Topics in Macroeconomics

Lecture 2: Root Finding and Function Approximation

Diego de Sousa Rodrigues
de_sousa_rodrigues.diego@uqam.ca

ESG-UQAM

Fall 2025

Road Map

Root Finding:

1. Bisection Method.
2. Local approximations:
   - Newton–Raphson (1D and multi-D).
   - Secant.

Function Approximation:

- Local vs. Global.
- Taylor expansions.
- Chebyshev & splines (overview).

Root-Finding vs Fixed-Point

Two common problems:

$$f(x) = 0, \quad f : \mathbb{R}^n \to \mathbb{R}^n, \qquad g(x) = x, \quad g : \mathbb{R}^n \to \mathbb{R}^n.$$

The former is a **root-finding** problem, the latter a **fixed-point** problem. They are equivalent:

$$f = 0 \iff (f + x) = x, \qquad g = x \iff (g - x) = 0.$$

Analytical solutions often infeasible $\Rightarrow$ use numerical methods.

Root-finding in One Dimension

- We begin with the simplest case: **one-dimensional root-finding.**

$$f(x) = 0, \quad f : \mathbb{R} \to \mathbb{R}.$$

- Provides the foundation for understanding higher-dimensional root-finding methods.

Bisection Method

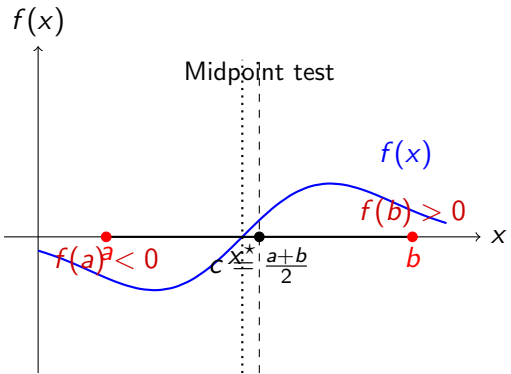
- A **simple and robust method for finding the root of a univariate continuous function** $f(x)$ on a closed interval $[a, b]$.
- Always converges to a solution-if one exists and if the initial interval includes it.
- Does not rely on derivatives $\Rightarrow$ can be used to find roots of non-smooth functions.
- **Basic idea:** If f is continuous and $f(a)$ and $f(b)$ have opposite signs, then

$$\exists\, x^* \in [a, b] \text{ such that } f(x^*) = 0.$$

(Intermediate Value Theorem).

Bisection Method

- Start with $f : [a, b] \to \mathbb{R}$ continuous and $f(a) \cdot f(b) < 0$.
- Compute midpoint $c = \frac{a+b}{2}$ and test the sign of $f(c)$.
- Shrink the interval depending on the sign of $f(c)$; repeat until $|f(c)| \leq \varepsilon$.



Bisection Method: Algorithm

Assume $f(a) < 0$ and $f(b) > 0$.

1. Define **lower and upper bounds** of the interval: use $\underline{x} = a$ and $\bar{x} = b$.
2. Compute the **midpoint of the interval**:

$$c = \frac{\bar{x} + \underline{x}}{2}.$$

3. If $f(c) > 0$, set $\bar{x} = c$; if $f(c) < 0$, set $\underline{x} = c$.
4. If $|f(c)| \leq \varepsilon$, stop and call c a root; otherwise, return to step (ii).

Bisection Method to find the root of $f(x) = x^2 - 4$

```
% Bisection Method. Root of f(x)=x^2-4 for x in [a,b];
a=1;    % Lower bound
b=10;   % Upper bound
fa=a^2-4; % Value at the lower bound
fb=b^2-4; % Value at the upper bound
tol=0.001;
if fa*fb>0 % Condition of the IVT
    disp('Wrong choice for a and b')
else
    c=(a+b)/2; % Midpoint
    err=abs(c^2-4);
    while err>tol
        if c^2-4>0
            b=c;
        elseif c^2-4<0
            a=c;
        end
        c=(a+b)/2;
        err=abs(c^2-4);
    end
end
```

A More Elegant and Efficient Way

- Write a **general `bisection` function in MATLAB** (script/`.m` file).
- **Then call this function to find the root of any univariate $f(x)$** on a bracket $[a, b]$.
- Advantages:
  - Reusable: just pass a different `f`, `a`, `b`.
  - Keeps classroom code tidy (logic lives in one place).
  - Easy to change tolerances/criteria centrally.

