AGEC 652 - Lecture 3.2

Nonlinear Equations

Diego S. Cardoso

Spring 2022

Course roadmap

1. Intro to Scientific Computing
2. Numerical operations and representations
3. **Systems of equations**
    1. Linear equations
    2. **Nonlinear equations** ← You are here
4. Optimization
5. Structural estimation

*These slides are based on Miranda & Fackler (2002), Judd (1998), and course materials by Ivan Rudik.

Types of nonlinear equation problems

Problems involving nonlinear equations are very common in economics

We can classify them into two types

1. Systems of nonlinear equations
2. Complementarity problems

System of nonlinear equations

These come in two forms

A) **Rootfinding problems**

For a function $f : \mathbb{R}^n \to \mathbb{R}^n$, we want to find a n-vector x that satisfies $f(x) = 0$

$\to$ Any x that satisfies this condition is called a **root** of f

Examples?

- Market equilibrium in general: market clearing conditions
- No-arbitrage conditions (pricing models)
- Solving first-order optimality conditions

System of nonlinear equations

B) **Fixed-point problems**

For a function $g : \mathbb{R}^n \to \mathbb{R}^n$, we want to find a n-vector x that satisfies $x = g(x)$
$\to$ Any x that satisfies this condition is called a **fixed point** of g

Examples?

- Best-response functions
- Many equilibrium concepts in game theory
  - A *Nash equilibrium* is a fixed point in the strategy space

System of nonlinear equations

Rootfinding and fixed-point problems are equivalent

We can easily convert one into another

- *Rootfinding* $\rightarrow$ *fixed-point*
  - Define a new $g(x) = x - f(x)$
- *Fixed-point* $\rightarrow$ *rootfinding*
  - Define a new $f(x) = x - g(x)$

Complementarity problems

In these problems, we have

- A function $f : \mathbb{R}^n \to \mathbb{R}^n$
- n-vectors a and b, with $a < b$

And we are looking for an n-vector $x \in [a, b]$ such that for all $i = 1, \ldots, n$

$$x_i > a_i \Rightarrow f_i(x) \geq 0$$
$$x_i < b_i \Rightarrow f_i(x) \leq 0$$

Examples of complementarity problems?

- Market equilibrium with constraints: quotas, price support, non-negativity, limited capacity, etc
- First-order conditions of constrained function maximization/minimization

Complementarity problems

$$x_i > a_i \Rightarrow f_i(x) \geq 0$$
$$x_i < b_i \Rightarrow f_i(x) \leq 0$$

What do these equations mean?

If the constraints on x_i do not bind ($a_i < x_i < b_i$) , then the first-order condition is precisely zero

But suppose the upper bound binds ($x_i = b_i$). Then $f_i(x) \geq 0$ since $x_i > a_i$

- But we can't guarantee that $f_i(x) = 0$ because $f_i(x)$ might still be increasing at that point

Complementarity problems

Rootfinding is a special case of complementarity problems: $a = -\infty$ and $b = \infty$

But complementarity problems are *not just about finding a root within* $[a, b]$

- Remember: if some x_i is at the boundary ($x_i = a_i$ or $x_i = b_i$), some element of $f(x)$ can be non-zero!

Rootfinding and fixed-point problems

Rootfinding methods

Let's start simple: we have a continuous function $f : [a, b] \in \mathbb{R} \to \mathbb{R}$ and we know that $f(a) < 0$ and $f(b) > 0$

What does the Intermediate Value Theorem says here?

> If f is continuous and $f(a) \neq f(b)$, then f must assume all values in between $f(a)$ and $f(b)$

- So if $f(a) < 0$ and $f(b) > 0 \Rightarrow$ there must be at least one root $x \in [a, b]$ such that $f(x) = 0$

How would you go about finding a root?

Bisection method

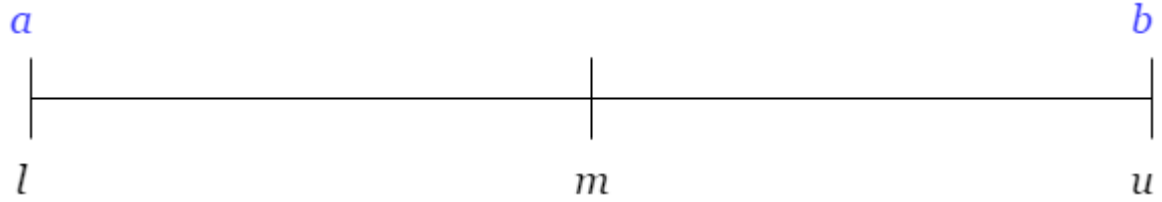
Basic idea: split the search interval in two parts and check whether there's a root in each part

How do we check that?

- By looking at the signs of $f(x)$ at the boundaries of each interval
  - If they are different, there's a root $\Rightarrow$ we keep looking there

Let's see an illustration

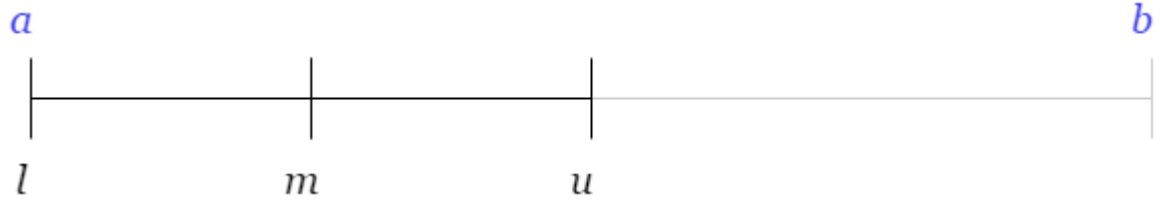
Bisection method



We start with $l = a, u = b$ and find $m = (u + l)/2$

Let's say $f(l) < 0$, $f(u) > 0$, and $f(m) > 0$. What do we do next?

Bisection method



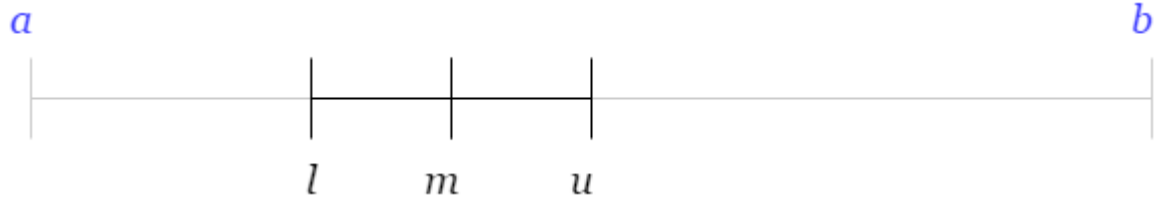
Since $f(l) < 0$ and $f(m) > 0$ have different signs, we continue our search in $[l, m]$

We set $u \leftarrow m$

Then we calculate the new midpoint m

Now say $f(l) < 0$, $f(u) > 0$, and $f(m) < 0$. What do we do next?

Bisection method



Since $f(m) < 0$ and $f(u) > 0$ have different signs, we continue our search in $[m, u]$

We set $l \leftarrow m$

And the search continues until we are satisfied with the precision

Bisection method

Here are the basic steps:

1. Start with a lower ($l = a$) and an upper ($u = b$) bounds
2. Get the midpoint $m = (u + l)/2$
3. Check the sign of $f(m)$
    1. If $sign(f(m)) = sign(f(l))$, move lower bound up: $l \leftarrow m$
    2. If $sign(f(m)) = sign(f(u))$, move upper bound down: $u \leftarrow m$
4. Repeat 2 and 3 until our interval is short enough ($(u - l)/2 < tol$) and return $x = m$

Your turn! Write a function that takes `(f, a, b)` and returns a root of f using the bisection method. Then use it to find the root of $f(x) = -x^{-2} + x - 1$ between $[0.2, 4]$

Bisection method

```
function bisection(f, lo, up)
    tolerance = 1e-3           # tolerance for solution
    mid = (lo + up)/2          # initial guess, bisect the interval
    difference = (up - lo)/2   # initialize bound difference

    while difference > tolerance          # loop until convergence
        println("Intermediate guess: $mid")
        if sign(f(lo)) == sign(f(mid)) # if the guess has the same sign as the lower bound
            lo = mid                   # a solution is in the upper half of the interval
            mid = (lo + up)/2
        else                           # else the solution is in the lower half of the interval
            up = mid
            mid = (lo + up)/2
        end
        difference = (up - lo)/2       # update the difference
    end
    println("The root of f(x) is $mid")
end;
```

Bisection method

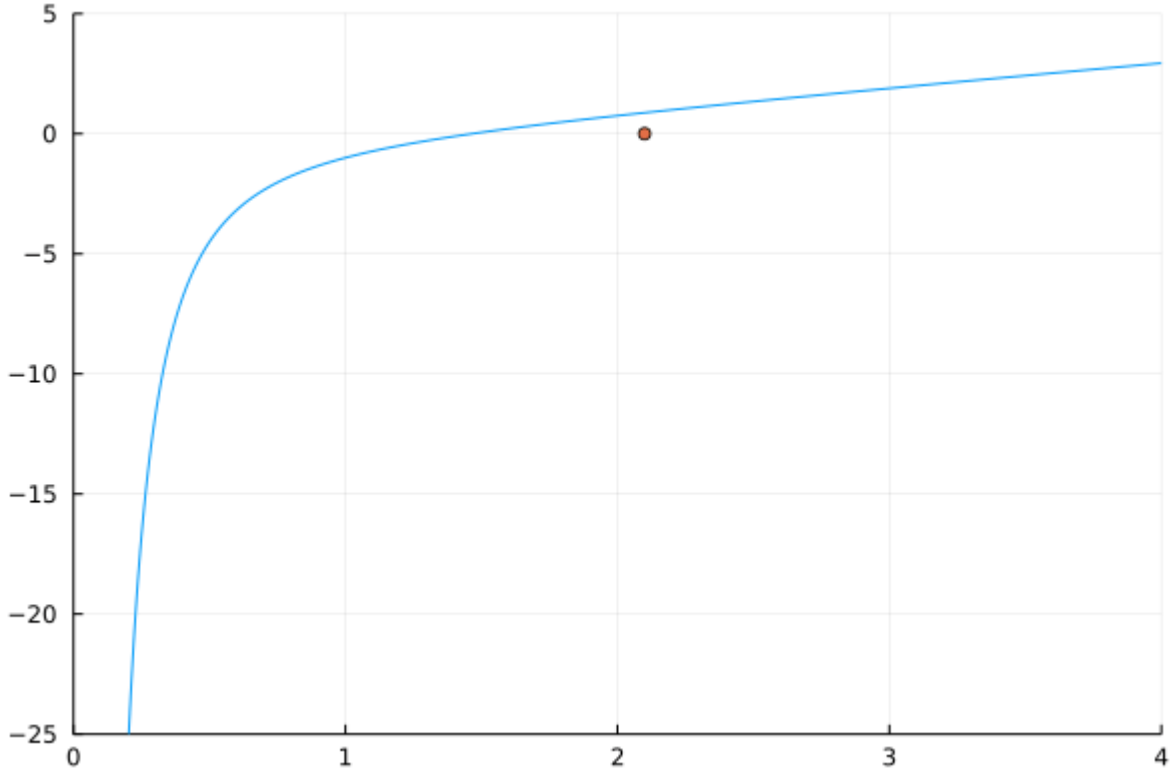
```
f(x) = -x^(-2) + x - 1
```

```
## f (generic function with 1 method)
```

```
bisection(f, 0.2, 4.0)
```

```
## Intermediate guess: 2.1
## Intermediate guess: 1.1500000000000001
## Intermediate guess: 1.625
## Intermediate guess: 1.3875000000000002
## Intermediate guess: 1.50625
## Intermediate guess: 1.4468750000000001
## Intermediate guess: 1.4765625
## Intermediate guess: 1.4617187500000002
## Intermediate guess: 1.469140625
## Intermediate guess: 1.4654296875000001
## Intermediate guess: 1.46728515625
## The root of f(x) is 1.4663574218750002
```

