MA5233 Computational Mathematics

Lecture 6: Nonlinear Equations

Simon Etter

Semester I, AY 2020/2021

Nonlinear Equations

Problem statement

Given a continuous function $f : \mathbb{R}^n \to \mathbb{R}^n$,
find $x \in \mathbb{R}^n$ such that $f(x) = 0$.

Examples

- $$ax^2 + bx + c = 0 \quad \Longleftrightarrow \quad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
- $$\begin{Bmatrix} x^2 - y^2 = 0 \\ 1 + xy = 0 \end{Bmatrix} \quad \Longleftrightarrow \quad \begin{Bmatrix} x = \pm 1, \\ y = \mp 1. \end{Bmatrix}$$

Terminology

- A point x such that $f(x) = 0$ is called a *zero* or *root* of f.
- Solving $f(x) = 0$ is also called *root-finding*.

Nonlinear Equations

Applications

- Function inversion: $x = f^{-1}(y) \iff f(x) = y.$

  In particular, methods for solving $f(x) = 0$ allow us to implement $1/x$ and $\sqrt{x}$ using only addition and multiplication.
- Optimisation: $x = \arg\min f(x) \iff \nabla f(x) = 0.$
- Actually solve nonlinear equations, e.g. to determine launch parameters so your rocket reaches the moon.

One equation vs. many equations

The mathematical properties and the algorithms for solving $f(x) = 0$ are quite different depending on whether $f : \mathbb{R}^n \to \mathbb{R}^n$ is a scalar function ($n = 1$) or a multi-dimensional function ($n > 1$).

Correspondingly, we will discuss these two cases separately, starting with the scalar case $f : \mathbb{R} \to \mathbb{R}$.

Nonlinear Equations

Discussion: Existence of solutions

The "computers as function evaluators" concept from Lecture 1 requires that every input specifies exactly one output. This requirement is not necessarily satisfied in the case of nonlinear equations since an arbitrary function $f(x)$ may have zero, one or many roots.

Before discussing how to implement $f \mapsto \big(x : f(x) = 0\big)$, we should hence first clarify under what conditions on f this mapping is well defined.

The result on the next slide provides one such condition.

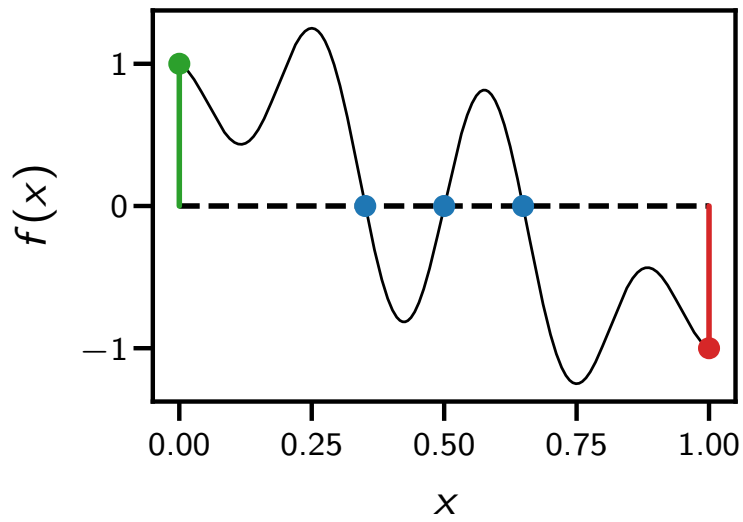
Nonlinear Equations

Bracketing theorem

If $f : [a, b] \to \mathbb{R}$ is continuous and $\text{sign}(f(a)) \neq \text{sign}(f(b))$, then $f(x)$ has at least one root in $[a, b]$.

This result is sometimes called Bolzano's theorem in the literature.

Proof. Straightforward application of the intermediate value theorem.



Terminology

An interval $[a, b]$ such that $\text{sign}(f(a)) \neq \text{sign}(f(b))$ is called a *bracketing interval*.

Nonlinear Equations

Discussion

The bracketing theorem provides a simple algorithm for iteratively narrowing the bracketing interval. This algorithm is known as the *bisection method*.