bisection.m (Function Script)

```
function c = bisection(f,a,b)
% Simple code to find a root of a univariate function f on [a,b]
% using the bisection method.

if f(a)*f(b) > 0
    error('Wrong choice for a and b');  % IVT requires opposite signs
else
    c   = (a + b)/2;          % Midpoint
    err = abs(f(c));          % Residual at midpoint
    while err > 1e-7          % Tolerance
        if f(a)*f(c) < 0
            b = c;            % Root is in [a, c]
        else
            a = c;            % Root is in [c, b]
        end
        c   = (a + b)/2;      % Update midpoint
        err = abs(f(c));      % Update residual
    end
end
```

Calling the bisection Function

How do you call the function (you need f, a, and b)?

```
>> f = @(x) x.^2 - 4;    % Define the function
>> a = 1;                % Left endpoint
>> b = 10;               % Right endpoint
>> yourroot = bisection(f, a, b)

yourroot =

     2.0000
```

- Define the function handle `f`.
- Provide an initial interval $[a, b]$ with $f(a) \cdot f(b) < 0$.
- Call `bisection(f,a,b)` to obtain the root.

Activity: Using `bisection.m`

Task:

Use your `bisection.m` function to find the root of

$$f(x) = \cos(x) - x$$

on the interval $[0, 1]$.

MATLAB starter code:

```
% Define function handle
f = @(x) cos(x) - x;

% Bracket interval [a,b]
a = 0;
b = 1;

% Call the bisection function
root = bisection(f, a, b);

% Display result
disp(['Root found at x = ', num2str(root)]);
```

Deliverable: Report the approximate root $x^\star$ and check that $f(x^\star) \approx 0$.

Bisection Method: Advantages and Disadvantages

Advantages:

- Works for any continuous (C^0) function with a sign change on $[a, b]$.
- Extremely simple to implement.
- Frequently used as a "safe" method when others fail.

Disadvantages:

- Convergence is slow relative to Newton or Secant methods.
- Does not exploit curvature (derivative) information.
- Error decreases by only ≈ 1 decimal place every 3 iterations.
- Requires a valid initial bracket $[a, b]$ (true of most bracketing methods).

Function Approximation

Classifications:

- **Local approximation methods:**
  - Taylor approximations.
- **Global approximation methods:**
  - Ordinary regression and interpolation.
  - Orthogonal polynomials (e.g., Chebyshev).
  - Chebyshev regression in R and R^2.
  - Splines and shape-preserving approximations.

Local vs Global Approximations

Key idea: Solving macro models computationally usually boils down to approximating unknown functions (e.g., value functions, policy functions).
Getting good approximations for functions (e.g. value functions, policy functions, etc.) that are not known to the modeller and cannot be computed analytically is crucial.

Local vs Global Approximations

Key idea: Solving macro models computationally usually boils down to approximating unknown functions (e.g., value functions, policy functions).
Getting good approximations for functions (e.g. value functions, policy functions, etc.) that are not known to the modeller and cannot be computed analytically is crucial.
Two approaches:

- **Local approximations:**
  - Focus on one point of interest (often the steady state of an economy and approximate locally around that point).
  - Use Taylor expansions around that point.
- **Global approximations:**
  - Approximate the entire function with "nice" functions (e.g., polynomials, splines).
  - Ensure closeness in a well-defined sense (uniform or mean-square).

When to Use Local vs Global Approximations in Economics?

Question: Think about models you have seen in macroeconomics. In which cases would you expect each method to be useful?

When to Use Local vs Global Approximations in Economics?

Question: Think about models you have seen in macroeconomics. In which cases would you expect each method to be useful?

Local approximations (Taylor expansions):

- Around which objects (e.g., steady state, balanced growth path) are they usually applied?
- What kinds of shocks or policy questions make them informative?

Global approximations (Chebyshev, splines, regression):

- When do we need to capture behavior far away from the steady state?
- What kinds of models (e.g., with occasionally binding constraints, large shocks, nonlinearities) require them?

Answers: When Do We Use Each?

Local approximations:

- DSGE models around the steady state.
- Small shocks, small policy changes.
- Linearization or 2nd-order expansions.

Global approximations:

- Models with large shocks or strong nonlinearities.
- Occasionally binding constraints (e.g., zero lower bound, borrowing limits).
- Models with default, inventories, or precautionary savings.

Taylor vs. Weierstrass

The starting point for each method are the following two powerful theorems:

Taylor vs. Weierstrass

The starting point for each method are the following two powerful theorems:

Taylor's Theorem:

- Any sufficiently smooth function can be locally approximated by a polynomial.
- **Advantage:** Exact approximation near the expansion point.
- **Limitation:** Accuracy deteriorates far from the point (e.g., outside the steady state).

