Lecture 3

Nonlinear Equations

Randall Romero Aguilar, PhD
II Semestre 2018
Last updated: August 26, 2018

Universidad de Costa Rica
SP6534 - Economía Computacional

Table of contents

Introduction

- The nonlinear equation takes one of two forms.
- Rootfinding Problem: Given a function $f : \Re^n \mapsto \Re^n$, compute an n-vector x^*, called a root of f, such that

$$f(x^*) = 0.$$

- Fixed-Point Problem: Given a function $g : \Re^n \mapsto \Re^n$, compute an n-vector x^*, called a fixed-point of g, such that

$$g(x^*) = x^*.$$

- The two forms are equivalent:
    - A root of f is a fixed-point of $g(x) = x - f(x)$.
    - A fixed-point of g is root of $f(x) = x - g(x)$.

- Nonlinear equations arise naturally in economics:
    - Multicommodity market equilibrium models
    - Multiperson static game models
    - Unconstrained optimization models

- Nonlinear equations also arise indirectly when numerically solving economic models involving functional equations:
    - Dynamic optimization models
    - Rational expectations models
    - Arbitrage pricing models

Function Iteration

- Function iteration is an algorithm for computing a fixed-point of a function $g : \Re^n \mapsto \Re^n$.
- Guess an initial value x_0 and successively form the iterates

$$x_{k+1} = g(x_k)$$

until the iterates converge.

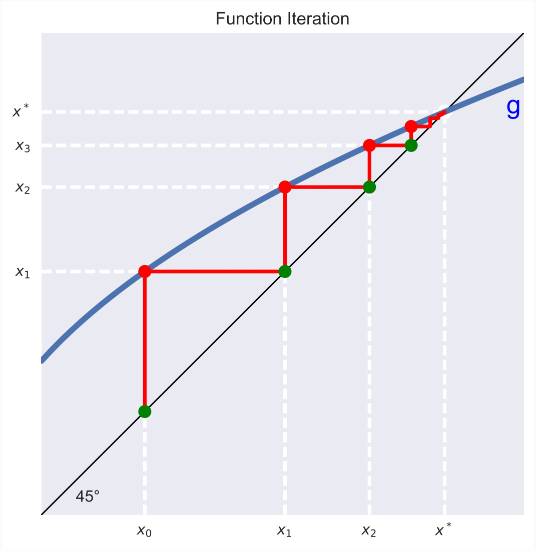

Figure 1: Computing Fixed-Point of g Using Function Iteration

- When is function iteration guaranteed to converge?
- By the Contraction Mapping Theorem, if for some $\delta < 1$,

$$\|g(x_1) - g(x_2)\| \leq \delta\|x_1 - x_2\|$$

  for all x_1 and x_2, then g possesses an unique fixed-point x^* and function iteration converges to it from any initial value x_0.
- Moreover,

$$\|x_k - x^*\| \leq \frac{\delta}{1-\delta}\|x_k - x_{k-1}\|.$$

- More generally, function iteration will also converge if the initial value x_0 is “close” to a fixed-point x^* of g at which

$$\|g'(x^*)\| < 1.$$

- There is no general practical way to determine what “close” is.

Example 1:

Fixed-point of $g(x) = \sqrt{x + 0.2}$

- To compute the fixed-point of

$$g(x) = \sqrt{x + 0.2}$$

  using function iteration, one employs the iteration rule

$$x_{k+1} = g(x_k) = \sqrt{x_k + 0.2}.$$

- In Python, execute the script

```
x=0.4
for it in range(50):
    xold, x = x, np.sqrt(x+0.2)
    if abs(x-xold)<1.e-10: break
```

- After 28 iterations, x converges to 1.1708.

Newton’s Method

- Newton's method is an algorithm for computing a root of a function $f : \Re^n \mapsto \Re^n$.
- Guess an initial value x_0 and successively form the iterates

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k)$$

  until the iterates converge.
- If $n = 1$, the iteration rule takes the simpler form

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

- Newton's method employs a strategy of successive linearization.
- The strategy calls for the nonlinear function f to be approximated by a sequence of linear functions whose roots are easily computed and, ideally, converge to the root of f.
- In particular, the $k+1^{th}$ iterate

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k)$$

  is the root of the Taylor linear approximation of f around the preceding iterate x_k:

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k).$$

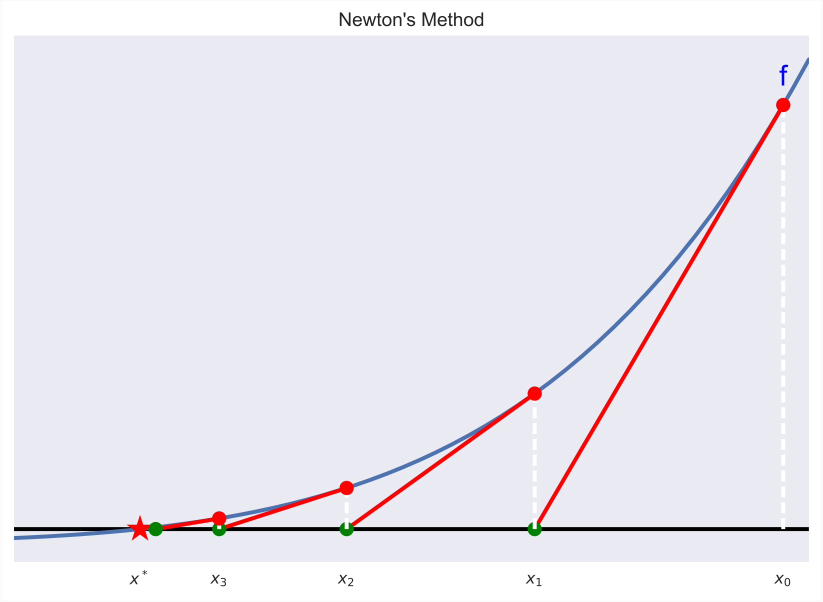

Figure 2: Computing Root of f Using Newton’s Method

- When is Newton's method guaranteed to converge?
- In theory, Newton's method will converge if the initial value x_0 is "close" to a root x^* of f at which $f'(x^*)$ is non-singular.
- Theory, however, provides no practical definition of "close".
- Moreover, in practice, Newton's method will fail to converge to x^* if $f'(x^*)$ is ill-conditioned, i.e., nearly singular.

Example 2:

Root of function $f(x) = x^4 - 2$

- To compute the root of $f(x) = x^4 - 2$ using Newton's method, one employs the iteration rule:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = x_k - \frac{x_k^4 - 2}{4x_k^3}$$

- In Python, execute the script

```python
x = 2.3
for it in range(50):
    print(it, x)
    step = -(x**4-2)/(4*x**3)
    x += step
    if abs(step)<1.e-10: break
```

- After 8 iterations, x converges to 1.1892.

Quasi-Newton Methods

- Quasi-Newton methods replace the Jacobian in Newton's method with an estimate that is easier to compute.
- Specifically, quasi-Newton methods use an iteration rule

$$x_{k+1} = x_k - A_k^{-1} f(x_k)$$

where A_k is an estimate of the Jacobian $f'(x_k)$.

Secant Method

- The Quasi-Newton method for univariate root-finding problems is called the secant method.
- The secant method replaces the derivative in Newton's method with the estimate

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}},$$

  which leads to the iteration rule

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} f(x_k).$$

- The secant method is so called because it approximates the function f using "secant" lines drawn through successive pairs of points on its graph.

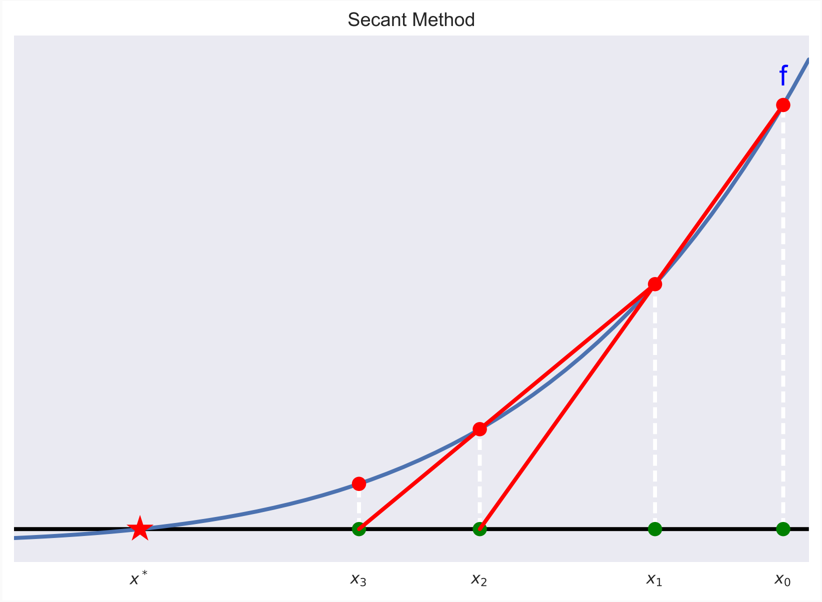

Figure 3: Computing Root of f Using Secant Method

Example 3:

Root of function $f(x) = x^4 - 2$

- To compute the root of $f(x) = x^4 - 2$ using the secant method, execute the script

```python
f = lambda z: z**4-2
x, xlag = 2.3, 2.4

for it in range(50):
    d = (f(x)-f(xlag)) / (x-xlag)
    x, xlag = x-f(x)/d, x
    if abs(x-xlag)<1.e-10: break
```

- After 10 iterations, x converges to 1.1892.