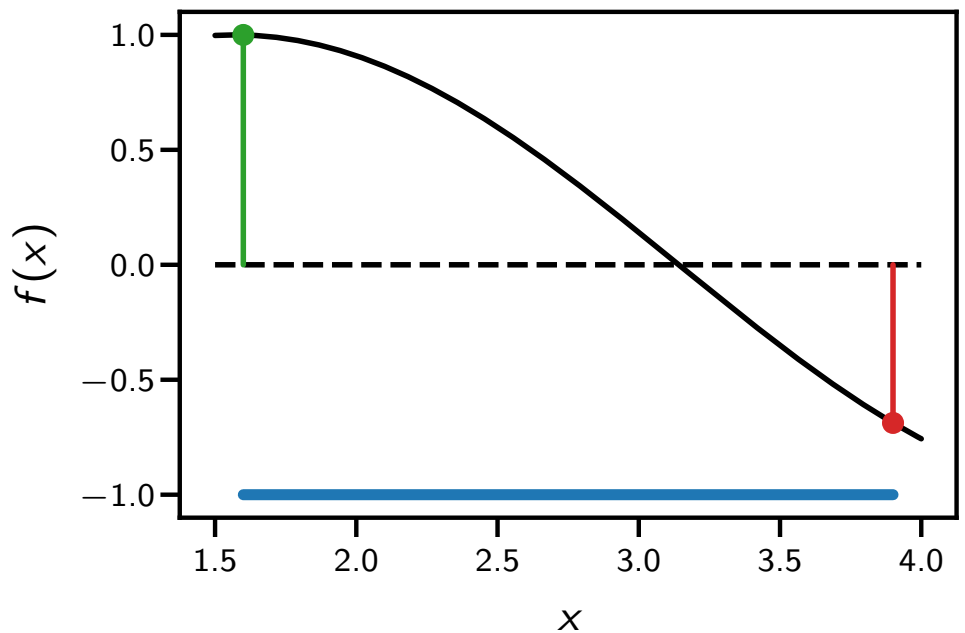
Algorithm Bisection method

1: Start with any bracketing interval $[a_0, b_0]$.
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Compute $m_k = \frac{a_k + b_k}{2}$.
4: Update $[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, m_k] & \text{if } \operatorname{sign}\big(f(a_k)\big) \neq \operatorname{sign}\big(f(m_k)\big), \\ [m_k, b_k] & \text{otherwise.} \end{cases}$
5: **end for**

Nonlinear Equations

Algorithm Bisection method

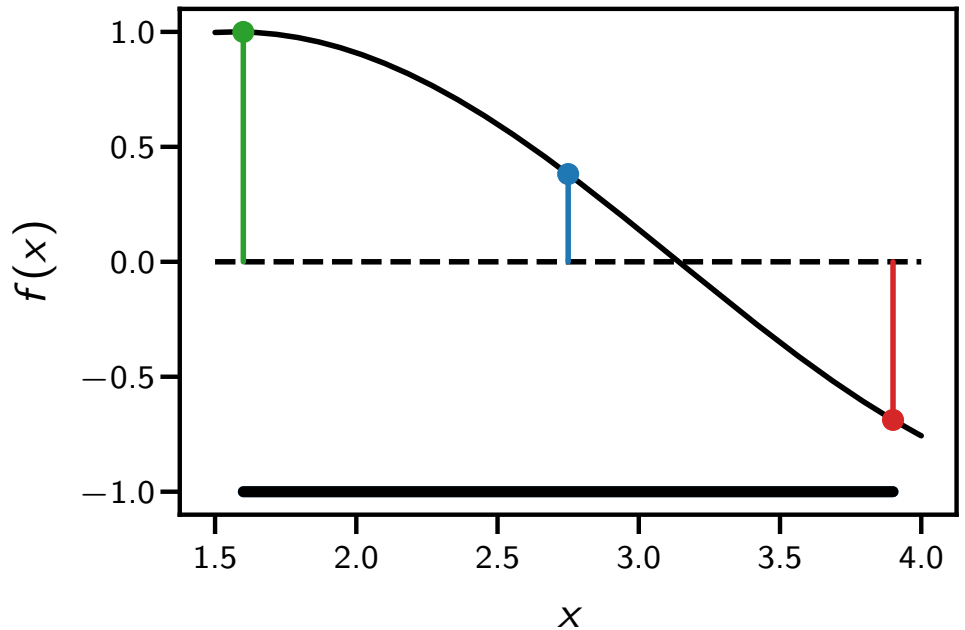
1: Start with any bracketing interval $[a_0, b_0]$.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute $m_k = \frac{a_k + b_k}{2}$.
4: Update $[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, m_k] & \text{if } \operatorname{sign}\big(f(a_k)\big) \neq \operatorname{sign}\big(f(m_k)\big), \\ [m_k, b_k] & \text{otherwise.} \end{cases}$
5: **end for**