Taylor vs. Weierstrass

The starting point for each method are the following two powerful theorems:

Taylor's Theorem:

- Any sufficiently smooth function can be locally approximated by a polynomial.
- **Advantage:** Exact approximation near the expansion point.
- **Limitation:** Accuracy deteriorates far from the point (e.g., outside the steady state).

Weierstrass' Theorem:

- Any continuous function on $[a, b]$ can be approximated uniformly by polynomials.
- **Advantage:** Global approximation guaranteed.
- **Limitation:** Does not tell us which polynomial to pick; may be numerically tricky.

Taylor Approximations

Univariate: For $f : \mathbb{R} \to \mathbb{R}$ where $f \in C^{n+1}$ and $x^* \in \mathbb{R}$, the n-th order Taylor approximation is

$$f(x) \approx f(x^*) + (x - x^*) f^{(1)}(x^*) + \frac{1}{2}(x - x^*)^2 f^{(2)}(x^*) + \cdots + \frac{1}{n!}(x - x^*)^n f^{(n)}(x^*) + e_{n+1},$$

$$\text{where } e_{n+1} \to 0 \text{ as } n \to \infty.$$

Multivariate: For $f : \mathbb{R}^m \to \mathbb{R}$ where $f \in C^{n+1}$ and $x^* \in \mathbb{R}^m$, the n-th order Taylor approximation is

$$f(x) \approx f(x^*) + \sum_{i=1}^{m} \frac{\partial f(x^*)}{\partial x_i}(x_i - x_i^*) + \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \frac{\partial^2 f(x^*)}{\partial x_i \partial x_j}(x_i - x_i^*)(x_j - x_j^*) + \dots$$

$$+ \frac{1}{n!} \sum_{i_1=1}^{m} \cdots \sum_{i_n=1}^{m} \left(\frac{\partial^n f(x^*)}{\partial x_{i_1} \cdots \partial x_{i_n}} \prod_{k=1}^{n} (x_{i_k} - x_{i_k}^*) \right) + e_{n+1}, \quad e_{n+1} \to 0 \text{ as } n \to \infty.$$

Local Approximations and Their Limits

- A **local approximation is accurate only near the expansion point** x^*.
- **Accuracy can deteriorate rapidly** once we move away from x^*.
- Useful theorem: if f (or some derivative) has a singularity at y, then the Taylor expansion around x^* is reliable only within

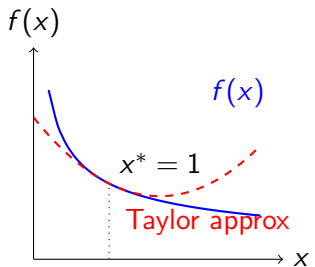
$$[\,x^* - d,\; x^* + d\,], \quad d = |x^* - y|.$$

- **Example:** $f(x) = x^{-1/2}$ has a singularity at $x = 0$. If we approximate around $x^* = 1$, the expansion is valid only for $x < 2$.

Example: Taylor Approximation with Singularity

Function: $f(x) = x^{-1/2}$, singularity at $x = 0$, expansion around $x^* = 1$.

- The Taylor expansion provides a good local approximation near $x = 1$.
- Accuracy deteriorates as $x \to 0$ or $x > 2$.



Two-Period Saving Problem

Example:

$$\max_{a_1,\, c_1,\, c_2} \{u(c_1) + \beta u(c_2)\} \quad \text{s.t.} \quad c_1 + a_1 = m, \quad c_2 = (1+r)a_1.$$

Equivalently:

$$\max_{a_1} \big\{u(m - a_1) + \beta u((1+r)a_1)\big\}.$$

First-order conditions:

$$-u'(m - a_1) + \beta(1+r)\, u'((1+r)a_1) = 0,$$

$$c_1 + a_1 = m, \qquad c_2 = (1+r)a_1.$$

Goal:

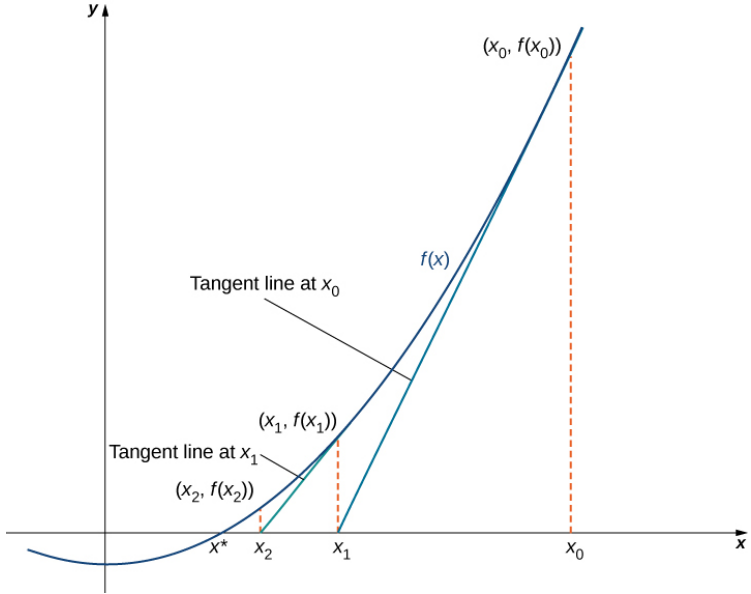
- Find the policy function $a_1 = g(m)$ or $c_1 = h(m) = m - g(m)$.
- First step: solve the 1D problem given by the Euler equation.

Next: Another Method to Find Zeros

We now consider a **local, derivative-based** method:

- Uses a first-order Taylor approximation of f,
- Updates via slope information at the current guess,
- Typically converges faster than bisection near the root.

Newton's Method



Newton–Raphson (one dimension)

- Suppose you need to solve $f(x) = 0$ and $f : \mathbb{R} \to \mathbb{R}$.
- Start at x_0 ($f(x_0)$ must be defined).
- Take the first-order Taylor approximation of f around x_0:

$$f(x) \approx v(x) = f(x_0) + f'(x_0)(x - x_0).$$

- Find x_1 which solves the zero of v implying:

$$v(x_1) = 0 \quad \Rightarrow \quad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

- Check if $f(x_1)$ is defined (if not, choose a point between x_0 and x_1). If yes, repeat the procedure at x_1, such that:

$$x_{s+1} = x_s - \frac{f(x_s)}{f'(x_s)}.$$

- The solution x^* is found if $f(x_{s+1}) \approx 0$.

Newton–Raphson Method (one dimension)

Suppose we apply it to our two period saving problem.

We need:

$$f(x) = -u'(m-a_1)+\beta(1+r)\,u'\big((1+r)a_1\big), \qquad f'(x) = u''(m-a_1)+\beta(1+r)^2\,u''\big((1+r)a_1\big).$$

For a given m choose an initial condition $a_{1,0}$ and compute:

$$a_{1,s+1} = a_{1,s} - \frac{f(a_{1,s})}{f'(a_{1,s})} \quad \text{for } s = 0, 1, \ldots, S$$

until

$$|f(a_{1,s+1})| < \delta.$$

Example 1: Newton–Raphson for $y = (x-4)(x+4)$

We want to find the zeros of

$$y = (x-4)(x+4).$$

MATLAB code (newton.m):

```
function x = newton(func, x0, param, crit, maxit)
% Newton.m Program to solve a system of equations
% Usage: x = newton(func, x0, param, crit, maxit)

for i = 1:maxit
    [f,J] = feval(func, x0, param);
    x = x0 - inv(J)*f;
    if norm(x - x0) < crit
        break
    end
    x0 = x;
end

if i >= maxit
    sprintf('WARNING: Maximum number of %g iterations reached', maxit)
end
```

Inputs: Solving $y = (x-4)(x+4)$ with Newton

```
% This program solves by the Newton-Raphson method
% the following equation: y = (x-4)*(x+4)

clear all

% Seed
x0 = 1;

% Maximum number of iterations
maxit = 1000;

% Tolerance value
crit = 1e-3;

% Parameters of the system
param = 4;

% Call the newton.m program and specify the file
% with the equations and Jacobian.
sol = newton('nexp1', x0, param, crit, maxit);

sprintf('x=%g', sol(1))
```

nexp1.m: Function and Jacobian

```
function [f, J] = nexp1(z, p)
% nexp1.m
% Function and Jacobian for y = (x - a)*(x + a)

x = z(1);
a = p(1);

% Function value
f = (x - a) * (x + a);

% Jacobian (derivative wrt x)
J = 2*x;
```

Note: With `x0 = 1` and `a = 4`, Newton converges to $x = 4$; with `x0 = -1`, it converges to $x = -4$.