Bisection method



Bisection method

What happens if we specify the wrong interval (i.e, there's no root in there)?

It will go towards the boundaries

```
bisection(f, 2.0, 4.0)
```

```
## Intermediate guess: 3.0
## Intermediate guess: 3.5
## Intermediate guess: 3.75
## Intermediate guess: 3.875
## Intermediate guess: 3.9375
## Intermediate guess: 3.96875
## Intermediate guess: 3.984375
## Intermediate guess: 3.9921875
## Intermediate guess: 3.99609375
## Intermediate guess: 3.998046875
## The root of f(x) is 3.9990234375
```

Bisection method

So it's a good idea to check if you have the right boundaries

```
f(2.0)
```

```
## 0.75
```

```
f(4.0)
```

```
## 2.9375
```

These are both positive. So by the IVT, we can't know for sure if there's a root here

Bisection method

The bisection method is incredibly robust: if a function f satisfies the IVT, it is **guaranteed to converge in a finite number of iterations**

A root can be calculated to arbitrary precision ϵ in a maximum of $log_2 \frac{b-a}{\epsilon}$ iterations

But robustness comes with drawbacks:

1. It only works in one dimension
2. It is slow because it only uses information about the function's level but not its variation

Function iteration

For the next method, we recast rootfinding as a fixed-point problem

$$f(x) = 0 \Rightarrow g(x) = x - f(x)$$

Then, we start with an initial guess $x^{(0)}$ and iterate $x^{(k+1)} \leftarrow g(x^k)$ until convergence: $|x^{(k+1)} - x^{(k)}| \approx 0$

Your turn again!

- Write a function that takes `(f, initial_guess)` and returns a root of f using function iteration
- Then use it to find the root of $f(x) = -x^{-2} + x - 1$ with `initial_guess = 1`

Function iteration

```
function function_iteration(f, initial_guess)
    tolerance = 1e-3   # tolerance for solution
    difference = Inf   # initialize difference
    x = initial_guess  # initialize current value

    while abs(difference) > tolerance # loop until convergence
        println("Intermediate guess: $x")
        x_prev = x  # store previous value
        x = x_prev - f(x_prev) # calculate next guess
        difference = x - x_prev # update difference
    end
    println("The root of f(x) is $x")
end;
```

Function iteration

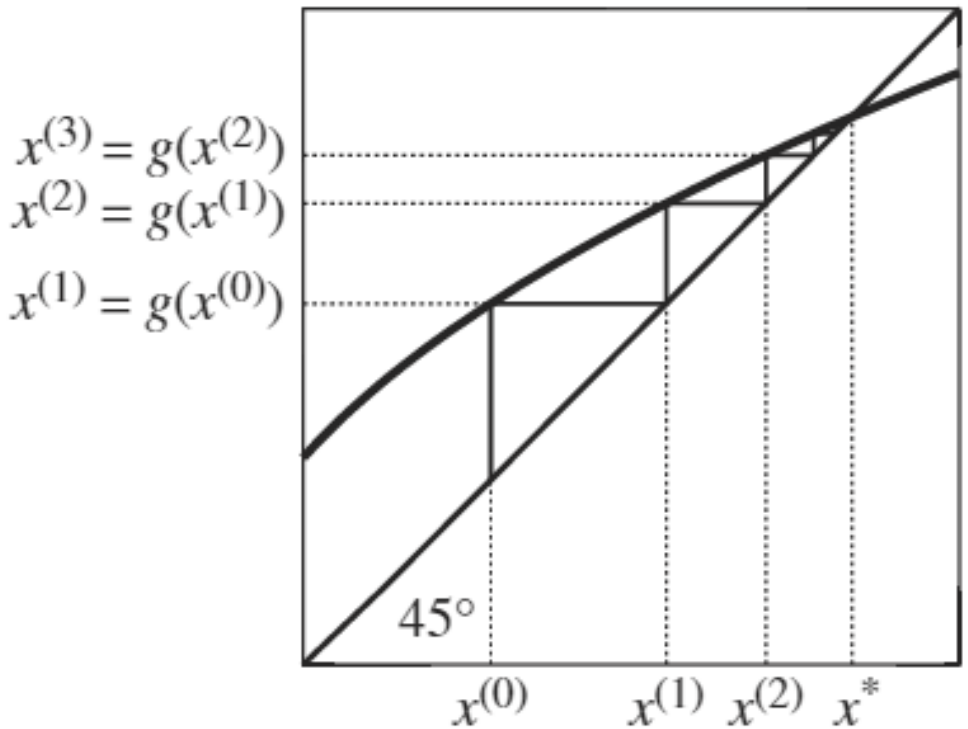
```
f(x) = -x^(-2) + x - 1;
function_iteration(f, 1.0)
```

```
## Intermediate guess: 1.0
## Intermediate guess: 2.0
## Intermediate guess: 1.25
## Intermediate guess: 1.6400000000000001
## Intermediate guess: 1.37180249851279
## Intermediate guess: 1.5313942135189396
## Intermediate guess: 1.426408640598956
## Intermediate guess: 1.4914870486759138
## Intermediate guess: 1.4495324290188554
## Intermediate guess: 1.475931147477801
## Intermediate guess: 1.4590582576091302
## Intermediate guess: 1.4697369615928917
## Intermediate guess: 1.4629358005751474
## Intermediate guess: 1.467250165675962
## Intermediate guess: 1.46450636089898
## Intermediate guess: 1.4662485297695576
## Intermediate guess: 1.4651412125747465
## The root of f(x) is 1.4658445625216316
```

Function iteration

A fixed point $x = g(x)$ is at the intersection between $g(x)$ and the 45° line

- Starting from $x^{(0)}$, we calculate $g(x^{(0)})$ and find the corresponding point on the 45° line for $x^{(1)}$
- We keep iterating until we (approximately) find the fixed point



Function iteration

Function iteration is guaranteed to converge to a fixed point x^* if

1. g is differentiable, and
2. the initial guess is "sufficiently close" an x^* at which $\|g'(x^*)\| < 1$

It may also converge when these conditions are not met

Since this is an easy method to implement, it's worth trying it before switching to more complex methods

Function iteration

But wait: What is "sufficiently close"?

Good question! There is no practical formula. As Miranda and Fackler (2002) put it

> Typically, an analyst makes a reasonable guess for the root f and counts his/her blessings if the iterates converge. If the iterates do not converge, then the analyst must look more closely at the properties of f to find a better starting value, or change to another rootfinding method.

This is where science also becomes a bit of an art

Newton's method

Newton's method and variants are the workhorses of solving n-dimensional non-linear problems

Key idea: take a hard non-linear problem and replace it with a sequence of linear problems $\rightarrow$ **successive linearization**

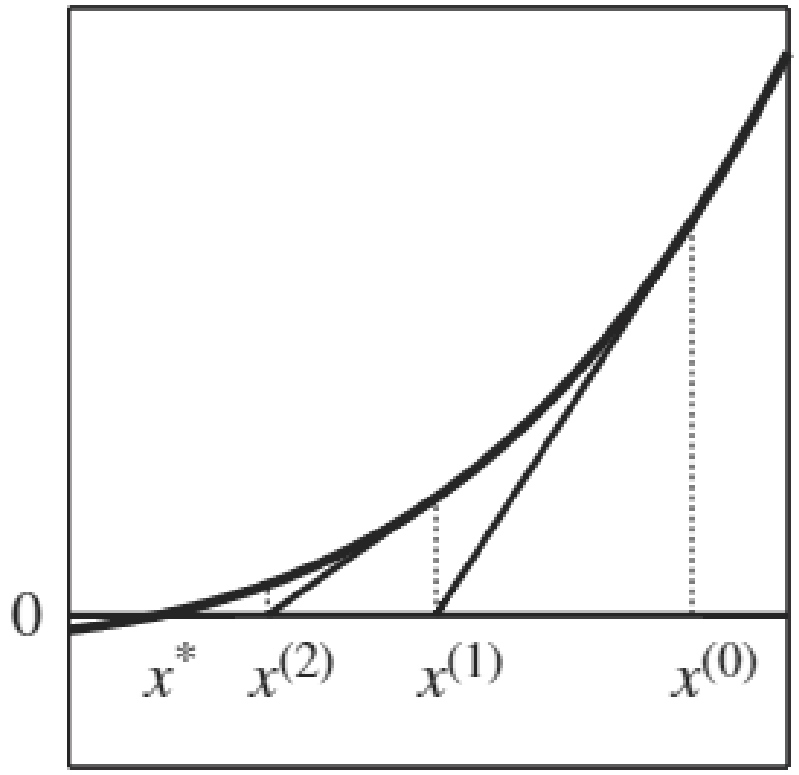
Newton's method

1) Start with an initial guess of the root at $x^{(0)}$

2) Approximate the non-linear function with its first-order Taylor expansion around $x^{(0)}$