Nonlinear Equations

Algorithm Bisection method

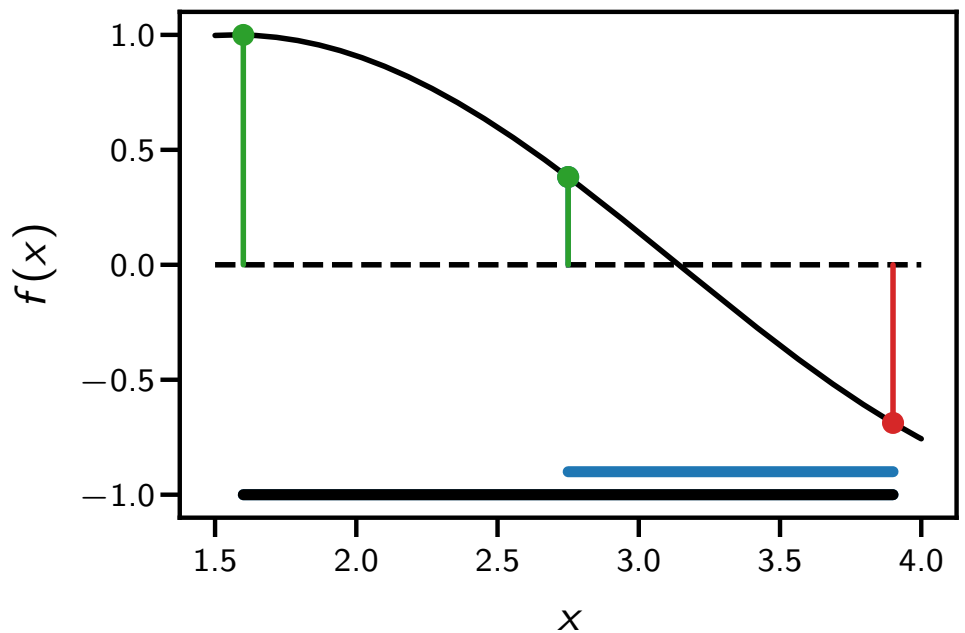
1: Start with any bracketing interval $[a_0, b_0]$.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute $m_k = \frac{a_k + b_k}{2}$.
4: Update $[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, m_k] & \text{if } \operatorname{sign}\big(f(a_k)\big) \neq \operatorname{sign}\big(f(m_k)\big), \\ [m_k, b_k] & \text{otherwise.} \end{cases}$
5: **end for**