Newton–Raphson Method (multidimension)

Suppose now:

$$f(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{pmatrix}, \quad x \in \mathbb{R}^n.$$

The Taylor expansion becomes:

$$f(x) \approx g(x) = f(x_0) + J_{x_0}(x - x_0),$$

where J_{x_0} is the Jacobian of f evaluated at x_0:

$$J(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}.$$

Newton–Raphson (multidimension): Update

- The solution is updated as:

$$x_1 = x_0 - J_{x_0}^{-1} f(x_0).$$

- Iteratively:

$$x_{s+1} = x_s - J_{x_s}^{-1} f(x_s).$$

- Works well and converges fast if:
  - The initial guess x_0 is sufficiently close to the true root.
  - The Jacobian J_{x_s} is well-conditioned.
- **Example 2:** Solve the system

$$x^2 - 4 + y = 0, \quad y + 5 = 0.$$

Example 2: Newton–Raphson in Two Dimensions

```
% This program solves by the Newton-Raphson method
% the system: x^2 - 4 + y = 0,  y + 5 = 0

clear all

% Seed (initial guess)
x0 = [1; 1];

% Maximum number of iterations
maxit = 1000;

% Tolerance value
crit = 1e-3;

% Parameters of the system
param = [2; 4; 5];

% Call the newton.m program and specify the file
% with the equations and Jacobian.
sol = newton('nexp2', x0, param, crit, maxit);

sprintf('x=%g', sol(1))
sprintf('y=%g', sol(2))
```

nexp2.m: Function and Jacobian

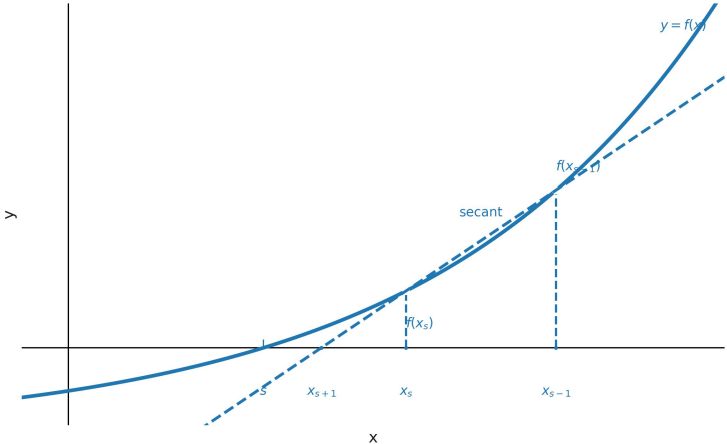
```
function [f, J] = nexp2(z, p)
% nexp2.m
% Function and Jacobian for the system:
%   x^2 - 4 + y = 0
%   y + 5 = 0

x = z(1);
y = z(2);
a = p(1);
b = p(2);
c = p(3);

% System of equations
f = [x^a - b + y;
     y + c];

% Jacobian matrix
J = [a*x^(a-1), 1;
     0,         1];
```

Solution: There is a unique root: $(x, y) = (3, -5)$.

Problems with Newton–Raphson

- **Choice of initial condition:** bad x_0 can lead to divergence or convergence to the wrong root.
- **Undefined iterates:** $f(x_{s+1})$ may be undefined (or $f'(x_s) = 0$), breaking the update.
- **Derivative costs:** computing f' (or the Jacobian in multi-D) can be costly or infeasible.
- **Nonlinear pathologies:** poor conditioning, flat regions, or oscillations can stall progress.

Secant Method



Another Method to Find Zeros: Secant

Idea: Like Newton, but *no derivatives required*.

- Replace the tangent slope $f'(x_s)$ by a *secant* slope built from two nearby points.
- In 1D:

$$x_{s+1} = x_s - f(x_s)\,\frac{x_s - x_{s-1}}{f(x_s) - f(x_{s-1})}.$$

- **Pros:** Often faster than bisection and avoids computing derivatives.
- **Cons:** Needs two initial points and can be less stable than Newton near multiple roots.

Multidimensional intuition (toward Broyden): Approximate the Jacobian numerically and update it iteratively instead of evaluating exact derivatives.