- This is just the tangent line at x^0

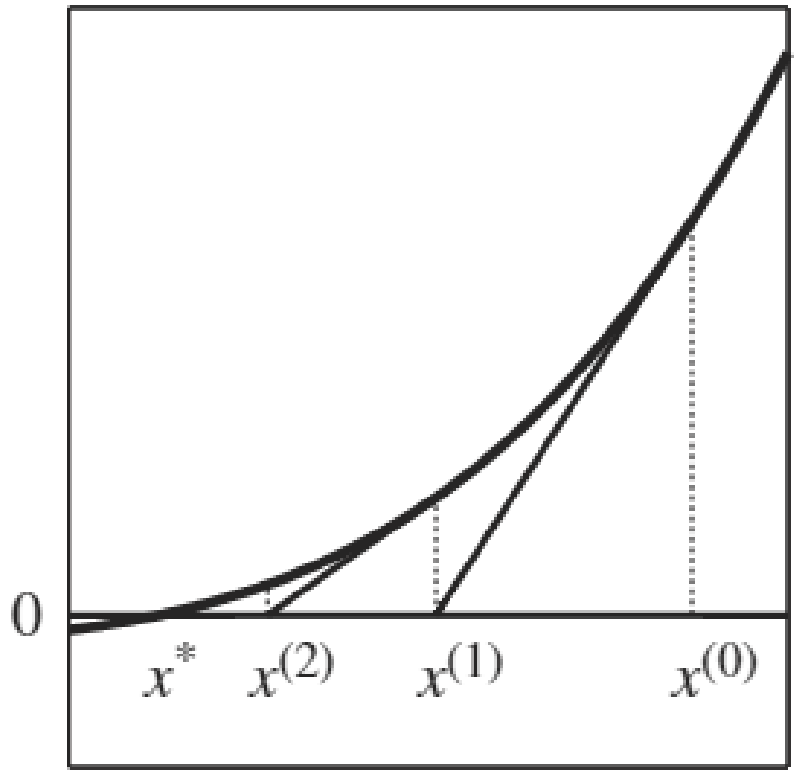
3) Solve for the root of this linear approximation, call it $x^{(1)}$



Newton's method

4) Repeat starting at $x^{(1)}$ until we converge to x^*

This can be applied to a function with an arbitrary number of dimensions



Newton's method

Formally: begin with some initial guess of the root vector $\mathbf{x^{(0)}}$

Given $\mathbf{x^{(k)}}$, our new guess $\mathbf{x^{(k+1)}}$ is obtained by approximating $f(\mathbf{x})$ using a first-order Taylor expansion around $\mathbf{x^{(k)}}$

$$f(\mathbf{x}) \approx f(\mathbf{x^{(k)}}) + f'(\mathbf{x^{(k)}})(\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}}) = 0$$

Then, solve for $\mathbf{x^{(k+1)}}$:

$$\mathbf{x^{(k+1)}} = \mathbf{x^{(k)}} - \left[f'(\mathbf{x^{(k)}})\right]^{-1} f(\mathbf{x^{(k)}})$$

Newton's method

$$\mathbf{x^{(k+1)}} \leftarrow \mathbf{x^{(k)}} - \left[f'(\mathbf{x^{(k)}})\right]^{-1} f(\mathbf{x^{(k)}})$$

Yep, your turn once again!

- Write a function that takes `(f, f_prime, initial_guess)` and returns a root of f using Newton's method
- Then use it to find the root of $f(x) = -x^{-2} + x - 1$ with `initial_guess = 1`

Newton's method

```
function newtons_method(f, f_prime, initial_guess)
  tolerance = 1e-3   # tolerance for solution
  difference = Inf   # initialize difference
  x = initial_guess  # initialize current value

  while abs(difference) > tolerance # loop until convergence
      println("Intermediate guess: $x")
      x_prev = x  # store previous value
      x = x_prev - f(x_prev)/f_prime(x_prev) # calculate next guess
      # ^ this is the only line that changes from function iteration
      difference = x - x_prev # update difference
  end
  println("The root of f(x) is $x")
end;
```

Newton's method

```
f(x) = -x^(-2) + x - 1;
f_prime(x) = 2x^(-1) + 1;
newtons_method(f, f_prime, 1.0)
```

```
## Intermediate guess: 1.0
## Intermediate guess: 1.3333333333333333
## Intermediate guess: 1.425
## Intermediate guess: 1.453066974004354
## Intermediate guess: 1.4617151894575402
## Intermediate guess: 1.4643819878673852
## The root of f(x) is 1.4652044470426357
```

Newton's method

Newton's method has nice properties regarding convergence and speed

It converges if

1. If $f(x)$ is continuously differentiable,
2. The initial guess is "sufficiently close" to the root, and
3. $f(x)$ is invertible near the root

We need $f(x)$ to be invertible so the algorithm above is well defined

If $f'(x)$ is ill-conditioned we can run into problems with rounding error

Newton's method: a duopoly example

Inverse demand: $P(q) = q^{-1/\eta}$

Two firms with costs: $C_i(q_i) = \frac{1}{2} c_i q_i^2$

Firm i's profits: $\pi_i(q_1, q_2) = P(q_1 + q_2) q_i - C_i(q_i)$

Firms take other's output as given. So their first-order conditions are

$$\frac{\partial \pi_i}{\partial q_i} = P(q_1 + q_2) + P'(q_1 + q_2) q_i - C_i'(q_i) = 0$$

Newton's method: a duopoly example

We are looking for an equilibrium: a pair (q_1, q_2) which are roots to two nonlinear equations

$$f_1(q_1, q_2) = (q_1 + q_2)^{-1/\eta} - (1/\eta)(q_1 + q_2)^{-1/\eta - 1} q_1 - c_1 q_1 = 0$$
$$f_2(q_1, q_2) = (q_1 + q_2)^{-1/\eta} - (1/\eta)(q_1 + q_2)^{-1/\eta - 1} q_2 - c_2 q_2 = 0$$

Can you solve this analytically? It's quite hard...

Let's do it numerically, starting by coding this function in Julia

Newton's method: a duopoly example

For this example, $\eta = 1.6, c_1 = 0.6, c_2 = 0.8$

```
eta = 1.6; c = [0.6; 0.8]; # column vector
```

```
## 2-element Vector{Float64}:
##  0.6
##  0.8
```

$f(q)$ will return a vector (*pay attention to how I used the dot syntax*)

```
function f(q)
    Q = sum(q)
    F = Q^(-1/eta) .- (1/eta)Q^(-1/eta-1) .*q .- c .*q
end;
f([0.2; 0.2])
```

```
## 2-element Vector{Float64}:
##  1.0989480808247896
##  1.0589480808247895
```

Newton's method: a duopoly example

What do we need next to use Newton's method?

The derivatives! In this case, we have a function $f : \mathbb{R}^2 \to \mathbb{R}^2$, so we need to define the Jacobian matrix

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial q_1} & \frac{\partial f_1}{\partial q_2} \\ \frac{\partial f_2}{\partial q_1} & \frac{\partial f_2}{\partial q_2} \end{bmatrix}$$

Newton's method: a duopoly example

We can derive these terms analytically

$$\frac{\partial f_i}{\partial q_i} = (1/\eta)(q_1+q_2)^{-1/\eta-1}\left[-2+(1/\eta+1)(q_1+q_2)^{-1}q_i\right]-c_i$$

$$\frac{\partial f_i}{\partial q_{j\neq i}} = \underbrace{(1/\eta)(q_1+q_2)^{-1/\eta-1}}_{A}\left[1+\underbrace{(1/\eta+1)(q_1+q_2)^{-1}}_{B}q_i\right]$$

Newton's method: a duopoly example

So we can write the Jacobian as

$$J = A \begin{bmatrix} -2 & 1 \\ 1 & -2 \end{bmatrix} + AB \begin{bmatrix} q_1 & q_1 \\ q_2 & q_2 \end{bmatrix} - \begin{bmatrix} c_1 & 0 \\ 0 & c_2 \end{bmatrix}$$

where $A \equiv (1/\eta)(q_1 + q_2)^{-1/\eta - 1}$, $B \equiv (1/\eta)(q_1 + q_2)^{-1}$

Let's turn this Jacobian into a Julia function so we can use Newton's method

Newton's method: a duopoly example

Now we define a function for the Jacobian

```
using LinearAlgebra
function f_jacobian(q)
    Q = sum(q)
    A = (1/eta)Q^(-1/eta-1)
    B = (1/eta)Q^(-1)
    J = A .* [-2 1; 1 -2] + (A*B) .* [q q] - LinearAlgebra.Diagonal(c)
end;
f_jacobian([0.2; 0.2])
```

```
## 2×2 Matrix{Float64}:
##  -5.27494   3.63607
##   3.63607  -5.47494
```

Newton's method: a duopoly example

We need to write a new version of our Newton's method so it can handle n-dimensional functions

```
function newtons_method_multidim(f, f_jacobian, initial_guess)
    tolerance = 1e-3
    difference = Inf
    x = initial_guess

    while norm(difference) > tolerance # <=== Changed here
        println("Intermediate guess: $x")
        x_prev = x
        x = x_prev - f_jacobian(x_prev)\f(x_prev) # <=== and here
        difference = x - x_prev
    end
    println("The root of f(x) is $x")
    return x
  end;
```

Newton's method: a duopoly example

Note that we only needed to change 2 things from the previous version:

1. Our tolerance is now over the norm of the difference vector
2. The "derivative" is a Jacobian matrix, so we multiply $f(x^{(k)})$ by the inverse of $J(x^{(k)})$
   - We use operator `\` because it is more efficient than inverting J

We also added a `return x` so that the function returns a solution

Newton's method: a duopoly example

Let's test it out with initial guess $(0.2, 0.2)$

```
x = newtons_method_multidim(f, f_jacobian, [0.2; 0.2])
```

```
## Intermediate guess: [0.2, 0.2]
## Intermediate guess: [0.8301210057769877, 0.811898634417584]
## Intermediate guess: [0.7911782905685016, 0.6885273839312711]
## Intermediate guess: [0.8384483524888013, 0.7064631551302145]
## Intermediate guess: [0.8323002559883395, 0.6888379457907813]
## Intermediate guess: [0.8394024229374498, 0.6913993149610369]
## Intermediate guess: [0.8384895974449257, 0.6888064663887703]
## Intermediate guess: [0.8395422926034856, 0.6891813928622154]
## The root of f(x) is [0.839407991494229, 0.6887983144711607]

## 2-element Vector{Float64}:
##  0.839407991494229
##  0.6887983144711607
```