Broyden's Method

- Broyden's method is the most popular multivariate generalization of the univariate secant method.
- Broyden's method replaces the Jacobian in Newton's method with an estimate A_k that is updated by making the smallest possible change that is consistent with the secant condition:

$$f(x_{k+1}) - f(x_k) = A_{k+1}(x_{k+1} - x_k).$$

- This yields the iteration rule

$$A_{k+1} = A_k + \left(f(x_{k+1}) - f(x_k) - A_k d_k\right) \frac{d_k'}{d_k' d_k}$$

where $d_k = x_{k+1} - x_k$.

- When is Broyden’s method guaranteed to converge?
- In theory, Broyden’s method will converge if the initial value x_0 is “close” to a root x^* of f at which the Jacobian $f'(x^*)$ is non-singular and if the initial Jacobian estimate A_0 is “close” to $f'(x^*)$.
- Theory, however, provides no practical definition of “close”.
- Moreover, in practice, Broyden’s method will fail to converge to x^* if the Jacobian estimates A_k become ill-conditioned, i.e., nearly singular.
- Also, even if the x_k converge to x^*, the Jacobian estimates A_k at the final iterate often does not provide an acceptable approximation to the Jacobian $f'(x^*)$ at the root.

Numerical Examples using CompEcon

The NLP class

- The CompEcon package provides class NLP (non-linear problem) for computing the root of function $f : \Re^n \mapsto \Re^n$.
- A nonlinear equations problem is created as follows:

```python
from compecon import NLP

def f(x):  #objective function
    fval = ...  #function value
    fjac = ...  #Jacobian value
    return fval, fjac

problem = NLP(f)
```

Solving by Newton's method

- To solve by Newton's method, simply call the `.newton` method on the `NLP` object, providing an initial value `x0`

```
x = problem.newton(x0)
```

- On output, `x` is the root of the problem.
- To check the value of f, get the attribute `problem.fx`
- Notice that the steps can be combined as in:

```
x = NLP(f).newton(x0)
```

Solving by Broyden’s method

- To solve by Broyden’s method, simply call the `.broyden` method on the `NLP` object, providing an initial value `x0`

```
x = problem.broyden(x0)
```

- On output, `x` is the root of the problem.
- To check the value of f, get the attribute `problem.fx`
- As usual, the steps can be combined as in:

```
x = NLP(f).broyden(x0)
```

- To use Broyden’s method it is not necessary to return the Jacobian of the objective function.

Example 4:

Roots of a function in $\mathcal{R}^2$

- Let us compute the root of

$$f(x,y) = \begin{bmatrix} y\exp(x) - 2y \\ xy - y^3 \end{bmatrix}$$

- To use broyden with initial guess $(1.0, 0.5)$

```
def f(z):
    x, y = z
    fval =[y*np.exp(x) - 2*y, x*y - y**3]
    return np.array(fval)

x0 = np.array([1.0, 0.5])
x = NLP(f).broyden(x0)
```

- On output, typing print(x) results in

```
[ 0.69314718  0.8325546 ]
```

- To use `newton`, we need to compute the jacobian

```
def f(z):
    x, y = z
    fval = [y*np.exp(x) - 2*y, x*y - y**3]
    fjac = [[y*np.exp(x), np.exp(x)-2],
            [y, x-3*y**2]]
    return np.array(fval), np.array(fjac)

x0 = np.array([1.0, 0.5])
x = NLP(f).newton(x0)
```

- On output, typing `print(x)` results in

```
[ 0.69314718  0.83255461]
```

- By default, both `broyden`, and `newton` work quietly.
- To print the iterations, just set the option `print=True`.

