L19: Root Finding

Finding r such that f(r)=0

Lucas A. J. Bastien

E7 Spring 2017, University of California at Berkeley

March 6, 2017

Version: release

Announcements

Lab 07 is due on March 10 at 12 pm (noon)

Today:

- Midterm feedback
- Root finding (*i.e.* solve $f(x) = 0$ for x)
  - Brute-force method (not recommended)
  - Bisection method
  - Newton-Raphson method
  - Matlab's built-in function `fzero`

The rest of the week:

- Wednesday: Systems of linear algebraic equations (Chapter 12)
- Friday: discussion

Need to reduce chatting and noise during lecture!

- Noise is distracting for others
- It has been getting better, thank you! Let us keep improving!

Motivation for numerical root finding

For many engineering and scientific applications, we need to solve equations of the form $f(x) = 0$ for x, where f is a real-valued function defined over an interval of $\mathbb{R}$

A number r such that $f(r) = 0$ is called a root of the function f

Sometimes, we can solve these equations analytically, for example:

$$x^2 - 2 = 0 \quad \rightarrow \quad x = -\sqrt{2} \text{ or } x = \sqrt{2}$$

Other times, it is more difficult, for example:

$$\cos(x) - x = 0 \quad \rightarrow \quad x = ???$$

Motivation for numerical root finding (continued)

One can use **numerical methods to find approximate solutions of equations of the form** $f(x) = 0$. This process is useful when:

- It is difficult to find the solutions analytically
- One wants to automate root finding for any function, without implementing the analytical solutions of all possible functions

Extremely important:
In this class, "solve" (with the quotes) means "find approximate solution(s)". Similarly, "solution" (with the quotes) means "approximate solution"

Today, we will learn three numerical methods for root finding:

- Brute-force (don't use it, but presented here for its learning value)
- Bisection
- Newton-Raphson

Intermediate value theorem

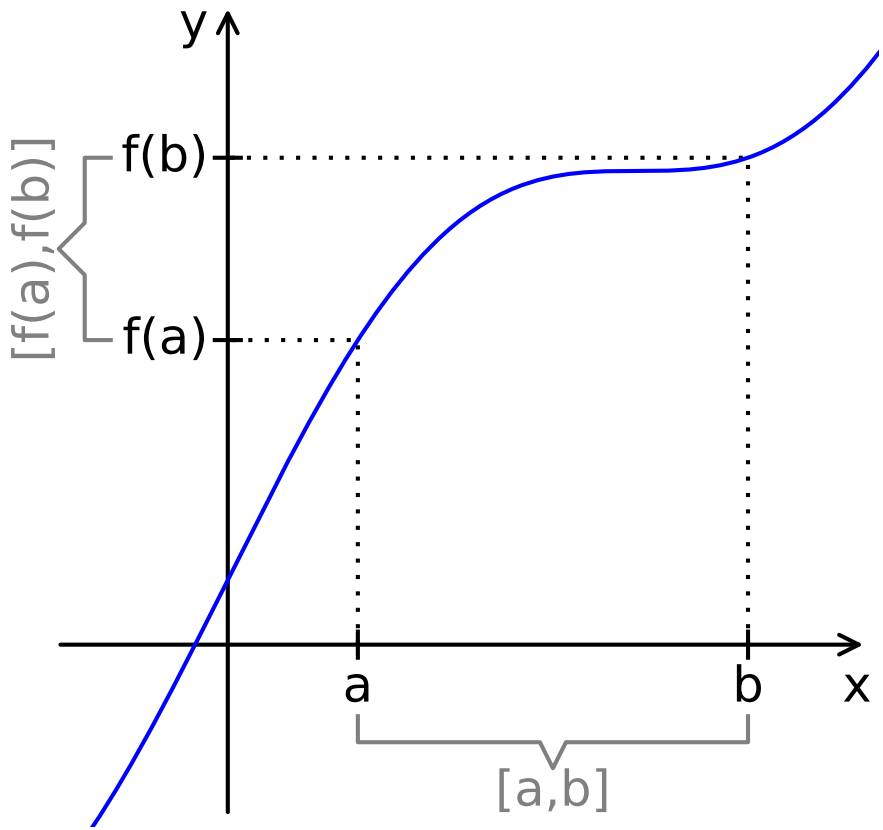
IF:

f is a **continuous** real-valued function defined on the interval $[a, b]$

THEN:

For all y in between $f(a)$ and $f(b)$, there exists at least one $x \in [a, b]$ such that $f(x) = y$

In other words: the function f takes all values between $f(a)$ and $f(b)$ over the interval $[a, b]$



Intermediate value theorem: Important corollary

IF:

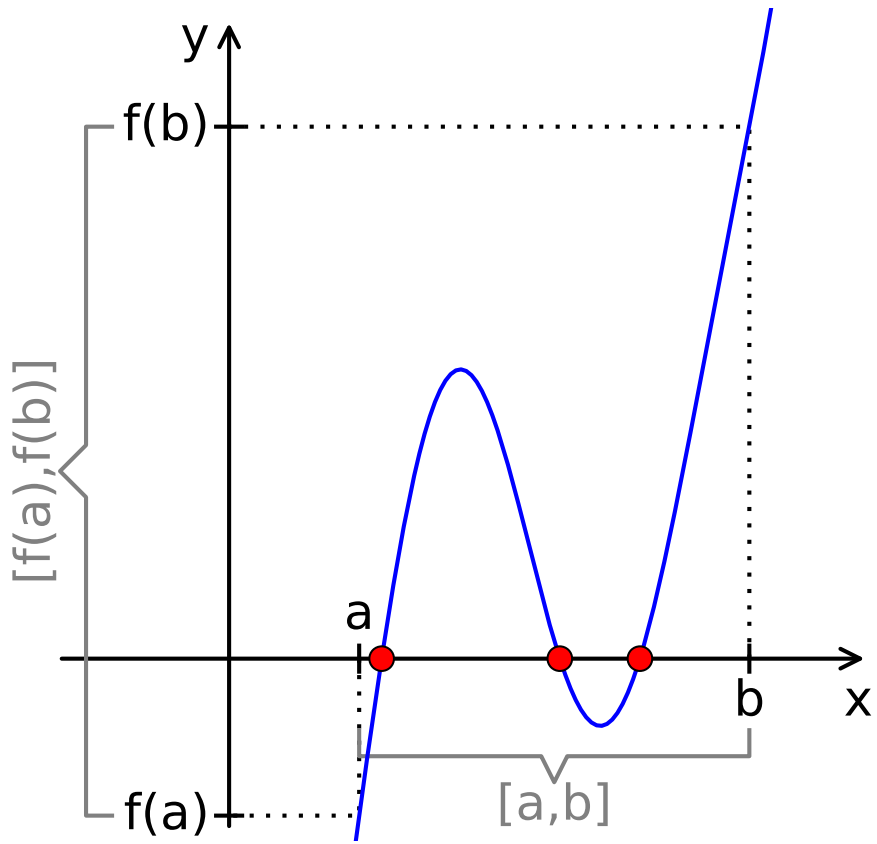
f is a **continuous** real-valued function defined on the interval $[a, b]$ and $f(a)f(b) \leqslant 0$

THEN:

There exists at least one $r \in [a, b]$ such that $f(r) = 0$

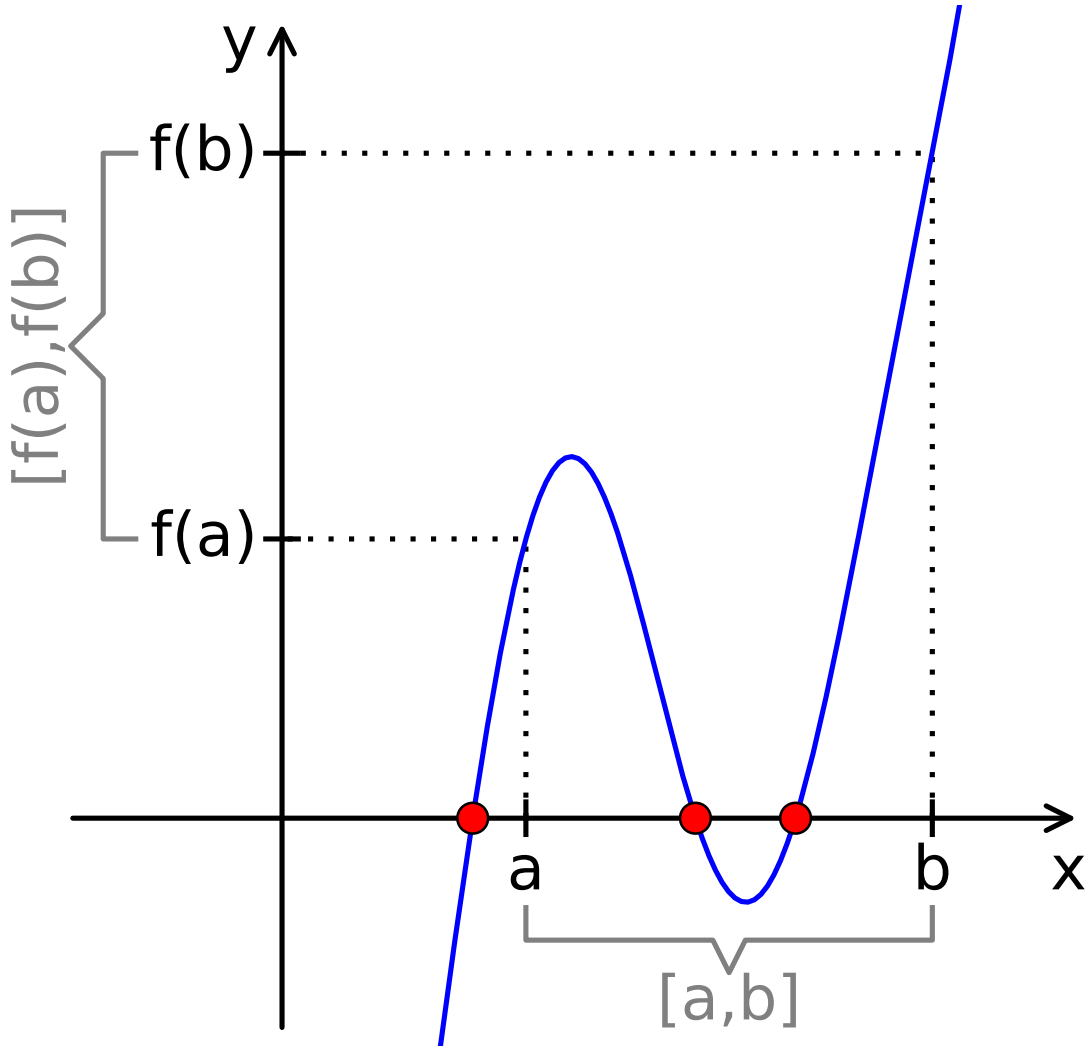
Notes:

- $f(a)f(b) < 0$ means that $f(a)$ and $f(b)$ have different signs
- $f(a)f(b) = 0$ means that $f(a) = 0$ and/or $f(b) = 0$

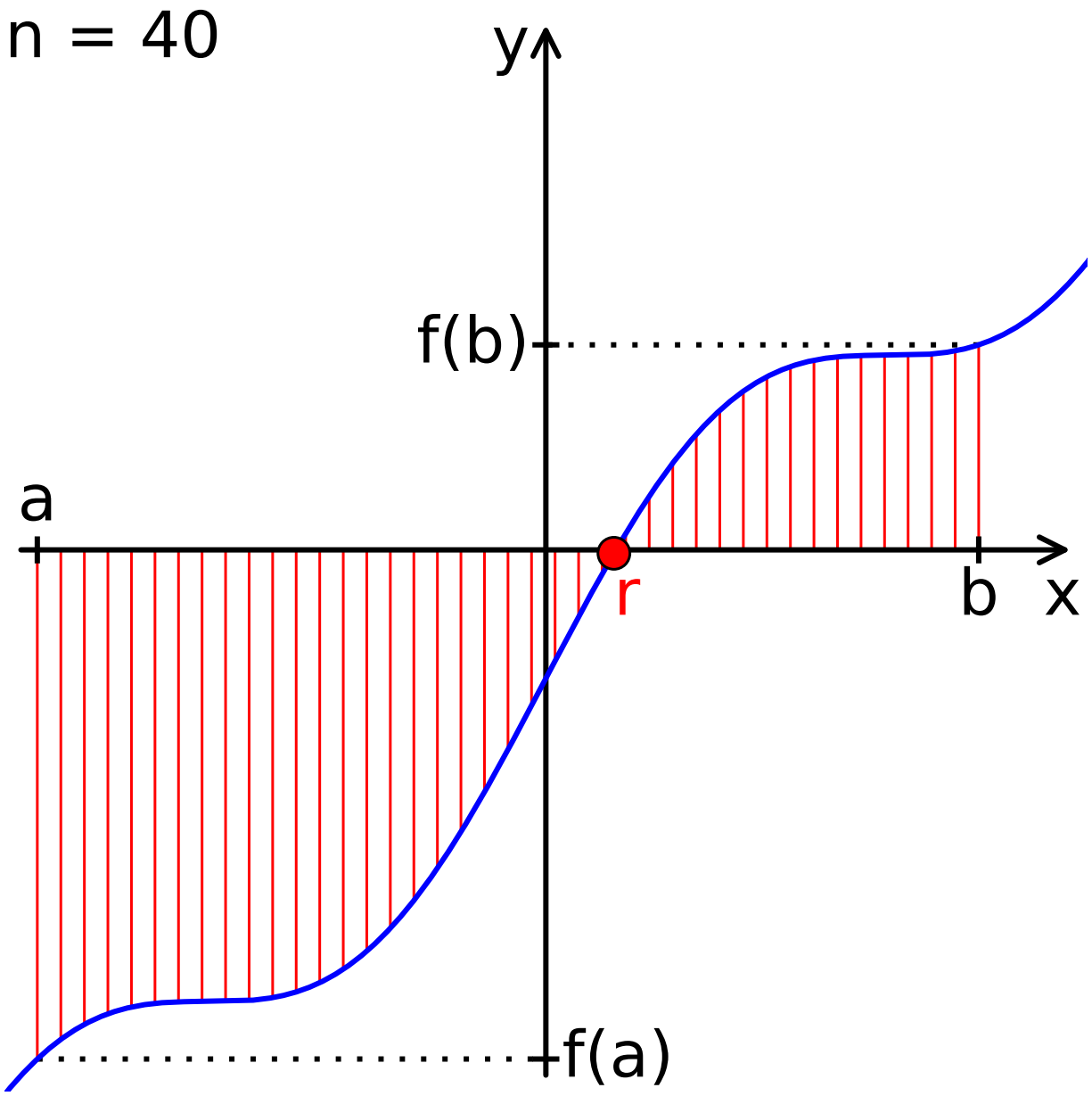


Intermediate value theorem: Important corollary (remark)

Note: it is not because $f(a)f(b) > 0$ that there is no root in the interval $[a, b]$. For example:



Brute-force method: introduction



Brute-force method: description

Requires:

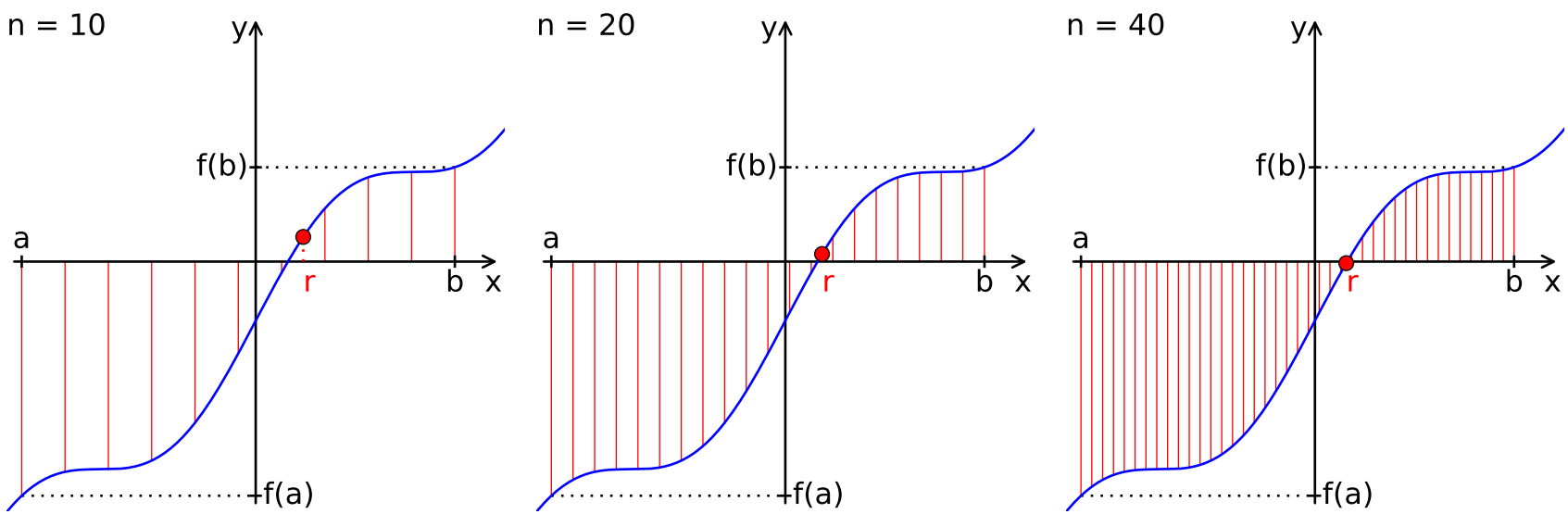
- f continuous over $[a, b]$ with $f(a)f(b) \leqslant 0$

Method:

- Divide $[a, b]$ in n intervals $[x_i, x_{i+1}]$ of equal width ($i = \{1, 2, \ldots, n\}$)
- The "solution" is the midpoint of the first of these intervals such that

$$f(x_i)f(x_{i+1}) \leqslant 0$$

- The larger n, the more accurate the method is



Brute-force method: basic implementation

```
function [r] = my_rootfind_bruteforce(f, a, b, n)

% Numerically "solve" f(r)=0 for r over the interval [a,b]
% by brute force, using a subdivision of the interval
% [a,b] into n intervals. This implementation assumes that f
% is continuous and that f(a)*f(b) <= 0.
% f is a function handle.

x = linspace(a, b, n+1);

for i = 1:n

    if f(x(i))*f(x(i+1)) <= 0
        r = (x(i)+x(i+1)) / 2;
        return
    end

end

end
```

Brute-force method: examples

```
>> % Calculate the square root of 2
>> f = @(x) x^2 - 2;
>> n = 50;
>> r = my_rootfind_bruteforce(f, 1, 2, n)
r =
    1.4100

>> % Let us increase the accuracy of the method
>> n = 500;
>> r = my_rootfind_bruteforce(f, 1, 2, n)
r =
    1.4150

>> % The root that is found depends on the interval specified
>> r = my_rootfind_bruteforce(f, -2, -1, n)
r =
   -1.4150
```

Note: even with $n = 500$, the "solution" is not very accurate! The brute-force method is **extremely inefficient!**

Brute-force method: advantages and downsides

Advantages:

- None (use the bisection method instead)

Downsides:

- Requires a and b such that $f(a)f(b) \leqslant 0$
- Requires f to be continuous over $[a, b]$
- This method is **extremely inefficient!**

Bisection method: description

Requires:

- f continuous over $[a, b]$ with $f(a)f(b) \leqslant 0$

Method:

1. Calculate $f(m)$ with $m = \text{midpoint} = (a + b)/2$

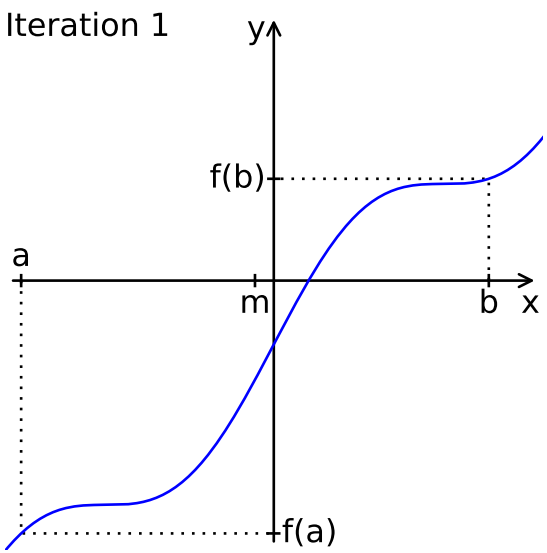
2. If $f(a)f(m) \leqslant 0$, then we know that there is a root in $[a, m]$
   - Set $b = m$

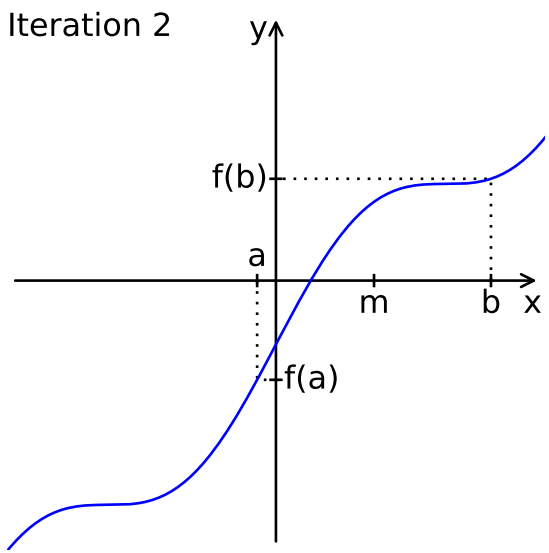
   Otherwise, we know that there is a root in $[m, b]$
   - Set $a = m$

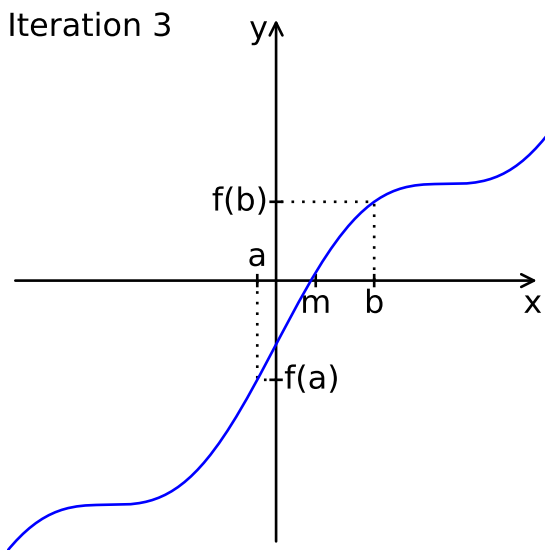
3. Repeat steps 1 and 2 until the "solution" (*i.e.* the value of m) is good enough *e.g.*,
   - $|f(m)| <$ specified tolerance; or
   - $|f(\text{current guess}) - f(\text{previous guess})| <$ specified tolerance

Bisection method: illustration

At every iteration, the size of the problem is divided by two!







Bisection method: basic implementation

```
function [r] = my_rootfind_bisection(f, a, b, tolerance)

% Numerically "solve" f(r)=0 for r over the interval [a,b]
% using the bisection method until abs(f(r)) <= tolerance.
% This implementation assumes that f is continuous and that
% f(a)*f(b) <= 0.
% f is a function handle.

r = (a+b) / 2;
while abs(f(r)) > tolerance
    if f(a)*f(r) <= 0
        b = r;
    else
        a = r;
    end
    r = (a+b) / 2;
end

end
```

See textbook (page 236) for a recursive implementation

Bisection method: examples

```
>> % Calculate the square root of 2
>> f = @(x) x^2 - 2;
>> tolerance = 1e-6;
>> r = my_rootfind_bisection(f, 1, 2, tolerance)
r =
    1.4142

>> % The root that is found depends on the interval specified
>> r = my_rootfind_bisection(f, -2, -1, tolerance)
r =
   -1.4142
```

Bisection: advantages and downsides

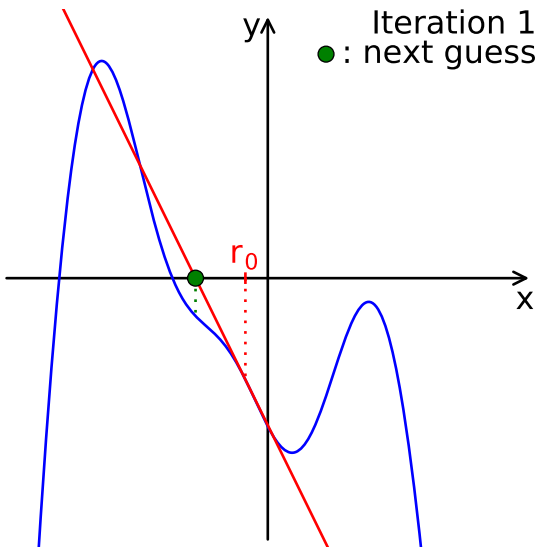
Advantages:

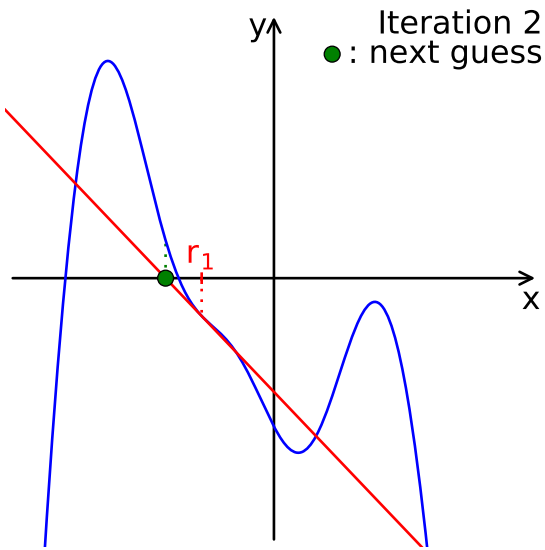
- This method always converges to a true solution if the conditions below are satisfied
- This method converges rather quickly

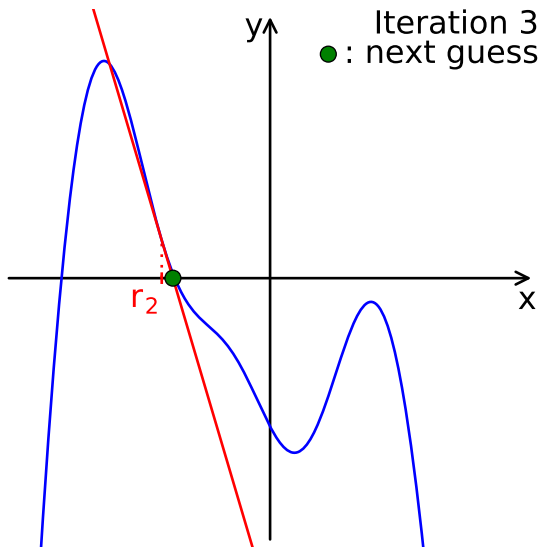
Downsides:

- Requires a and b such that $f(a)f(b) \leqslant 0$
- Requires f to be continuous over $[a, b]$

Newton-Raphson method: introduction







Newton-Raphson method: description

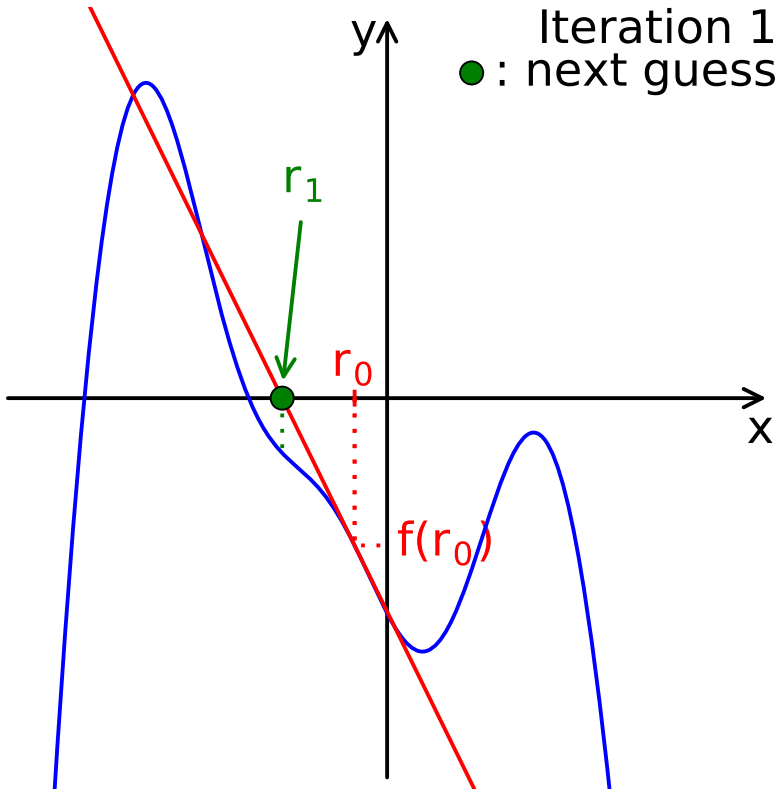
Requires:

- Function differentiable over $[a, b]$ (implies continuous over $[a, b]$)
- We know how to calculate the derivative of f

Method:

- Start with an initial guess r_0 for the root
  (hopefully close to the root we are looking for)
- Consider the tangent to the function's curve at that point
- Use the intersection of that tangent with the x-axis for the next guess r_1
- Repeat this process starting from the new guess until the "solution" (*i.e.* the last r_i calculated) is good enough *e.g.*,
  - $|f(r_i)| <$ specified tolerance; or
  - $|f(\text{current guess}) - f(\text{previous guess})| <$ specified tolerance