Newton's method: a duopoly example

Let's check our solution

```
f(x)
```

```
## 2-element Vector{Float64}:
##  0.00015115521619180 06
##  1.1144904443316506e-5
```

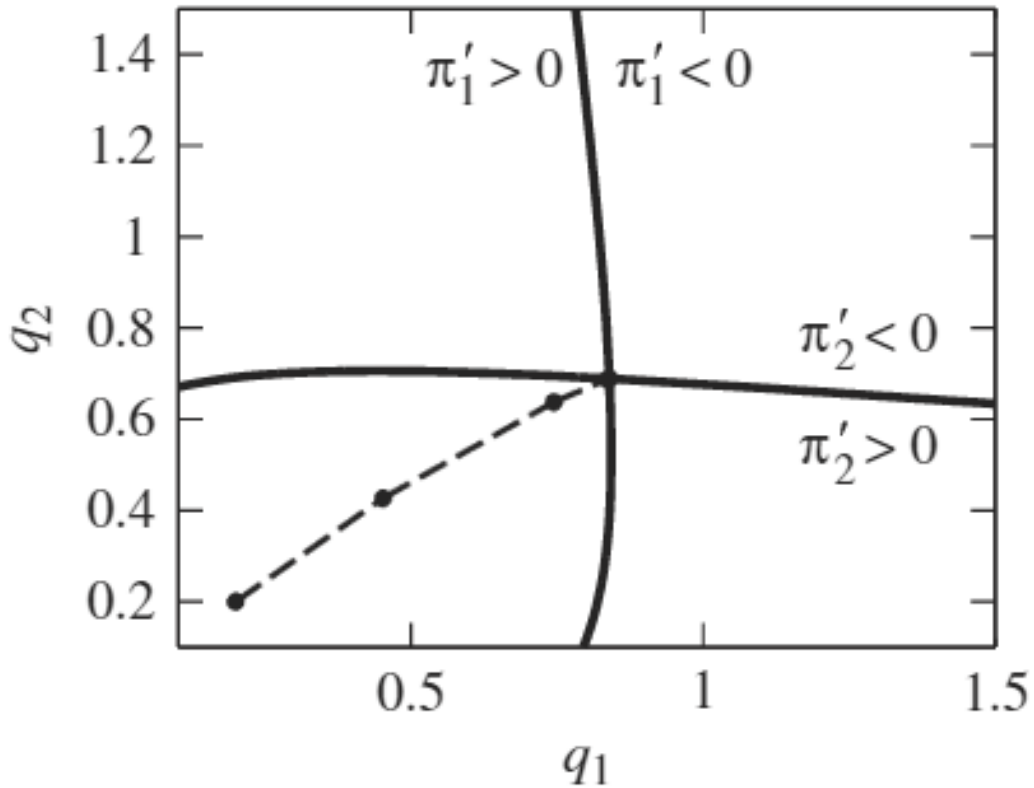
Pretty good result with quick convergence!

It was tedious but in the previous example we could calculate the Jacobian analytically. Sometimes it's much harder and we can make mistakes.

Here is a challenge for you: redo the previous example but, instead of defining the Jacobian analytically, use the ForwardDiff package to do the derivatives for you. Then, compare your solution with mine

Newton's method: a duopoly example

Visually, this is the path Newton's method followed



Quasi-Newton methods

We usually don't want to deal with analytic derivatives unless we have access to autodifferentiation

Why?

1. It can be difficult to do the analytic derivation
2. Coding a complicate Jacobian is prone to errors and takes time
3. Can actually be slower to evaluate than finite differences for a nonlinear problem

Alternative $\rightarrow$ *finite differences* instead of analytic derivatives

Quasi-Newton: Secant method

Using our current root guess $x^{(k)}$ and our previous root guess $x^{(k-1)}$:

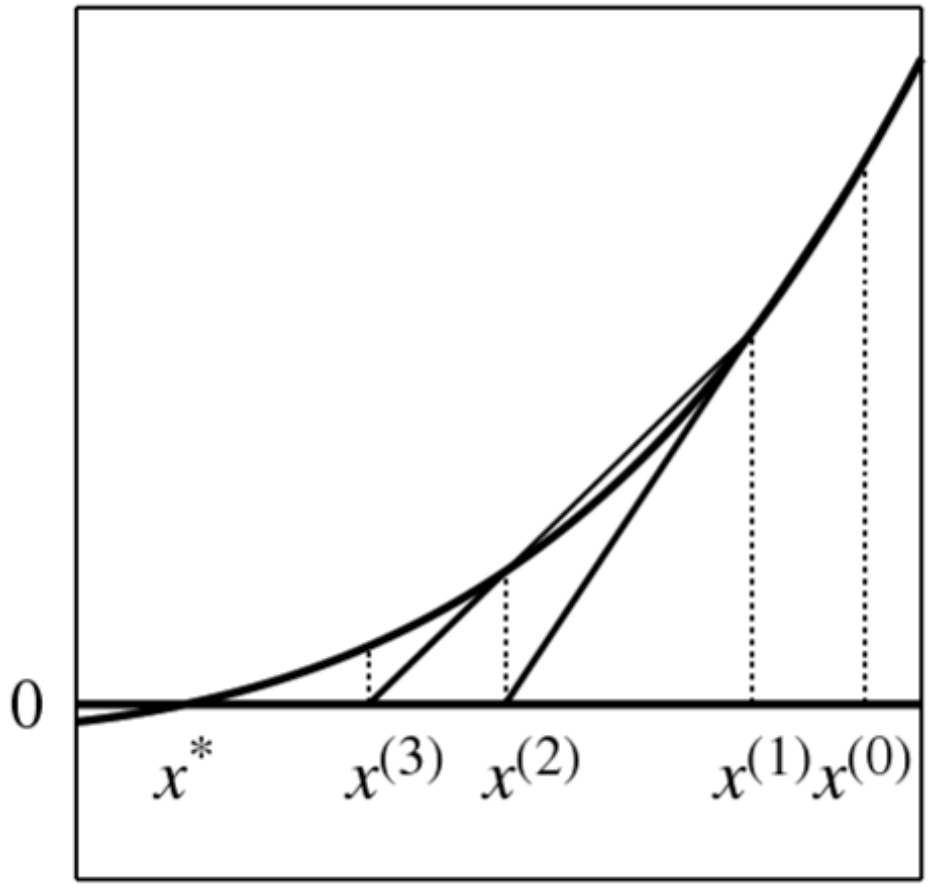
$$f'(x^{(k)}) \approx \frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}$$

Our new iteration rule then becomes

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - x^{(k-1)}}{f(x^{(k)}) - f(x^{(k-1)})} f(x^{(k)})$$

Now we **require two initial guesses** so that we have an initial approximation of the derivative

Quasi-Newton: Secant method



Quasi-Newton: Broyden's method

Broyden's method is the most widely used rootfinding method for n-dimensional problems

- This is a generalization of the secant method where have a *sequence of guesses of the Jacobian at the root*

It also relies on a 1st-order Taylor expansion about $\mathbf{x}$, but now in n dimensions

$$f(\mathbf{x}) \approx f(\mathbf{x^{(k)}}) + A^{(k)}(\mathbf{x} - \mathbf{x^{(k)}}) = 0$$

We must initially provide a guess of the root, $x^{(0)}$, but also a guess of the Jacobian, $A_{(0)}$

- A good guess for $A_{(0)}$ is to calculate it numerically at our chosen $x^{(0)}$

Quasi-Newton: Broyden's method

The iteration rule is the same as before but with our guess of the Jacobian substituted in for the actual Jacobian (or the finite difference approximation)

$$\mathbf{x^{(k+1)}} \leftarrow \mathbf{x^{(k)}} - (A^{(k)})^{-1} f(\mathbf{x^{(k)}})$$

We still need to update $A_{(k)}$: the ideia of Broyden's method is to choose a new Jacobian that satisfies the multidimensional **secant condition**

$$f(\mathbf{x^{(k+1)}}) - f(\mathbf{x^{(k)}}) = A^{(k+1)} \left(\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}} \right)$$

- Any reasonable guess for the Jacobian should satisfy this condition

But this gives n conditions with n^2 elements to solve for in A

Quasi-Newton: Broyden's method

Broyden's methods solves this under-determined problem with an assumption that focuses on the direction we are most interested in: $d^{(k)} = \mathbf{x^{(k+1)}} - \mathbf{x^{(k)}}$

For any direction q orthogonal to $d^{(k)}$, it assumes that $A^{(k+1)}q = A^{(k)}q$

- In other words, our next guess is as good as the current ones for any changes in x that are orthogonal to the one we are interested right now

Jointly, the secant condition and the orthogonality assumption give the iteration rule for the Jacobian:

$$A^{(k+1)} \leftarrow A^{(k)} + \left[f(\mathbf{x^{(k+1)}}) - f(\mathbf{x^{(k)}}) - A^{(k)}d^{(k)}\right] \frac{d^{(k)T}}{d^{(k)T}d^{(k)}}$$

Quasi-Newton: Broyden's method

The general algorithm for Broyden's method is:

1. Choose an initial guess for the root $\mathbf{x}^{(0)}$ and the Jacobian $A^{(0)}$
2. Calculate the next guess for the root: $\mathbf{x}^{(k+1)}$
3. Calculate the next guess for the Jacobian: $A^{(k+1)}$
4. Repeat 2 and 3 until convergence: $||\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}|| \leq tolerance$

Quasi-Newton: Broyden's method

One costly part of Broyden's method is that we update the Jacobian and "invert" it every iteration

But, since we only need the Jacobian's inverse to update $\mathbf{x}$, we can make it faster by updating the inverse Jacobian directly

$$B^{(k+1)} = B^{(k)} + \frac{[d^{(k)} - u^{(k)}]d^{(k)T}B_{(k)}}{d^{(k)T}u^{(k)}}$$

where $u^{(k)} = B^{(k)}\left[f(\mathbf{x^{(k+1)}}) - f(\mathbf{x^{(k)}})\right]$

- Most canned implementations of Broyden's method use the inverse update rule

Quasi-Newton: Broyden's method

Broyden converges under relatively weak conditions:

1. f is continuously differentiable,
2. $x^{(0)}$ is "close" to the root of f
3. f' is invertible around the root
4. A_0 is sufficiently close to the Jacobian