  For example,

  ```
  x = NLP(f).newton(x0, print=True)
  ```

  prints the following output

  ```
  Solving nonlinear equations by Newton's method
  it    bstep  change
  --------------------
     0     5  3.69e-01
     1     0  6.76e-02
     2     0  4.69e-03
     3     0  2.89e-05
     4     0  1.11e-09
  ```

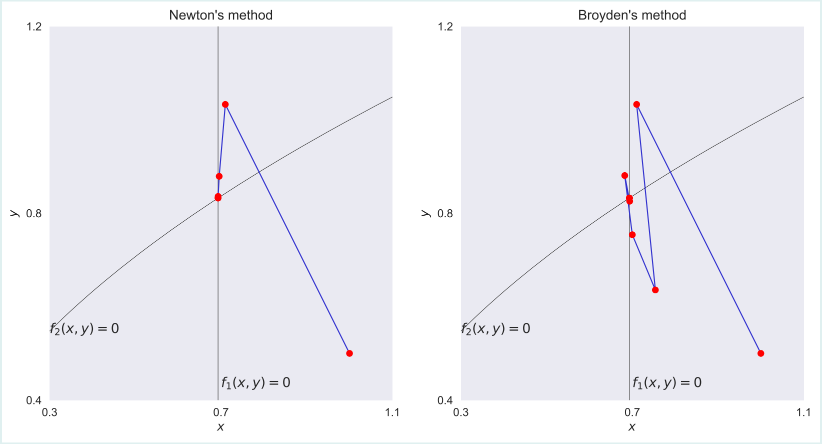

Figure 4: Convergence Paths for Newton and Broyden's Methods

Example 5:

A Cournot Equilibrium

- Consider a market with two firms producing the same good.
- Firm i's total cost of production is a function

$$C_i(q_i) = \frac{\beta_i}{2} q_i^2$$

of the quantity q_i it produces.

- The market clearing price is a function

$$P(q_1 + q_2) = (q_1 + q_2)^{-\alpha}$$

of the total quantity produced by both firms.

- Firm i chooses production q_i so as to maximize its profit

$$\pi_i(q_1, q_2) = P(q_1 + q_2)q_i - C_i(q_i),$$

taking the other firm's output as given.
- Thus, in equilibrium,

$$\frac{\partial \pi_i}{\partial q_i} = P + P'q_i - C_i' = 0$$

for $i = 1, 2$.

- Suppose $\alpha = 0.6$, $\beta_1 = 0.6$, and $\beta_2 = 0.8$.
- To solve the problem using Broyden's method:

```
alpha, beta = 0.6,  np.array([0.6, 0.8])

def cournot(q):
    qsum = q.sum()
    P = qsum**(-alpha)
    P1 = -alpha*qsum**(-alpha-1)
    return P + (P1-beta)*q

NLP(cournot).broyden([0.2,0.2])
```

- Here, `cournot` computes the marginal profits of both firms.
- After 9 iterations, q converges to $q^* = (0.8562, 0.700)$.

To compute the equilibrium using Newton's method, define the function `cournot2` so that it also returns the Jacobian `J`:

```
alpha, beta = 0.6,  np.array([0.6, 0.8])

def cournot2(q):
    qsum = q.sum()
    P = qsum**(-alpha)
    P1 = -alpha*P/qsum
    P2 = (-alpha-1)*P1/qsum
    f = P + (P1-beta)*q
    J = P1 + P2*np.repeat(q,[2]).reshape(2,2) + np.diag(P1-beta)
    return f, J
```

The market equilibrium can then be computed by calling the `newton` method on `NLP`:

```
NLP(cournot).newton([0.2,0.2])
```

After 5 iterations, q converges to $q_1^* = 0.856$ and $q_2^* = 0.700$.

Of course, if you do not have access to the CompEcon Toolbox, you can always directly compute the market equilibrium using Newton's method:

```
alpha, beta = 0.6,  np.array([0.6, 0.8])
q = np.array([0.2,0.2])

for it in range(40):
    f, J = cournot2(q)
    step = -np.linalg.solve(J,f)
    q += step
    if np.linalg.norm(step)<1e-10: break
```

On output, typing `print(q)` results in

```
[0.8562 0.7   ]
```

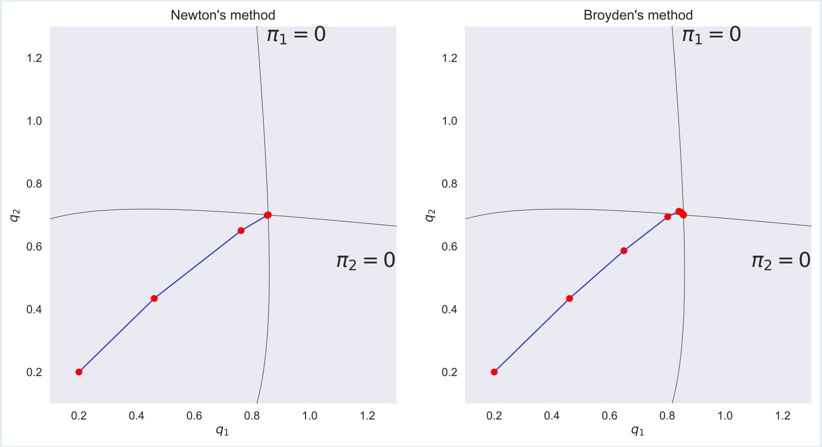

Figure 5: Convergence Paths for Newton and Broyden’s Methods

Practical Issues

Failure to Converge

In practice, nonlinear equation algorithms can fail for various reasons:

- Human error
- Bad initial value
- Ill-conditioning

Human error

- Math error: analyst incorrectly derives function or Jacobian.
- Coding error: analyst incorrectly codes function or Jacobian.
- Errors more likely with Newton’s method because Jacobian must be correctly derived and coded.
- Errors less likely with function iteration and Broyden’s method because they are derivative-free.