Using the Secant instead of the Tangent

- What if we cannot compute the derivative?
- Recall that:

$$J_1(x) = \lim_{h_1 \to 0} \frac{F(x) - F(x_1 + h_1, x_2, \ldots, x_n)}{h_1},$$

$$J_2(x) = \lim_{h_2 \to 0} \frac{F(x) - F(x_1, x_2 + h_2, \ldots, x_n)}{h_2},$$

$$J_n(x) = \lim_{h_n \to 0} \frac{F(x) - F(x_1, x_2, \ldots, x_n + h_n)}{h_n}.$$

Using the Secant instead of the Tangent

- Notice that $J_i(x)$ is a column vector with the n partial derivatives with respect to x_i. Therefore,

$$J(x) = [J_1(x), J_2(x), \ldots, J_n(x)].$$

We can define a small value h, such that

$$J_1(x) \approx \frac{F(x) - F(x_1 + h, x_2, \ldots, x_n)}{h},$$

$$J_2(x) \approx \frac{F(x) - F(x_1, x_2 + h, \ldots, x_n)}{h},$$

$$J_n(x) \approx \frac{F(x) - F(x_1, x_2, \ldots, x_n + h)}{h}.$$

Secant in $\mathbb{R}^n$: algorithm behind `secant.m`

Goal: Solve $F(x) = 0$ without analytic derivatives (build J by finite differences).

Inputs (`secant(func, x0, param, crit, maxit)`):

- Initial guess $x_0 \in \mathbb{R}^n$, tolerances `crit`, max iterations `maxit`.
- Per-coordinate step sizes: $\Delta = \mathrm{diag}\big(\max\big(|x_0|\,10^{-4},\, 10^{-8}\big)\big)$.

Secant in $\mathbb{R}^n$: algorithm behind `secant.m`

Goal: Solve $F(x) = 0$ without analytic derivatives (build J by finite differences).

Inputs (`secant(func, x0, param, crit, maxit)`):

- Initial guess $x_0 \in \mathbb{R}^n$, tolerances `crit`, max iterations `maxit`.
- Per-coordinate step sizes: $\Delta = \mathrm{diag}\big(\max\big(|x_0|\,10^{-4}, 10^{-8}\big)\big)$.

Loop (for $s = 0, 1, \ldots$):

1. Evaluate $f_s = F(x_s)$.
2. Build J_s by columns with backward differences:

$$\big(J_s\big)_{:j} \;=\; \frac{f_s - F\big(x_s - \Delta e_j\big)}{\Delta_{jj}}, \quad j = 1, \ldots, n.$$

3. Solve the linear system $J_s\,\Delta x_s = f_s$ and update

$$x_{s+1} \;=\; x_s - \Delta x_s.$$

4. Stop if $\|x_{s+1} - x_s\| < $ `crit`; else set $x_s \leftarrow x_{s+1}$ and repeat.

Notes: Δ scales steps to each component; the FD Jacobian uses only function calls. In code this appears as: *(i)* step matrix `del`, *(ii)* Jacobian build in a `for j` loop, *(iii)* update `x = x0 - inv(J)*f`, and *(iv)* convergence check `norm(x-x0)<crit`.

```
function x = secant(func, x0, param, crit, maxit)
% secant.m solves a system f(z1,...,zn)=0 with the secant method
% x is the solution vector; 'func' is a string with the function name.

% Per-coordinate finite-difference steps
del = diag(max(abs(x0)*1e-4, 1e-8));
n   = length(x0);

for i = 1:maxit
    f = feval(func, x0, param);
    % Build Jacobian approximation by columns (backward differences)
    for j = 1:n
        J(:,j) = ( f - feval(func, x0 - del(:,j), param) ) / del(j,j);
    end
    % Newton-secant step
    x = x0 - inv(J)*f;
    if norm(x - x0) < crit
        break
    end
    x0 = x;
end

if i >= maxit
    sprintf('maximum number of iterations was reached')
end
```

Before, we have to provide the inputs of the code

```
% This program solves by the Secant method
% the following equation: y = (x-4)*(x+4)
clear all

% Seed
x0 = 1;

% Maximum number of iterations
maxit = 1000;

% Tolerance value
crit = 1e-3;

% Parameters of the system
param = 4;

% Call the secant.m program and specify the file
% with the equations (no Jacobian needed).
sol = secant('sexp1', x0, param, crit, maxit);

sprintf('x=%g', sol(1))
```

Program with the function sexp1.m

```
function [f] = sexp1(z, p)
% sexp1.m
% Function for the scalar equation: y = (x - a)*(x + a)

x = z(1);
a = p(1);

f = (x - a) * (x + a);
```

With x0 = 1 and a = 4, the secant routine converges to the root $x = 4$; with x0 = -1, it converges to $x = -4$.