Broyden's method: a duopoly example

Let's revisit our previous example but now solve it using Broyden's method

We won't code it by hand. Instead, we use package `NLsolve.jl`

Function `NLsolve.nlsolve` has a ton of options to solve nonlinear equations

Once again, we are looking a pair (q_1, q_2) which are roots to two nonlinear equations

$$f_1(q_1, q_2) = (q_1 + q_2)^{-1/\eta} - (1/\eta)(q_1 + q_2)^{-1/\eta - 1} q_1 - c_1 q_1 = 0$$
$$f_2(q_1, q_2) = (q_1 + q_2)^{-1/\eta} - (1/\eta)(q_1 + q_2)^{-1/\eta - 1} q_2 - c_2 q_2 = 0$$

Broyden's method: a duopoly example

Using our previously defined f, we run

```
using NLsolve
NLsolve.nlsolve(f, [1.0; 2.0], method=:broyden,
                xtol=:1e-8, ftol=:0.0, iterations=:1000, show_trace=:true)
```

```
Results of Nonlinear Solver Algorithm
 * Algorithm: broyden without line-search
 * Starting Point: [1.0, 2.0]
 * Zero: [0.8395676035355293, 0.6887964311629567]
 * Inf-norm of residuals: 0.000000
 * Iterations: 10
 * Convergence: true
   * |x - x'| < 1.0e-08: true
   * |f(x)| < 0.0e+00: false
 * Function Calls (f): 37
 * Jacobian Calls (df/dx): 0
```

Broyden's method: a duopoly example

- The first and second arguments are f and an initial guess
  - `nlsolve` will automatically generate a Jacobian guess for us
- `method=:broyden` tells `nlsolve` to use Broyden's methods

The other arguments are optional

- `xtol` is the convergence tolerance over x: $\|\mathbf{x^{(k+1)}} - \mathbf{x^{(k)}}\| \leq$ `xtol` (default is `0.0` meaning no criterion)
- `ftol` is the convergence tolerance over $f(x)$: $\|f(\mathbf{x^{(k+1)}}) - f(\mathbf{x^{(k)}})\| \leq$ `ftol` (default is `1e-8`)
- `iterations` set the maximum number of iterations before declaring non-convergence (default is `1000`)
- `show_trace` will print all the iterations if you set to `true` (default is `false`)

NLsolve.jl

We can use this package to solve with Newton's method as well. Here we make use of the analytic Jacobian we defined earlier

```
NLsolve.nlsolve(f, f_jacobian, [1.0; 2.0], method=:newton,
                xtol=:1e-8, ftol=:0.0, iterations=:1000)
```

```
Results of Nonlinear Solver Algorithm
 * Algorithm: Newton with line-search
 * Starting Point: [1.0, 2.0]
 * Zero: [0.8395676017146293, 0.6887964307621443]
 * Inf-norm of residuals: 0.000000
 * Iterations: 21
 * Convergence: true
   * |x - x'| < 1.0e-08: true
   * |f(x)| < 0.0e+00: false
 * Function Calls (f): 22
 * Jacobian Calls (df/dx): 22
```

Note: For Problem Set 2, you can't use nlsolve

NLsolve.jl

If you omit the Jacobian, `nlsolve` will calculate it numerically for you using centered finite differencing

```
NLsolve.nlsolve(f, [1.0; 2.0], method=:newton,
                xtol=:1e-8, ftol=:0.0, iterations=:1000)
```

```
Results of Nonlinear Solver Algorithm
 * Algorithm: Newton with line-search
 * Starting Point: [1.0, 2.0]
 * Zero: [0.83956760353566, 0.6887964311630005]
 * Inf-norm of residuals: 0.000000
 * Iterations: 5
 * Convergence: true
   * |x - x'| < 1.0e-08: true
   * |f(x)| < 0.0e+00: false
 * Function Calls (f): 6
 * Jacobian Calls (df/dx): 6
```

NLsolve.jl

You can add argument `autodiff=:forward` to use forward autodifferentiation instead of finite differences

```
NLsolve.nlsolve(f, [1.0; 2.0], method=:newton, autodiff=:forward,
                xtol=:1e-8, ftol=:0.0, iterations=:1000)
```

```
Results of Nonlinear Solver Algorithm
 * Algorithm: Newton with line-search
 * Starting Point: [1.0, 2.0]
 * Zero: [0.8395676035356598, 0.6887964311630005]
 * Inf-norm of residuals: 0.000000
 * Iterations: 5
 * Convergence: true
   * |x - x'| < 1.0e-08: true
   * |f(x)| < 0.0e+00: false
 * Function Calls (f): 6
 * Jacobian Calls (df/dx): 6
```

Convergence speed

Rootfinding algorithms will converge at different speeds in terms of the number of operations

A sequence of iterates $x^{(k)}$ is said to converge to x^* at a rate of order p if there is a constant C such that

$$|x^{(k+1)} - x^*| \leq C|x^{(k)} - x^*|^p$$

for sufficiently large k

Convergence speed

$$|x^{(k+1)} - x^*| \leq C|x^{(k)} - x^*|^p$$

- If $C < 1$ and $p = 1$: linear convergence
- If $1 < p < 2$: superlinear convergence
- If $p = 2$: quadratic convergence

The higher order the convergence rate, the faster it converges

Convergence speed

How fast do the methods we've seen converge?

- Bisection: linear rate with $C = 0.5$
- Function iteration: linear rate with $C = ||f'(x^*)||$
- Secant and Broyden: superlinear rate with $p \approx 1.62$
- Newton: $p = 2$

Convergence speed

Consider an example where $f(x) = x - \sqrt(x) = 0$

This is how the 3 main approaches converge in terms of the L^1-norm for an initial guess $x^{(0)} = 0.5$

k	Function Iteration	Broyden's Method	Newton's Method
1	2.9e-001	-2.1e-001	-2.1e-001
2	1.6e-001	3.6e-002	-8.1e-003
3	8.3e-002	1.7e-003	-1.6e-005
4	4.2e-002	-1.5e-005	-6.7e-011
5	2.1e-002	6.3e-009	0.0e+000
6	1.1e-002	2.4e-014	0.0e+000
7	5.4e-003	0.0e+000	0.0e+000
8	2.7e-003	0.0e+000	0.0e+000
9	1.4e-003	0.0e+000	0.0e+000
10	6.8e-004	0.0e+000	0.0e+000
15	2.1e-005	0.0e+000	0.0e+000
20	6.6e-007	0.0e+000	0.0e+000
25	2.1e-008	0.0e+000	0.0e+000

Choosing a solution method

Convergence rates only account for the number of iterations of the method

The steps taken in a given iteration of each solution method may vary in computational cost because of differences in the number of arithmetic operations

Although an algorithm may take more iterations to solve, each iteration may be solved faster and the overall algorithm takes less time

Choosing a solution method

- Bisection method only requires a single function evaluation during each iteration
- Function iteration only requires a single function evaluation during each iteration
- Broyden's method requires both a function evaluation and matrix multiplication
- Newton's method requires a function evaluation, a derivative evaluation, and solving a linear system

$\rightarrow$ Bisection and function iteration are usually slow

$\rightarrow$ Broyden's method can be faster than Newton's method if derivatives are costly to compute

Choosing a solution method

Besides convergence rates and algorithm speed, you should also factor development time

- Newton's method is fastest to converge
  - If deriving and programing the Jacobian is relatively simple and not too costly to compute, this method is a good choice
- If derivatives are complex, quasi-Newton methods are good candidates
- Bisection and function iteration are generally dominated options but are easy to program/debug, so they have value as a quick proof-of-concept
  - Bisection is often used in hybrid methods, such as Dekker's and Brent's. Hybrid methods select between bisection, secant, or other basic solution methods every iteration depending on a set of criteria

Complementarity problems

Complementarity problems

In these problems, we have

- A function $f : \mathbb{R}^n \to \mathbb{R}^n$
- n-vectors a and b, with $a < b$

And we are looking for an n-vector $x \in [a, b]$ such that for all $i = 1, \ldots, n$

$$x_i > a_i \Rightarrow f_i(x) \geq 0$$
$$x_i < b_i \Rightarrow f_i(x) \leq 0$$

- This formulation is usually referred to as a *Mixed Complementarity Problem* because it has an upper and lower bounds
  - Standard complementarity problems have one-sided bounds

Complementarity problems

An economic interpretation

- x is an n-dimensional vector of some economic action
- a_i and b_i are the lower and upper bounds on action i
- $f_i(x)$ is the marginal arbitrage profit of action i

There are disequilibrium profit opportunities if

1. $x_i < b_i$ and $f_i(x) > 0$ (we can increase profits by raising x_i)
2. $x_i > a_i$ and $f_i(x) < 0$ (we can increase profits by decreasing x_i)

Complementarity problems

We obtain a no-arbitrage equilibrium if and only if x solves the complementary problem $CP(f, a, b)$

We can write out the problem as finding a vector $x \in [a, b]$ that solves

$$x_i > a_i \Rightarrow f_i(x) \geq 0 \ \ \forall i = 1, \ldots, n$$
$$x_i < b_i \Rightarrow f_i(x) \leq 0 \ \ \forall i = 1, \ldots, n$$

At interior solution, the function must be precisely be zero

Corner solution at the upper bound b_i for $x_i \to f$ must be increasing in direction i

The opposite is true if we are at the lower bound

Complementarity problems

Most economic problems are complementarity problems where we are

- Looking for a root of a function (e.g. marginal profit)
- Subject to some constraint (e.g. price floors)

The Karush-Kuhn-Tucker theorem shows that x solves the constrained optimization problem ($\max_x F(x)$ subject to $x \in [a, b]$) only if it solves the complementarity problem $CP(f, a, b)$, where $f_i = \partial F / \partial x_i$

Let's use a linear f to visualize what do we mean by a solution in complementarity problems

Complementarity problems

Case 1

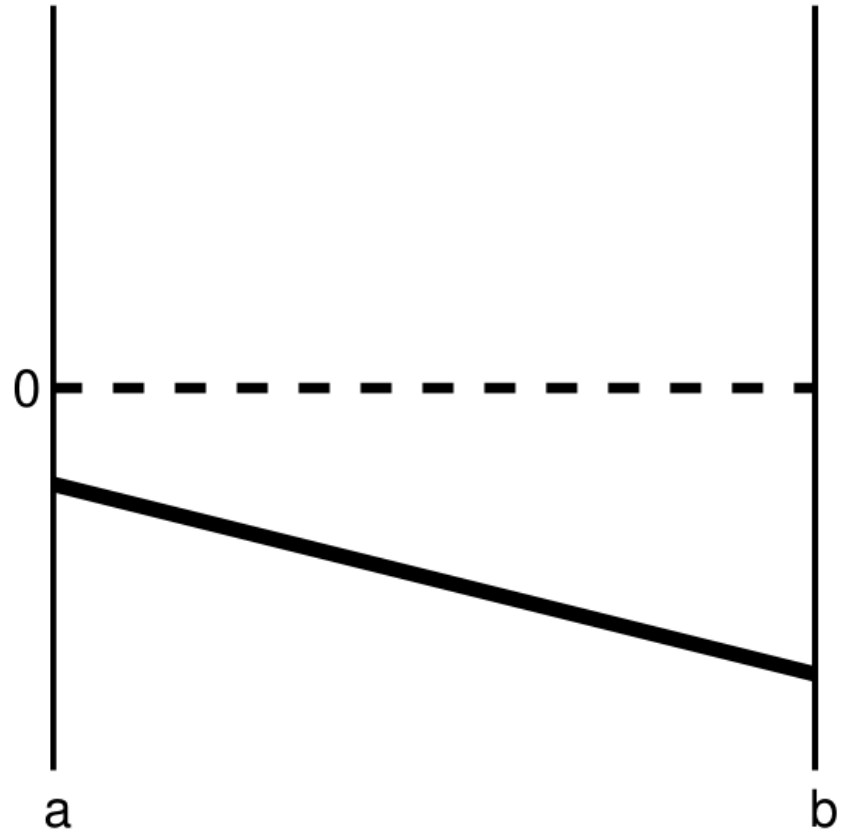
What is the solution?