Bad initial value

- Nonlinear equation algorithms require initial values.
- If initial value is far from desired root, algorithm may diverge or converge to the “wrong” root.
- Theory provides no guidance on how to specify initial value.
- Analyst must supply good guess from knowledge of model.
- If iterates diverge, try another initial value.
- Well behaved functions more robust to initial value.
- Poorly behaved functions more sensitive to initial value.

Ill-conditioning

- Computing iteration step in Newton's and Broyden's methods requires solution to a linear equation involving the Jacobian or its estimate.
- If the Jacobian or estimate is ill-conditioned near the solution, the iteration step cannot be accurately computed.
- Very little can be done about this.
- It arises more often than we like.

Execution Speed

Two factors determine the speed with which a properly coded and initiated algorithm will converge to a solution:

- Asymptotic rate of convergence
- Computational effort per iteration

Asymptotic rate of convergence

- The asymptotic rate of convergence measures improvement afforded per iteration near the solution.
- A sequence x_k converges to x^* at an asymptotic rate of order p if there is constant $C > 0$ such that for k sufficiently large,

$$\|x_{k+1} - x^*\| \leq C\|x_k - x^*\|^p.$$

- Function iteration converges at a “linear” rate with $p = 1$ and $C < 1$ – relatively slow.
- Broyden’s method converges at a “superlinear” rate with $p \approx 1.62$ – relatively fast.
- Newton’s method converges at a “quadratic” rate with $p = 2$ – extremely fast.

Convergence rates

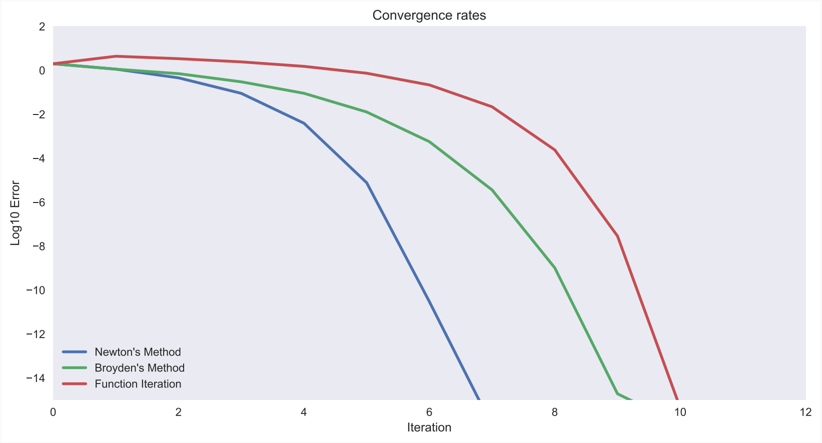

Figure 6: Rate of Convergence When Computing Fixed-Point of $x - \exp(x) + 1$ Using Various Methods, $x_0 = 2$

Computational effort per iteration

- However, algorithms differ in computations per iteration.
- Function iteration requires a function evaluation.
- Broyden’s method additionally requires a linear solve.
- Newton’s method additionally requires a Jacobian evaluation.
- Thus, a faster rate of convergence typically can be achieved only by investing greater computational effort per iteration.
- The optimal tradeoff between rate of convergence and computational effort per iteration varies across applications.

Choosing a Solution Method

- Concerns about execution speed, however, are exaggerated.
- The time that must be invested by the analyst to write and debug code typically is far more important.
- Derivative-free methods such as function iteration and Broyden's method can be implemented faster in real time and more reliably than Newton's method.
- Newton's method should be used only if
    - dimension is low or derivatives are simple,
    - other methods have failed to converge, or
    - producing general purpose, re-usable code.

Complementarity Problems

Complementarity Problems

Many phenomena in economics and finance are naturally modelled by requiring pairs of inequalities to be satisfied in complementary fashion, that is, such that at least one of the two inequalities holds with strict equality. These include

- Partial equilibrium models in which prices and quantities are bounded (quotas, capacity bounds, price floors, price ceilings, nonnegativity).
- Static optimization models in which a function must be maximized subject to bound constraints.