Newton-Raphson method: formula



$$\text{Slope of tangent} = f'(r_0) = \frac{\text{rise}}{\text{run}} = \frac{f(r_0)}{r_0 - r_1} \rightarrow r_1 = r_0 - \frac{f(r_0)}{f'(r_0)}$$

Generally:

$$r_{i+1} = r_i - \frac{f(r_i)}{f'(r_i)}$$

Newton-Raphson method: basic implementation

```
function [r] = my_rootfind_newtonraphson(f, df, r0, tolerance)

% Numerically "solve" f(r)=0 for r starting with the initial
% guess r0 using the Newton-Raphson method until
% abs(f(r)) <= tolerance. df is the derivative of f.
% f and df are function handles.

r = r0;
while abs(f(r)) > tolerance
    r = r - f(r)/df(r);
end

end
```

See textbook (page 239) for a recursive implementation

Newton-Raphson method: examples

```
>> % Calculate the square root of 2
>> f = @(x) x^2 - 2;
>> df = @(x) 2*x;
>> tolerance = 1e-6;
>> r = my_rootfind_newtonraphson(f, df, 2, tolerance)
r =
    1.4142

>> % The root that is found depends on the initial guess
>> r = my_rootfind_newtonraphson(f, df, -2, tolerance)
r =
   -1.4142
```

Newton-Raphson method: advantages and downsides

Advantages:

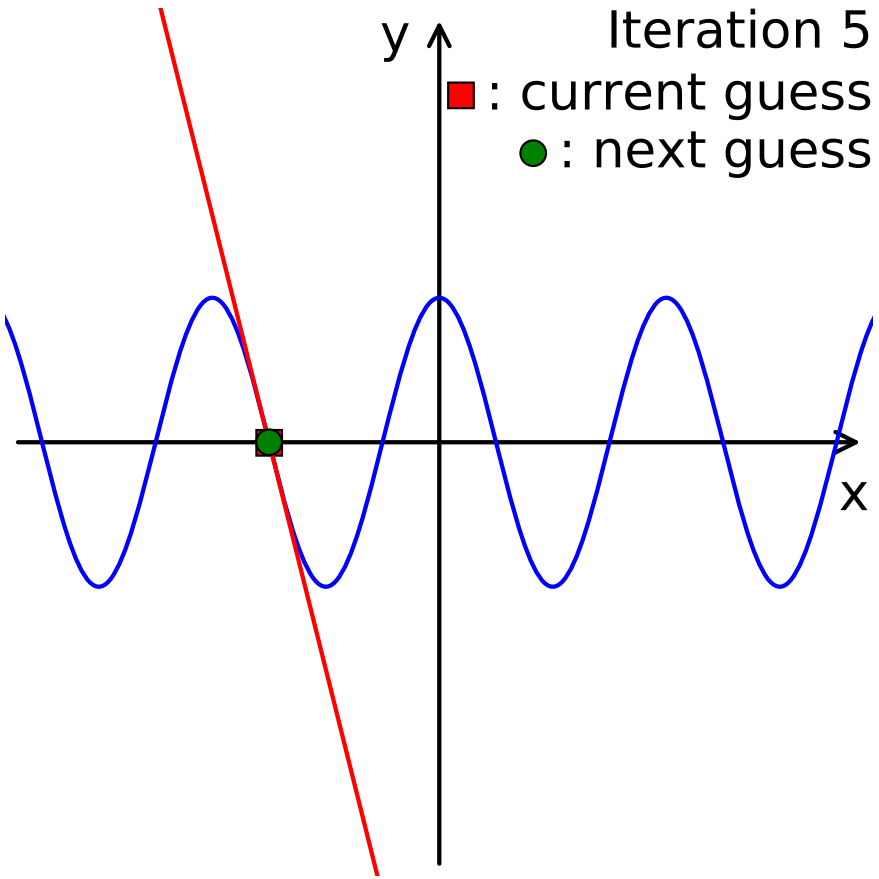
- **When it works well, it converges to a solution very fast!**
  i.e. we can find the root with high accuracy in few iterations
- Does not require to find an interval $[a, b]$ such that $f(a)f(b) \leqslant 0$

Downsides:

- Requires a good initial guess
- Requires to be able to calculate the derivative of f
- Sometimes the method finds a root that is not the one closest to the initial guess (this situation can be a problem in engineering applications where only one root makes physical sense)
- Sometimes it fails at finding any root

Newton-Raphson method: example of "partial failure"

In the example below, the method does yield an actual root of the function, but one that is very far from the initial guess. This situation often happens when $f'(r_i) \approx 0$. Note that the Newton-Raphson method completely fails when $f'(r_i) = 0$ (horizontal tangent $\rightarrow$ next guess is at infinity)



Function:

$f : x \mapsto 2\cos(2x)$

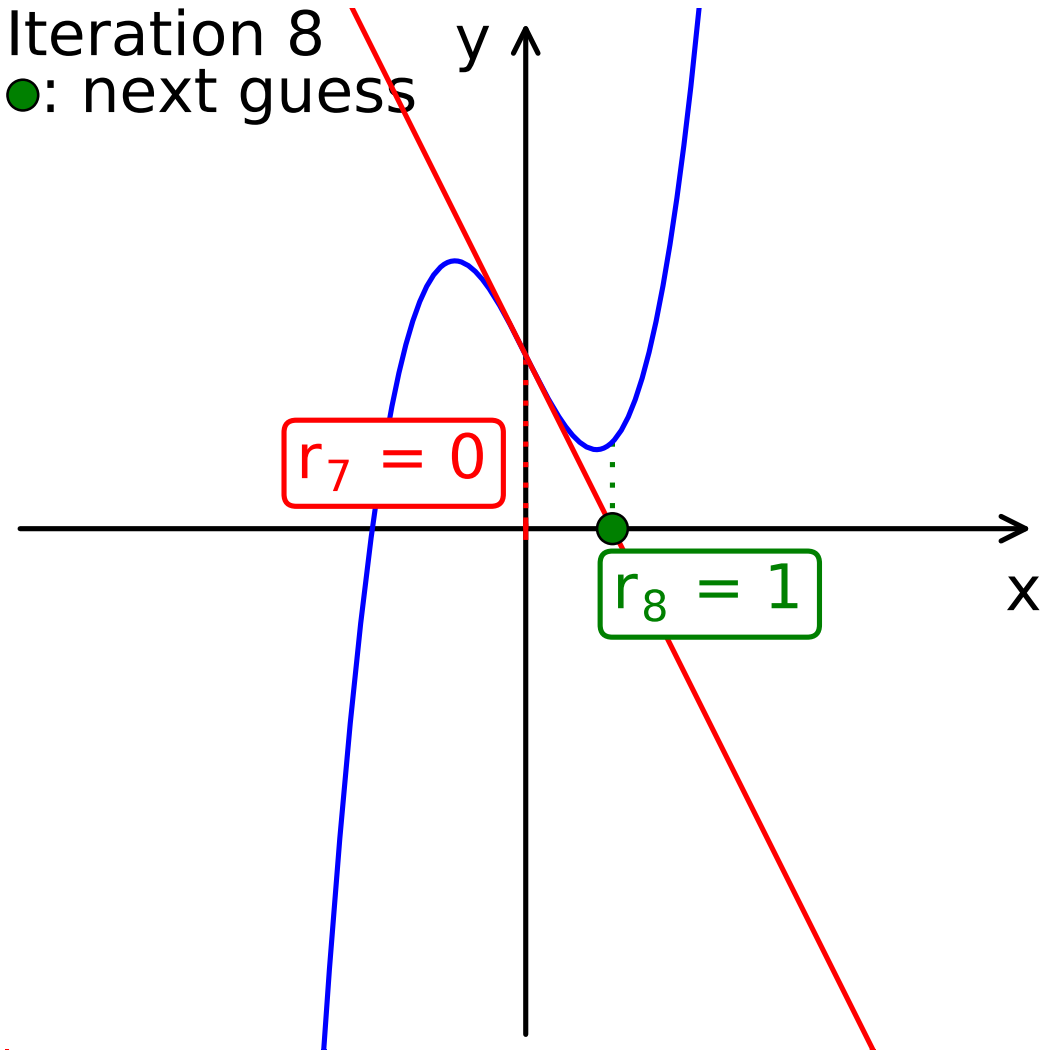
Initial guess:

$r_0 = 3.10$

i	r_i	r_{i+1}
0	3.10	-2.90
1	-2.90	-1.96
2	-1.96	-2.47
3	-2.47	-2.35
4	-2.35	-2.36
...	...	...

Newton-raphson method: example of complete failure

In the example below, the Newton-Raphson method never finds a root



Function:

$f : x \mapsto x^3 - 2x + 2$

Initial guess:

$r_0 = 1$

i	r_i	r_{i+1}
0	1	0
1	0	1
2	1	0
3	0	1
4	1	0
5	0	1
6	1	0
7	0	1
...	...	...

Matlab's built-in function fsolve

```
>> % r = fsolve(f, r0): "solves" f(r) = 0 for r,
>> % starting  with the initial guess r0. f should be
>> % a function handle that takes a numerical scalar
>> % as input, and outputs a numerical scalar

>> % Example, calculate the square root of 2
>> f = @(x) x^2 - 2;
>> r = fzero(f, 2)
r =
    1.4142

>> % The value returned by fzero depends on the initial guess
>> r = fzero(f, -2)
r =
   -1.4142
```

Important: sometimes, `fzero` returns a number that is not a root of the function (`help fzero` for more details)

Conclusions: Bisection versus Newton-Raphson

Both the bisection and the Newton-Raphson methods require the function to be continuous

The Newton-Raphson method generally converges faster than the bisection method, but:

- It requires making a good first guess
- It sometimes fails at finding a "solution" (or the expected "solution")
- It requires being able to calculate the derivative of the function

The bisection method requires finding an interval $[a, b]$ such that $f(a)f(b) \leqslant 0$; the Newton-Raphson method does not have this requirement