$x^* = a$, with $f(x^*) < 0$

Another way of seeing it: imagine we're trying to maximize F

What would F look like between a and b?

- The decreasing part of a concave parabola



Complementarity problems

Case 2

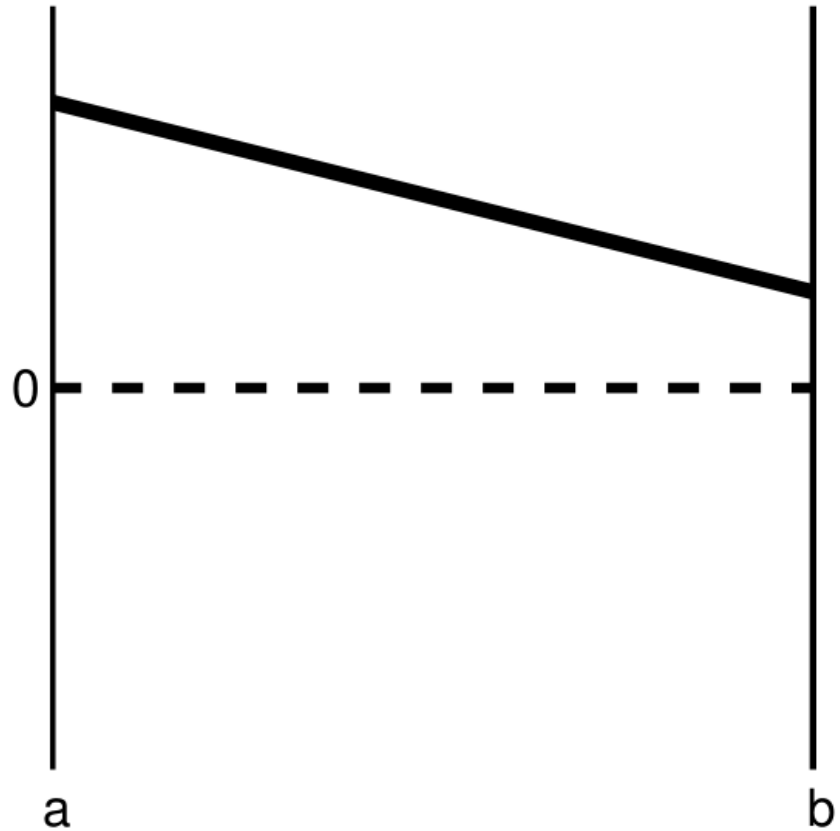
What is the solution?

$x^* = b$, with $f(x^*) > 0$

Once again, imagine we're trying to maximize F

What would F look like between a and b?

- The increasing part of a concave parabola



Complementarity problems

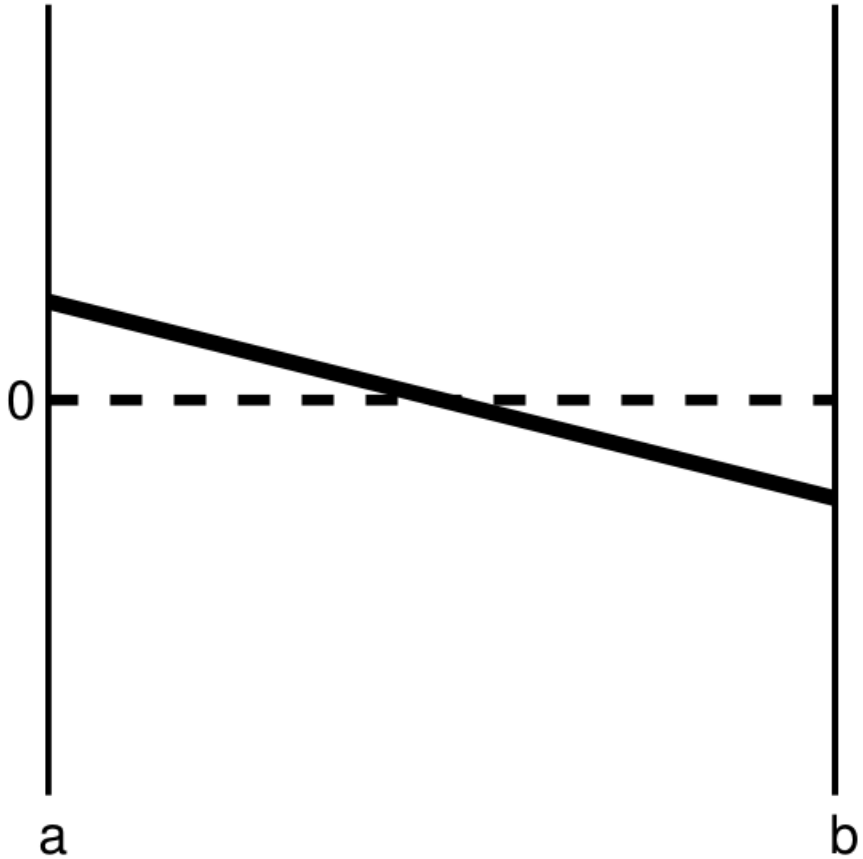
Case 3

What is the solution?

Some x^* between a and b, with $f(x^*) = 0$

What would F look like between a and b?

- A concave parabola with an interior maximum



Complementarity problems

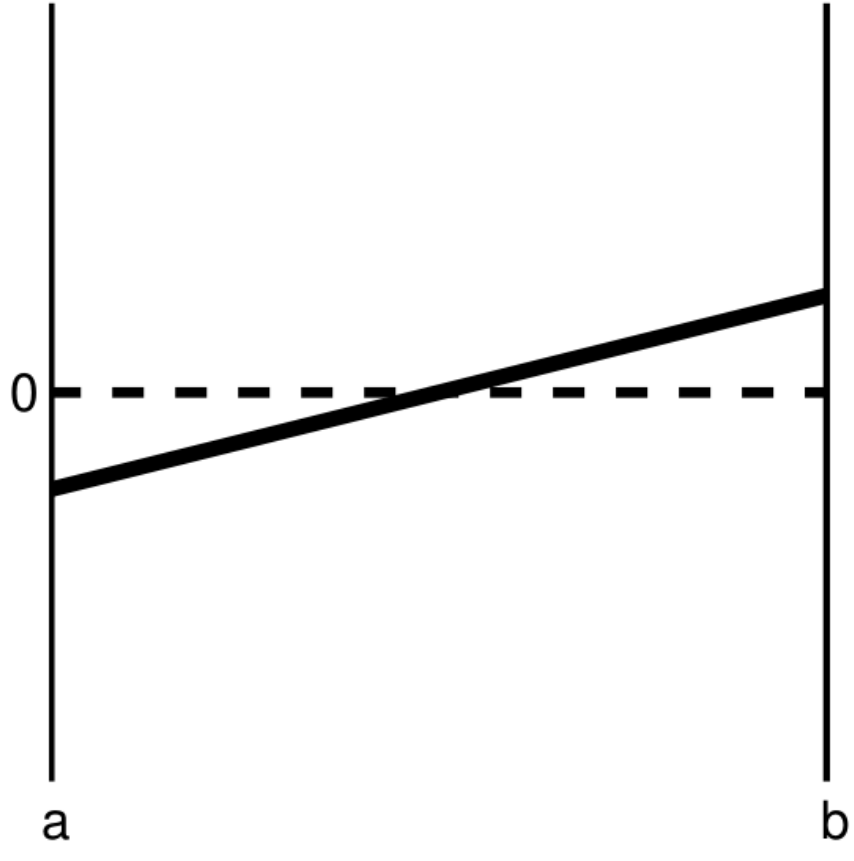
Case 4

What is the solution?

Actually, we have 3 solutions:

1. $x^* = a$, with $f(x^*) < 0$
2. $x^* = b$, with $f(x^*) > 0$
3. Some $x^* \in (a, b)$, with $f(x^*) = 0$
    - And this is an unstable solution

What would F look like between a and b?