General setup

The general complementarity problem takes the following form: Given a function f from $\Re^n$ to $\Re^n$, $a \in \Re^n$ and $b \in \Re^n$, find $x \in \Re^n$ such that for every $i = 1, \ldots, n$,

$$a_i \leq x_i \leq b_i$$

$$x_i > a_i \;\Rightarrow\; f_i(x) \geq 0$$

$$x_i < b_i \;\Rightarrow\; f_i(x) \leq 0.$$

For brevity, we denote the complementarity problem

$$f(x) \perp [a, b].$$

Rootfinding vs. complementarity problems

- The rootfinding problem is a special case of the complementarity problem in which $a_i = -\infty$ and $b_i = \infty$ for all i.
- The complementarity problem, however, is not to find a root that lies within specified bounds.
- $f_i(x)$ may be positive at a solution, though only if x_i equals its upper bound.
- $f_i(x)$ may be negative at a solution, though only if x_i equals its upper bound.

Arbitrage conditions as complementarity problems

Complementarity problems in economics and finance typically admit an arbitrage interpretation:

- There are n economic activities.
- The level of activity i is denoted x_i.
- The level of activity i is bounded below by a_i and above by b_i.
- The marginal profit of activity i is $f_i(x)$.
- Profits can be raised by decreasing x_i if $x_i > a_i$ and $f_i(x) < 0$.
- Profits can be raised by increasing x_i if $x_i < b_i$ and $f_i(x) > 0$.
- Equilibrium obtains if and only if all profit opportunities have been eliminated, that is, if and only if x solves $f(x) \perp [a, b]$.

The Karush-Kuhn-Tucker theorem asserts that x maximizes a function $f : \Re^n \mapsto \Re$ subject to the bound constraint $a \leq x \leq b$ only if it solves the complementarity problem $f'(x) \perp [a, b]$, that is, only if, for all i,

$$\begin{aligned} & a_i \leq x_i \leq b_i \\ & x_i > a_i \;\Rightarrow\; f_i'(x) \geq 0 \\ & x_i < b_i \;\Rightarrow\; f_i'(x) \leq 0. \end{aligned}$$

Example 6:

Price Ceiling

- Excess demand for a good, quantity demanded less quantity supplied, is a function $E(p)$ of the good's market price.
- If the commodity market is competitive, the equilibrium price is characterized by the rootfinding problem $E(p) = 0$.
- However, if the government imposes a price ceiling $\bar{p}$, the equilibrium price is characterized by the complementarity problem $E(p) \perp [-\infty, \bar{p}]$:

$$\begin{aligned}
&-\infty \leq p \leq \bar{p} \\
&E(p) \geq 0 \\
&p < \bar{p} \ \Rightarrow \ E(p) \leq 0.
\end{aligned}$$

- Excess demand for the good, $E(p) > 0$, may exist in equilibrium if the price ceiling is binding.

Example 7:

Minimum Wage

- Excess demand for labor, labor demanded less labor supplied, is a function $E(w)$ of the wage rate.
- If the labor market is competitive, the equilibrium wage is characterized by the rootfinding problem $E(w) = 0$.
- However, if the government imposes a minimum wage $\bar{w}$, the equilibrium wage is characterized by the complementarity problem $E(w) \perp [\bar{w}, \infty]$

$$
\begin{aligned}
&\bar{w} \leq w \leq \infty \\
&w > \bar{w} \;\Rightarrow\; E(w) \geq 0 \\
&E(w) \leq 0.
\end{aligned}
$$

- Excess supply of labor, $E(w) < 0$, may exist in equilibrium if the minimum wage is binding.

Example 8:

Spatial Price Equilibrium

- A commodity is produced and consumed in n regions.
- The quantity x_{ij} produced in region i and consumed in region j cannot be negative or exceed a shipping capacity b_{ij}.
- The cost c_{ij} of transporting one unit of the commodity from producers in region i to consumers in region j is constant.
- In region i, an inverse supply function gives the price p_i^s producers must be paid to produce quantity $q_i^s = \sum_j x_{ij}$.
- In region i, an inverse demand function gives the price p_i^d consumers are willing to pay to purchase quantity $q_i^d = \sum_j x_{ji}$.

