$
  - May not converge
  - Converges rapidily (if it converges)
- On the other hand, a minimization of $SSR(x) = \sum_i f_i(x)^2$
  - Always converges to something;
  - May converge to a local *min*
  - May converge slowly

Powell's Hybrid Method to Find the Root - fsolve

- Powell's method checks if a Newton step reduces the value of SSR.
  - Suppose that the current guess is x_k, the Newton step is
    $$x_{k+1} = x_k + s_k = x_k - J(x_k)^{-1} f(x_k).$$
  - Powell method checks if $SSR(x_k + s_k) < SSR(x_k)$.
  - Formally:
    $$\min_{\lambda} SSR(x) \text{ subject to } x = x_k + \lambda s_k.$$
- Newton's method just sets $\lambda = 1$ and throws caution to the wind.
- This procedure always "works" in that it will converge to some x that has weakly smaller SSR than your original point. However, there is no guarantee of finding a root.

Matlab has the built-in function fsolve, which uses the Powell's Hybrid Method.

- x = fsolve(fun,x0) starts at $x0$ and tries to solve the equations $fun(x) = 0$, an array of zeros.
- Matlab calls this trust region dogleg.

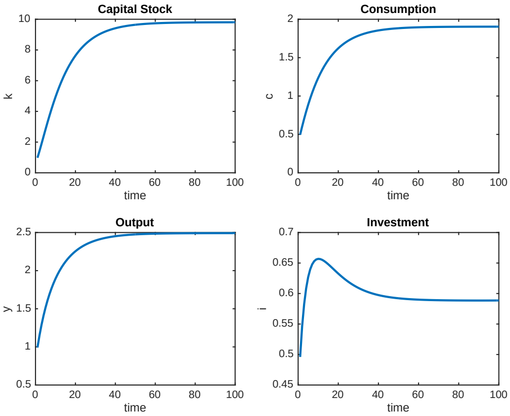
Program to use MatLab fsolve function.

```
% fsolve
%   f(x) = 0
%
% where x is a vector of unknowns and f is a function
% vector. Our system of equations is
%
%   2*x(1) -   x(2) - exp(-x(1)) = 0
%    -x(1) + 2*x(2) - exp(-x(2)) = 0.
%

% Left-hand side of our system of equations:
myfun = @(x) [2*x(1) -   x(2) - exp(-x(1)); ...
               -x(1) + 2*x(2) - exp(-x(2))];
% Makte a starting guess at the solution
x0 = [-5; -5];
% Set option to display information after each iteration
options=optimset('Display','iter');
% Solve the system
[x,fval,exitflag] = fsolve(myfun,x0,options)
```

```
% growthsolve.m This program solves the growth model
% Time period utility: u(c)=(c^(1-eta)-1)/(1-eta)
% Technology: c(t)+k(t+1)-(1-delta)k(t)=Ak(t)^alpha
%             given k(0) This is solved with fsolve
%  PARAMETER VALUES
A=1;         % TFP
alpha=0.4; % Capital share
delta=0.06; % depreciation rate
eta=0.99;   % CRRA coeffficient
beta=0.96;  % Subjective discount factor
T=100;      % Periods for transition
% STEADY-STEATE and INITIAL CAPITAL STOCK
kss=((A*beta*alpha)/(1-(1-delta)*beta))^(1/(1-alpha));
k0=0.10*kss;
% seed
for j=1:T;
    x0(j)=k0*(1-j/T)+j/T*kss; % libear convex combination of k0 and kss
end
x0=x0';
param= [T A alpha delta eta k0 beta kss]';
options=optimoptions('fsolve','Display','iter');
sol=fsolve(@(z)focg(z,param),x0,options);
```

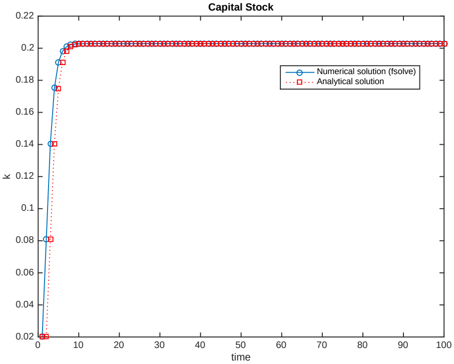
```
function f=focg(z,p)
% Vector of parameters p (defined in the main file)
T=p(1);
A=p(2);
alpha=p(3);
delta=p(4);
eta=p(5);
k0=p(6);
beta=p(7);
kss=p(8);
for t=1:T
    k(t)=z(t); % ENDOGENOUS VARIABLE z
end
k(T+1)=kss;
f(1)=beta*(A*k(1)^alpha+(1-delta)*k(1)-k(2))^(-eta)*(
    alpha*A*k(1)^(alpha-1)+(1-delta))-(A*k0^alpha+(1-
    delta)*k0-k(1))^(-eta);
for t=2:T
    f(t)=beta*(A*k(t)^alpha+(1-delta)*k(t)-k(t+1))^(-eta
        )*(alpha*A*k(t)^(alpha-1)+(1-delta))-(A*k(t-1)^
        alpha+(1-delta)*k(t-1)-k(t))^(-eta);
end
f=f';
```



Capital Stock
k
time
Consumption
c
time
Output
y
time
Investment
i
time

Comparison: Numerical vs Analytical

Assume: η very close to one, $\delta = 1$, then: $k_{t+1} = \beta\alpha A k(t)^{\alpha}$.



Minimum of constrained nonlinear multivariable function - fmincon

- An alternative to fsolve, which finds the zero of the system is to minimize the system.
- This can be useful for constrained problems.
- $x = fmincon(fun, x0, A, b)$ starts at $x0$ and attempts to find a minimizer x of the function described in ‘fun’ subject to the linear inequalities $A * x \leq b$. x0 can be a scalar, vector, or matrix.