- A convex parabola! So we end up with multiple local maxima that satisfy the 1st-order condition

Solving CPs

A complementarity problem $CP(f, a, b)$ can be re-framed as a rootfinding problem

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

Let's revisit those 4 cases to understand why this works

Solving CPs

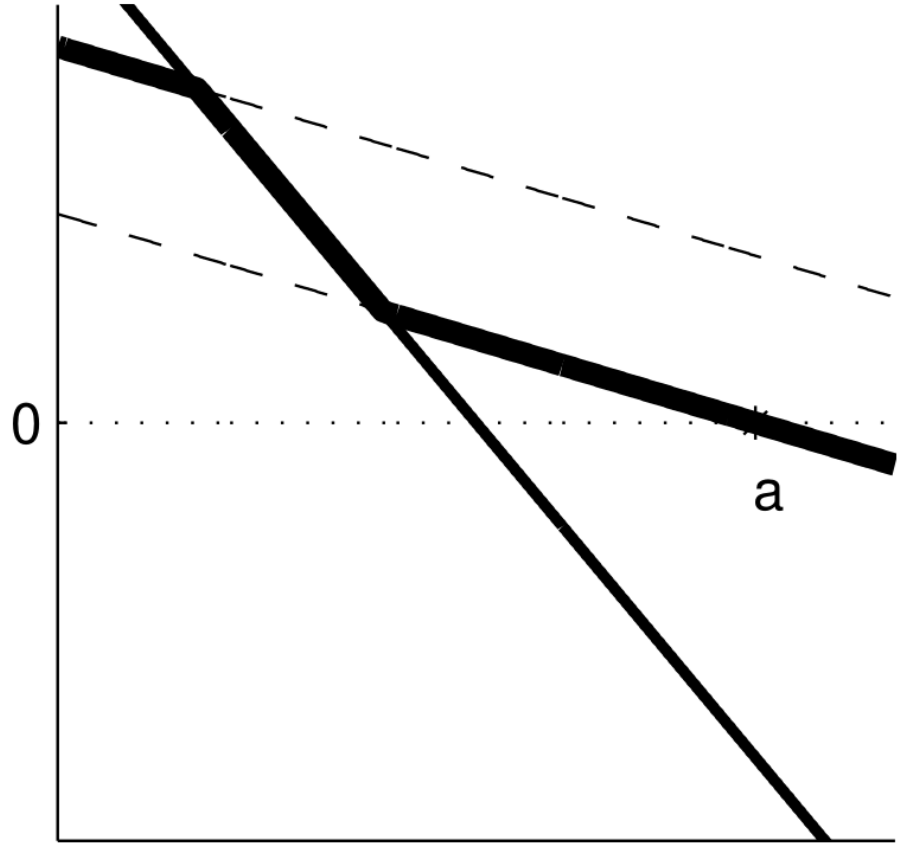
Case 1

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

- Dashed lines are $b - x$ and $a - x$
- Thin solid line is $f(x)$
- Thick solid line is $\hat{f}(x)$

$\rightarrow$ The solution is $x^* = a$

- Note that $f(x) < 0$ for all $x \in [a, b]$, so this is still case 1



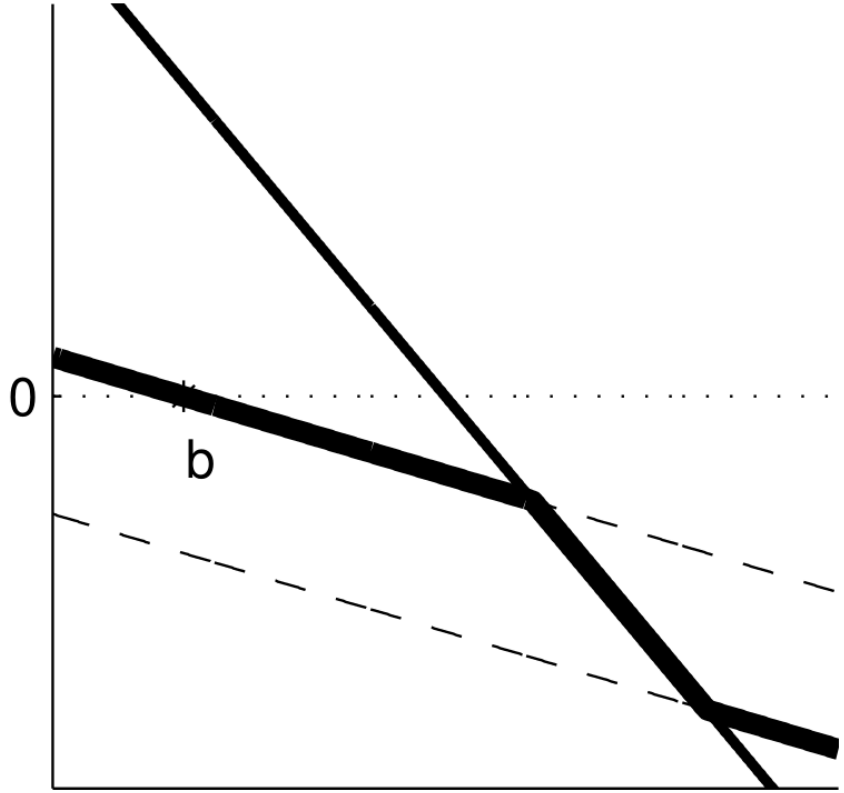
Solving CPs

Case 2:

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

In this case, $f(x) > 0$ for all $x \in [a, b]$

$\rightarrow$ The solution is $x^* = b$



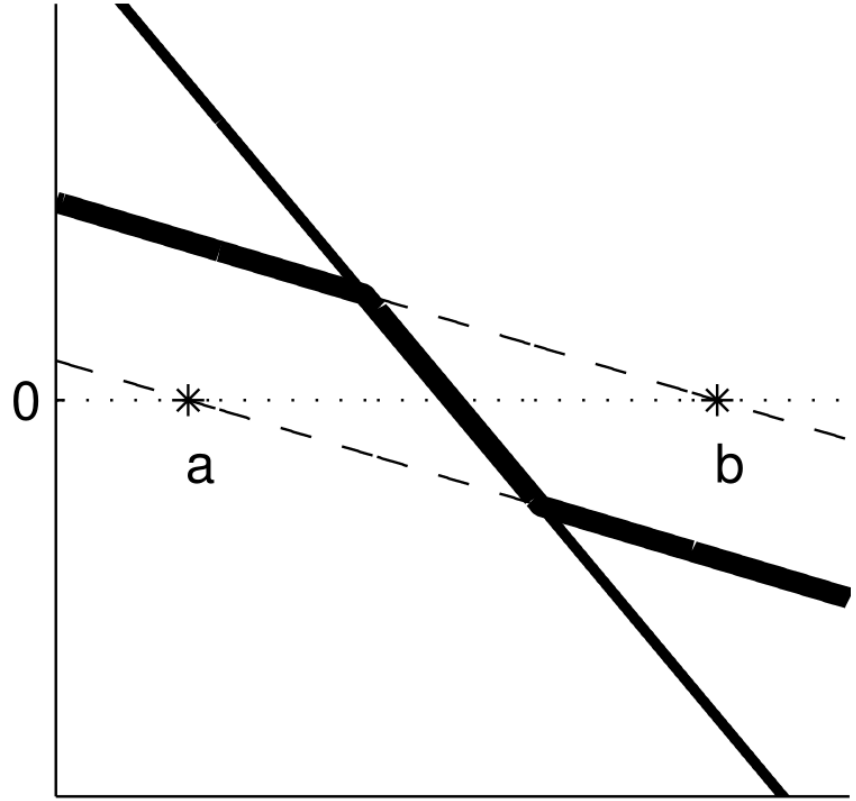
Solving CPs

Case 3:

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

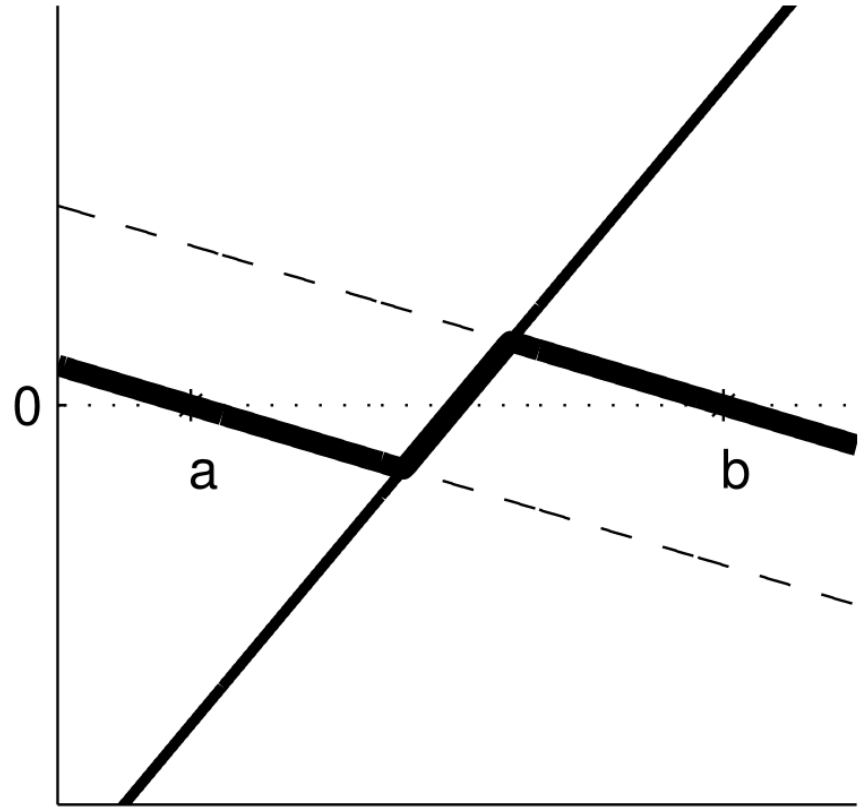
In this case, $f(a) > 0$, $f(b) < 0$

$\rightarrow$ The solution is some $x^* \in (a, b)$



Solving CPs

Case 4:



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

Solving CPs

Once $\hat{f}$ is defined, we can use Newton's or quasi-Newton methods to solve a CP

If using Newton, we need to define the Jacobian $\hat{J}(x)$ with row i being

- $\hat{J}_i(x) = J_i(x)$, if $a_i - x_i < f_i(x) < b_i - x_i$
- $\hat{J}_i(x) = -I_i(x)$, otherwise where I is the identity matrix

Solving CPs

Rootfinding with $\hat{f}$ works well in many cases but can be problematic in others

One problem is that $\hat{f}$ has nondiferentiable kinks. This can lead to slower convergence, cycles, and incorrect answers[1]