- The profit from producing one unit of the commodity in region i and selling it to consumers in region j is

$$\pi_{ij}(x) = p^d_j(x) - p^s_i(x) - c_{ij}$$

- Equilibrium obtains only if all profit opportunities have been eliminated, that is, only if the commodity flows x solve the complementary problem $\pi(x) \perp [0, b]$:

$$\begin{array}{lcl} 0 \leq x_{ij} \leq b_{ij} & & \\ x_{ij} > 0 & \Rightarrow & \pi_{ij}(x) \geq 0 \\ x_{ij} < b_{ij} & \Rightarrow & \pi_{ij}(x) \leq 0. \end{array}$$

Existence and Uniqueness of Solution

- The complementarity problem $f(x) \perp [a, b]$ is guaranteed to possess an unique solution if f is strictly negative monotone, that is, if $(x - y)'(f(x) - f(y)) < 0$ whenever $x, y \in [a, b]$ and $x \neq y$.
- This is a multidimensional generalization of the one-dimensional condition that f be strictly decreasing.
- The condition is satisfied by most economic models.
- The condition is satisfied by the Karush-Kuhn-Tucker conditions of a bound-constrained maximization problem with strictly concave objective function.

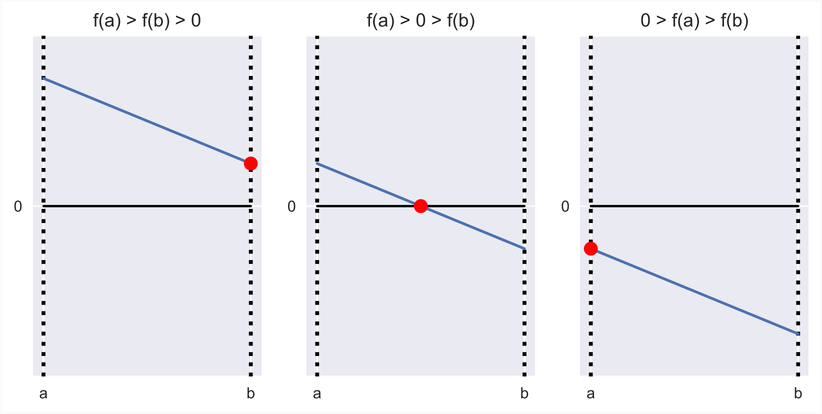

Figure 7: Possible Solutions to Complementarity Problem, f Strictly Decreasing

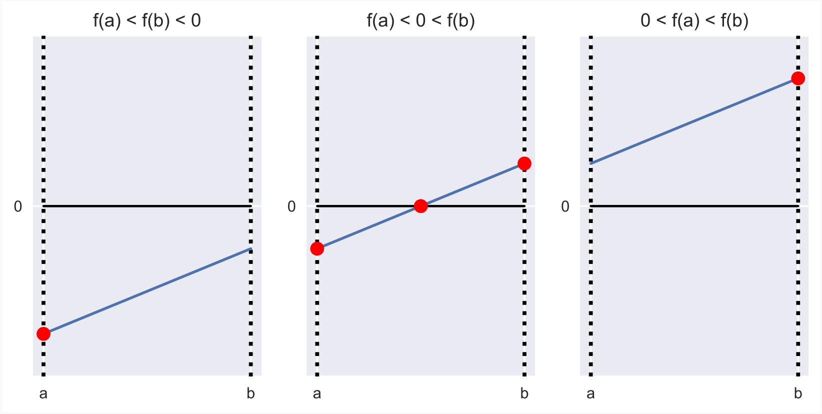

Figure 8: Possible Solutions to Complementarity Problem, f Strictly Increasing

Numerical Solution Methods

- A complementarity problem can be recast as an equivalent root-finding problem and solved using standard nonlinear equation methods such as Newton's or Broyden's method.
- In particular, x solves the complementarity problem $f(x) \perp [a,b]$ if and only if it solves the rootfinding problem

$$\hat{f}(x) = \min(\max(f(x), a - x), b - x) = 0$$

  where $\min$ and $\max$ are applied row–wise.
- Attempting to solve $f(x) \perp [a,b]$ by computing the root of $\hat{f}(x)$, the so-called "min-max" formulation, is computationally inexpensive and often works well in practice.

- However, the “min-max” formulation leads to a “kinky” problem that sometimes encounters computational difficulties.
- In can also be shown that x solves the complementarity problem $f(x) \perp [a, b]$ if and only if it solves the rootfinding problem

$$\tilde{f}(x) = \phi^{-}(\phi^{+}(f(x), a - x), b - x) = 0$$

where

$$\phi_i^{\pm}(u, v) = u_i + v_i \pm \sqrt{u_i^2 + v_i^2}.$$

- Attempting to compute the root of $\tilde{f}(x)$, the so-called “semi-smooth” formulation, is computationally more expensive, but presents a smoother problem that can often be solved if the other cannot.