Optimal Growth Problem in Discrete Time:

Suppose consumer maximizes:

$$\max_{\{c_t, k_{t+1}\}_{t=0}^{\infty}} U_0 = \sum_{t=0}^{\infty} \beta^t u(c_t), \qquad 0 < \beta < 1,$$

s.t.

$$k_{t+1} + c_t \leq f(k_t) + (1-\delta)k_t, \quad k_0 > 0 \text{ given}, \quad c_t \geq 0.$$

Note: $u(\cdot)$ is strictly concave and technology $f(\cdot)$ is strictly concave and satisfies INADA conditions.

Solution

Euler, resource constraint, TVC:

$$u'(c_t) = \beta\big(f'(k_{t+1}) + (1-\delta)\big)u'(c_{t+1}), \qquad k_{t+1} + c_t = f(k_t) + (1-\delta)k_t,$$

$$k_0 \text{ given}, \qquad \lim_{T\to\infty} \beta^T u'(c_T)\, k_{T+1} = 0.$$

Let: $u(c) = \dfrac{c^{1-\eta}}{1-\eta}$ and $f(k) = Ak^\alpha$. Then

$$\beta\big(Ak_{t+1}^\alpha + (1-\delta)k_{t+1} - k_{t+2}\big)^{-\eta}\Big(\alpha A k_{t+1}^{\alpha-1} + (1-\delta)\Big) - \big(Ak_t^\alpha + (1-\delta)k_t - k_{t+1}\big)^{-\eta} = 0,$$

a second-order difference equation $\phi(k_{t+2}, k_{t+1}, k_t) = 0$ with steady state:

$$k_{ss} = \left[\frac{\alpha A}{\frac{1}{\beta} - (1-\delta)}\right]^{\frac{1}{1-\alpha}}.$$

Approximate Transition & Implementation

Idea: $k_t \to k_{ss}$ only as $T \to \infty$, but convergence is fast. For given k_0, pick a finite T (e.g., $T = 30$) and construct:

$$k = \left[k_0, k_1, \ldots, k_{T+1}\right]$$

solving:

$$\phi(k_2, k_1, k_0) = 0,\ \phi(k_3, k_2, k_1) = 0,\ \ldots,\ \phi(k_{T+1}, k_T, k_{T-1}) = 0,$$

with k_0 given and $k_{T+1} = k_{ss}$ (T equations, T unknowns).

foc.m - First-Order Conditions (1/2)

```
function f = foc(x, param)
% foc.m
% Residuals of the Euler equation for the optimal growth model
% x:     T-by-1 vector with k_1,...,k_T
% param = [A alpha delta eta beta T k0 kss]

A      = param(1);
alpha  = param(2);
delta  = param(3);
eta    = param(4);
beta   = param(5);
T      = param(6);
k0     = param(7);
kss    = param(8);

% Build capital path including k0 and terminal k_{T+1}=kss
k = [k0; x(:); kss];                  % length T+2
f = zeros(T,1);
```

foc.m - First-Order Conditions (2/2)

```
19 for t = 1:T
20     kt   = k(t);
21     kt1  = k(t+1);
22     kt2  = k(t+2);
23     % Resource constraints
24     c_t  = A*kt^alpha  + (1-delta)*kt  - kt1;
25     c_tp = A*kt1^alpha + (1-delta)*kt1 - kt2;
26     % Euler (CRRA: u'(c)=c^(-eta))
27     f(t) = - c_t^(-eta) ...
28            + beta*(alpha*A*kt1^(alpha-1) + (1-delta)) * c_tp^(-eta);
29 end
30 end
```

growth.m - 1/2

```
% growth.m  Solve the finite-horizon shooting problem with Secant
clear; clc;

% --- Model parameters
A=10; alpha=0.36; delta=0.06; eta=0.99; beta=0.96;

% --- Horizon and steady state
kss = ((A*beta*alpha) / (1 - (1-delta)*beta))^(1/(1-alpha));
T   = 30;

% --- Initial capital and seed for path (k_1,...,k_T)
k0  = 0.8*kss;
x0  = [k0; k0*ones(T-1,1)];     % simple seed near kss

% --- Solver setup
maxit = 1000;
crit  = 1e-6;

% --- Pack parameters and call secant
param = [A alpha delta eta beta T k0 kss];
sol   = secant('foc', x0, param, crit, maxit);   % requires secant.m in path
```