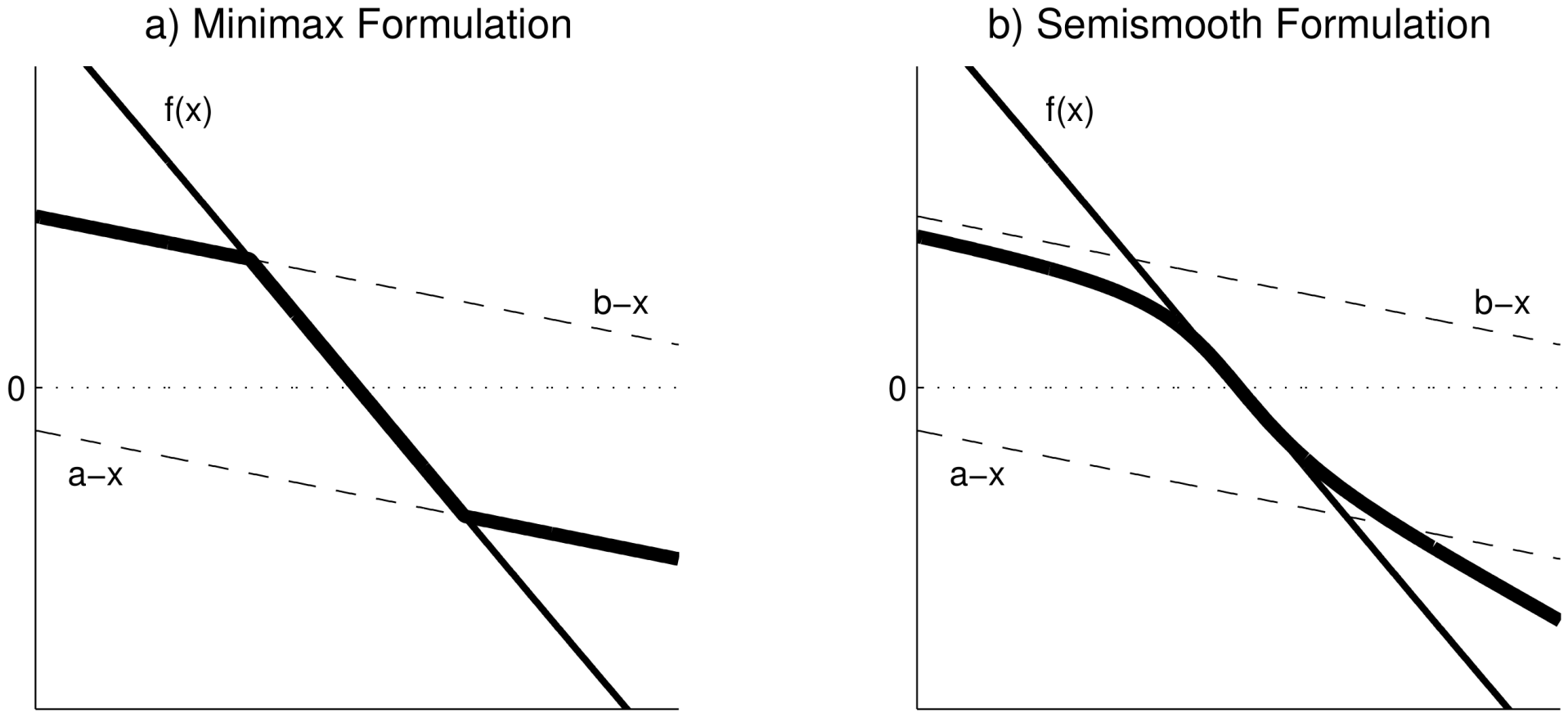
A workaround is to use an alternative function with smoother transitions, such as Fischer's function

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

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

[1]See Miranda & Fackler (2002) Ch. 3.8 for a pathological example that needs smoothing

Solving CPs


a) Minimax Formulation
f(x)
b–x
0
a–x
b) Semismooth Formulation
f(x)
b–x
0
a–x

Solving CPs in Julia

We can use `NLsolve.mcpsolve` to solve CPs for us. Let's see an example of $f : \mathbb{R}^2 \to \mathbb{R}^2$

$$f_1(x_1, x_2) = 3x_1^2 + 2x_1x_2 + 2x_2^2 + 4x_1 - 2$$
$$f_2(x_1, x_2) = 2x_1^2 + 5x_1x_2 - x_2^2 + x_2 + 1$$

```
function f(x)
    v = zeros(2)
    v[1] = 3*x[1]^2 + 2*x[1]*x[2] + 2*x[2]^2 + 4*x[1] - 2
    v[2] = 2*x[1]^2 + 5*x[1]*x[2] -   x[2]^2 + x[2] + 1
    return v
end;
```

Solving CPs in Julia

Let's check the solution to the standard rootfinding problem using Newton's method

```
using NLsolve
NLsolve.nlsolve(f, [1.0; 1.0], method=:newton)
```

```
## Results of Nonlinear Solver Algorithm
##  * Algorithm: Newton with line-search
##  * Starting Point: [1.0, 1.0]
##  * Zero: [-0.13736984903421823, 1.1872338075744586]
##  * Inf-norm of residuals: 0.000000
##  * Iterations: 7
##  * Convergence: true
##    * |x - x'| < 0.0e+00: false
##    * |f(x)| < 1.0e-08: true
##  * Function Calls (f): 8
##  * Jacobian Calls (df/dx): 8
```

Quick detour

Before we continue, take a look again at our f

```
function f(x)
    v = zeros(2)
    v[1] = 3*x[1]^2 + 2*x[1]*x[2] + 2*x[2]^2 + 4*x[1] - 2
    v[2] = 2*x[1]^2 + 5*x[1]*x[2] -   x[2]^2 + x[2] + 1
    return v
end;
```

Do you see any potential inefficiency?

We allocate a new v every time we call this function!

Instead, we can be more efficient by writing a function that modifies a pre-allocated vector

Quick detour: functions that modify arguments

By convention, in Julia we name functions that modify arguments with a `!` at the end. For our f, we can define

```
function f!(F, x)
    F[1] = 3*x[1]^2 + 2*x[1]*x[2] + 2*x[2]^2 + 4*x[1] - 2
    F[2] = 2*x[1]^2 + 5*x[1]*x[2] -   x[2]^2 + x[2] + 1
end;
F = zeros(2) # This allocates a 2-vector with elements equal to zero
```

```
## 2-element Vector{Float64}:
##  0.0
##  0.0
```

```
f!(F, [0.0; 0.0]);
F
```

```
## 2-element Vector{Float64}:
##  -2.0
##   1.0
```

Quick detour: functions that modify arguments

`NLsolve.nlsolve` understands when we pass a `!` function and pre-allocates the vector for us

Because it allocates only once, it will be more efficient

```
NLsolve.nlsolve(f!, [1.0; 1.0], method=:newton)
```

```
## Results of Nonlinear Solver Algorithm
##  * Algorithm: Newton with line-search
##  * Starting Point: [1.0, 1.0]
##  * Zero: [-0.13736984903421823, 1.1872338075744586]
##  * Inf-norm of residuals: 0.000000
##  * Iterations: 7
##  * Convergence: true
##    * |x - x'| < 0.0e+00: false
##    * |f(x)| < 1.0e-08: true
##  * Function Calls (f): 8
##  * Jacobian Calls (df/dx): 8
```

Solving CPs in Julia

Getting back to CPs, let's impose non-negativity bounds on x_1 and x_2 but no upper bounds (i.e., $b = \infty$)

```
a = [0.0; 0.0]; b = [Inf; Inf];
r = NLsolve.mcpsolve(f!, a, b, [1.0; 1.0], method=:newton, reformulation=:minmax)
```

```
## Results of Nonlinear Solver Algorithm
##  * Algorithm: Newton with line-search
##  * Starting Point: [1.0, 1.0]
##  * Zero: [0.3874258867227982, 0.0]
##  * Inf-norm of residuals: 0.000000
##  * Iterations: 10
##  * Convergence: true
##    * |x - x'| < 0.0e+00: false
##    * |f(x)| < 1.0e-08: true
##  * Function Calls (f): 17
##  * Jacobian Calls (df/dx): 10
```

Solving CPs in Julia

We can get the value of the root using `r.zero`. Checking $f(x^*)$

```
r.zero
```

```
## 2-element Vector{Float64}:
##  0.3874258867227982
##  0.0
```

```
f(r.zero)
```

```
## 2-element Vector{Float64}:
##  3.241851231905457e-14
##  1.300197635405893
```

So the non-negativity constraint is binding for x_2 but not x_1

Solving CPs in Julia

`mcpsolve` takes the same additional arguments `nlsolve` had: `xtol`, `ftol`, `iterations`, `autodiff`, `show_trace`, etc

In addition, it accepts argument `reformulation`, which can be `smooth` (default) or `minmax`

```
NLsolve.mcpsolve(f!, a, b, [1.0; 1.0], method=:newton, reformulation=:minmax, autodiff=:forward)
```

```
## Results of Nonlinear Solver Algorithm
##  * Algorithm: Newton with line-search
##  * Starting Point: [1.0, 1.0]
##  * Zero: [0.38742588672279815, 0.0]
##  * Inf-norm of residuals: 0.000000
##  * Iterations: 10
##  * Convergence: true
##    * |x - x'| < 0.0e+00: false
##    * |f(x)| < 1.0e-08: true
##  * Function Calls (f): 17
##  * Jacobian Calls (df/dx): 10
```

Example: spatial general equilibrium as CP

Let's see an example of a single-commodity competitive spatial equilibrium model

- n regions of the world
- excess demand for the commodity in region i is $E_i(p_i)$

If there's no trade $\rightarrow$ equilibrium condition is $E_i(p_i) = 0$ in all regions of the world: a simple rootfinding problem for each region

Adding trade:

- Trade between regions i and j has a unit transportation cost of c_{ij}
- x_{ij}: the amount of the good shipped from region i to region j
- Shipping is subject to capacity constraint: $0 \leq x_{ij} \leq b_{ij}$

Example: spatial general equilibrium as CP

Marginal arbitrage profit from shipping a unit of the good from i to j is

$$p_j - p_i - c_{ij}$$

- If positive: incentive to ship more goods to region i from region j
- If negative: incentive to decrease shipments

At an equilibrium, all the arbitrage opportunities are gone: for all region pairs i, j

$$0 \leq x_{ij} \leq b_{ij}$$
$$x_{ij} > 0 \Rightarrow p_j - p_i - c_{ij} \geq 0$$
$$x_{ij} < b_{ij} \Rightarrow p_j - p_i - c_{ij} \leq 0$$

Example: spatial general equilibrium as CP

How do we formulate this as a complementarity problem?

Market clearing in each region i requires that net imports = excess demand

$$\sum_k [x_{ki} - x_{ik}] = E_i(p_i)$$

This implies that we can solve for the price in region i,

$$p_i = E_i^{-1}\left(\sum_k [x_{ki} - x_{ik}]\right)$$

Example: spatial general equilibrium as CP

Finally, if we define marginal arbitrage profit from shipping another unit from i to j as

$$f_{ij}(x) = E_j^{-1}\left(\sum_k [x_{kj} - x_{jk}]\right) - E_i^{-1}\left(\sum_k [x_{ki} - x_{ik}]\right) - c_{ij}$$

then x is an equilibrium vector of trade flows if and only if x solves $CP(f, 0, b)$

- x, f, and b are $n^2 \times 1$ vectors

Even this complex trade-equilibrium model can be reduced to a relatively simple complementarity problem

Example: solving the spatial GE in Julia

See the corresponding Jupyter notebook in course materials.