The MCP class

- The CompEcon package provides class MCP for solving the nonlinear complementarity problem $f(x) \perp [a, b]$ where $f : \Re^n \mapsto \Re^n$.
- A nonlinear complementarity problem is created as follows:

```
from compecon import MCP
a, b = ...  #lower and upper bounds
def f(x):   #objective function
    fval = ...  #function value
    fjac = ...  #function Jacobian
    return fval, fjac

problem = MCP(f, a, b)
```

Solving by “min-max” transformation

- To use the “min-max” transformation, just call the `.zero` method on the `MCP` object, providing an initial value `x0`, and setting the option `transform='minmax'`

```
x = problem.zero(x0, transform='minmax')
```

- On output, `x` is the solution of the problem.
- To check the value of f, get the attribute `problem.fx`
- Notice that the steps can be combined as in:

```
x = MCP(f, a, b).zero(x0, transform='minmax')
```

Solving by “semismooth” transformation

- To use the “min-max” transformation, again call `.zero` on the `MCP` object, providing an initial value `x0`, and setting `transform='ssmooth'`

```
x = problem.zero(x0, transform='ssmooth')
```

- On output, `x` is the root of the problem.
- To check the value of f, get the attribute `problem.fx`
- Again, the steps can be combined as in:

```
x = MCP(f,a,b).zero(x0,transform='ssmooth')
```

Example 9:

A simple nonlinear complementarity problem

- Consider the complementarity problem $f(x, y) \perp [a, b]$ where

$$f(x,y) = \begin{bmatrix} 1 + xy - 2x^3 - x \\ 2x^2 - y \end{bmatrix}$$

and $0 \leq x \leq 1$, $0 \leq y \leq 1$, that is,

$$a = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

- To solve the model using CompEcon class MCP, first code the function, and pass it to MCP together with the bounds:

```python
def func(z):
    x, y = z
    return np.array([1 + x*y - 2*x**3 - x,
                     2*x**2 - y])
F = MCP(func, [0, 0], [1,1])
```

- Then, to solve it starting from $(x, y) = (0.5, 0.5)$ using the “minmax” transformation

```python
x0 = [0.5, 0.5]
x = F.zero(x0, transform='minmax')
```

- After 8 iterations, a solution $x = (0.7937, 1)$ is generated.

- The solution above was computed without coding the Jacobian
- To compute the solution with the Jacobian, change the definition of `func` to

```python
def func2(z):
    x, y = z
    f = [1 + x*y - 2*x**3 - x, 2*x**2 - y]
    J = [[y-6*x**2-1, x],[4*x, -1]]
    return np.array(f), np.array(J)

F2 = MCP(func2, [0, 0], [1,1])
x = F2.zero(x0, transform='minmax')
```

- After 4 iterations, a solution $x = (0.7937, 1)$ is generated.

Example 10:

Trade between 3 countries

- A commodity is produced and consumed in three countries.
- Demand and supply in the three countries is given by:

	Demand	Supply
Country 1:	$p = 42 - 2q$	$p = 9 + 1q$
Country 2:	$p = 54 - 3q$	$p = 3 + 2q$
Country 3:	$p = 51 - 1q$	$p = 18 + 1q$

- The unit costs of transportation are:

From / to	Country 1	Country 2	Country 3
Country 1:	0	3	9
Country 2:	3	0	3
Country 3:	6	3	0

To solve the model, set the parameter and define a function `market` that takes the vector of flows `x` as input and returns the potential arbitrage profits `fval`, with `fjac` empty:

```python
import numpy as np
from compecon import MCP

A = np.array
as_, bs = A([9, 3, 18]), A([1, 2, 1])  #supply
ad, bd = A([42, 54, 51]), A([2, 3, 1]) #demand
c = A([[0, 3, 9], [3, 0, 3],[6, 3, 0]]) #transp.cost

def market(x, jac=False):
    quantities = x.reshape((3,3))
    ps = as_ + bs * quantities.sum(0)
    pd = ad - bd * quantities.sum(1)
    ps, pd = np.meshgrid(ps, pd)
    fval = (pd - ps - c).flatten()
    return (fval, None) if jac else fval
```

Then create a MCP object

```
a = np.zeros(9)
b = np.full(9, np.inf)
Market = MCP(market, a, b)
```

and solving starting from zero quantities:

```
x0 = np.zeros(9)
x = Market.zero(x0, transform='minmax')

quantities = x.reshape(3,3)
prices = as_ + bs * quantities.sum(0)
exports = quantities.sum(0) - quantities.sum(1)
```

After 31 iterations, the solution

```
quantities =
 [[ 9.      -0.       0.     ]
 [ 1.6347  7.3653  0.     ]
 [ 4.3653  4.6347 12.     ]]

prices =
 [24. 27. 30.]

exports =
 [ 6.  3. -9.]
```

is generated.