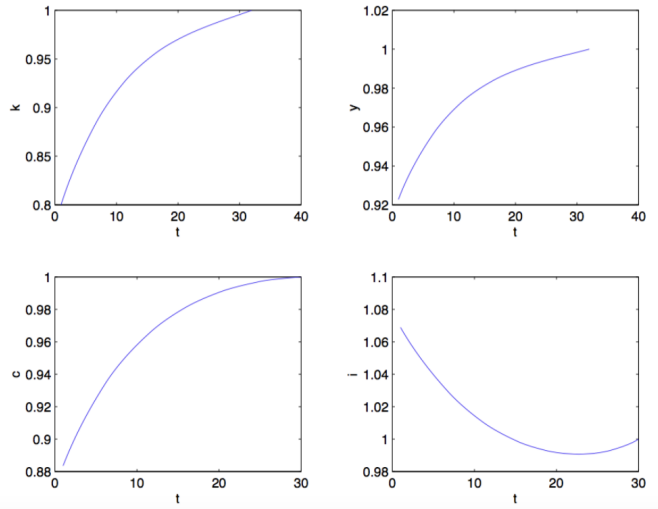
growth.m - 2/2

```
1
2  % --- Recover full paths
3  k = [k0; sol; kss];
4  y = A*k.^alpha;
5  i = k(2:end) - (1-delta)*k(1:end-1);
6  c = y(1:end-1) - i;
7
8  % --- Report
9  fprintf('Converged: k0=%.4f, kss=%.4f, k_T=%.4f\n', k0, kss, k(end-1));
10 fprintf('Min/Max c: [%.4f, %.4f]\n', min(c), max(c));
```

plot_growth_paths.m - Quick Plots

```
% plot_growth_paths.m (run after growth.m so k,y,i,c are in workspace)
t  = 0:(numel(k)-2);    % time for c,y,i (length T+1 for k)

figure;
subplot(3,1,1); plot(0:numel(k)-1, k, '-o'); grid on;
title('Capital k_t'); xlabel('t'); ylabel('k');

subplot(3,1,2); plot(t, [y(1:end-1), c, i]); grid on;
title('Output/Consumption/Investment'); xlabel('t');
legend('y_t','c_t','i_t','Location','best');

subplot(3,1,3); plot(t, abs(diff(k))); grid on;
title('|k_{t+1}-k_t| (convergence)'); xlabel('t'); ylabel('abs diff');
```

Transition Dynamics of the Neoclassical Growth Model



Issues on Numerical Differentiation

- The **secant method above is based on numerical differentiation**.
- Taking h very small (e.g., $h \ll \varepsilon$ where ε is machine precision) means $x = x + h$. Ensure $h > \varepsilon$.
- For h close to 0, $f(x + h) - f(x)$ loses many digits of accuracy.
- **Taking h large means you are probably not near the limit**.

Secant

- h was fixed but h can be relative to the distance between x_{s+1} and x_s. Correct way:

$$f'(x_{s+1}) \approx \frac{f(x_{s+1}) - f(x_s)}{x_{s+1} - x_s}.$$

Taylor approximation: $$x_{s+1} = x_s - \frac{f(x_s)}{f'(x_s)}.$$

Therefore: $$x_{s+1} = x_s - f(x_s)\,\frac{x_s - x_{s-1}}{f(x_s) - f(x_{s-1})}.$$

Secant Method for the Multidimensional Case (Broyden)

- In the one-dimensional case the slope m near two points x and z is given by $f(x) - f(z) = m(x - z)$ (unique if $x \neq z$).
- In the multidimensional case $f(x) - f(z) = M(x - z)$ does not uniquely define M (n conditions for n^2 elements).
- Broyden suggested approximating the Jacobian iteratively as iterations go.

Broyden's Method

- Need two initial points x_0 and x_1 (as in the secant method).
- Choose an initial Jacobian guess J_0 (e.g., $J_0 = I$).

Define:

$$\delta_s = x_s - x_{s-1}, \qquad \Delta_s = f(x_s) - f(x_{s-1}).$$

Jacobian update (rank-one):

$$J_s \;=\; J_{s-1} \;+\; \frac{\left(\Delta_s - J_{s-1}\delta_s\right)\delta_s^\top}{\delta_s^\top \delta_s}.$$

Step:

$$x_{s+1} \;=\; x_s \;-\; J_s^{-1}\, f(x_s).$$

Takeaways

- **Bisection**: robust, derivative-free, slow.
- **Newton**: fast near root, needs derivatives/initialization.
- **Secant/Broyden**: no derivatives; good practical performance.
- **Function Approximation**: choose local (Taylor) vs global (Chebyshev/splines) based on problem geometry.