Nonlinear Equations

Algorithm Bisection method

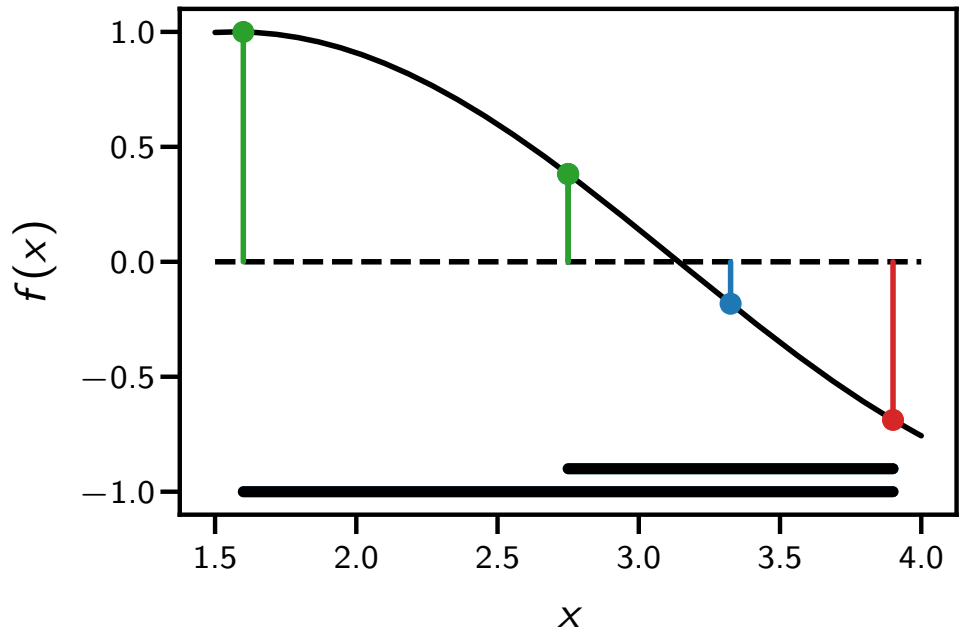
1: Start with any bracketing interval $[a_0, b_0]$.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute $m_k = \frac{a_k + b_k}{2}$.
4: Update $[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, m_k] & \text{if } \operatorname{sign}\big(f(a_k)\big) \neq \operatorname{sign}\big(f(m_k)\big), \\ [m_k, b_k] & \text{otherwise.} \end{cases}$
5: **end for**



Nonlinear Equations

Algorithm Bisection method

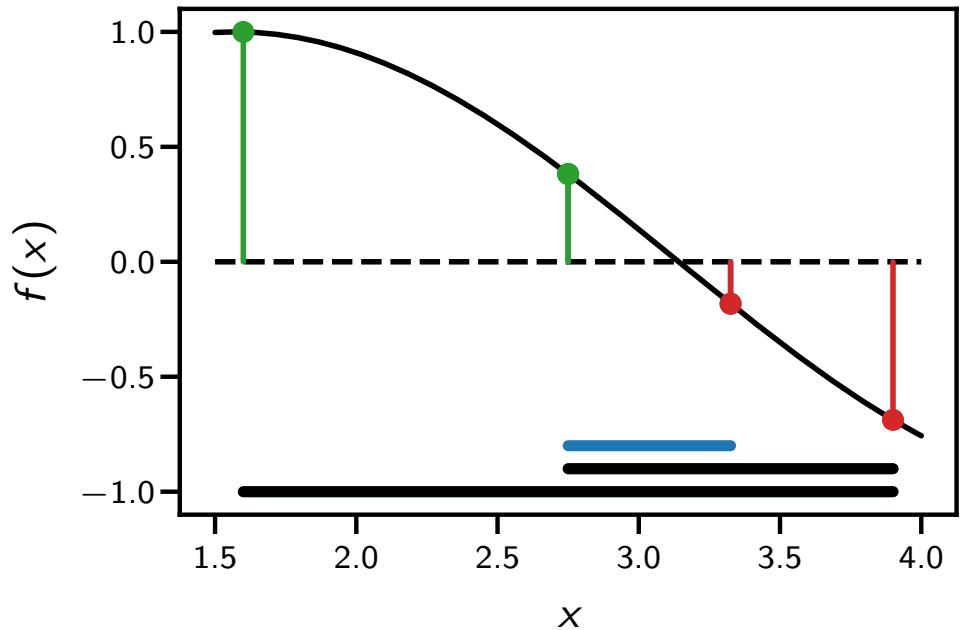
1: Start with any bracketing interval $[a_0, b_0]$.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute $m_k = \frac{a_k + b_k}{2}$.
4: Update $[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, m_k] & \text{if } \operatorname{sign}\big(f(a_k)\big) \neq \operatorname{sign}\big(f(m_k)\big), \\ [m_k, b_k] & \text{otherwise.} \end{cases}$
5: **end for**



Nonlinear Equations

Algorithm Bisection method

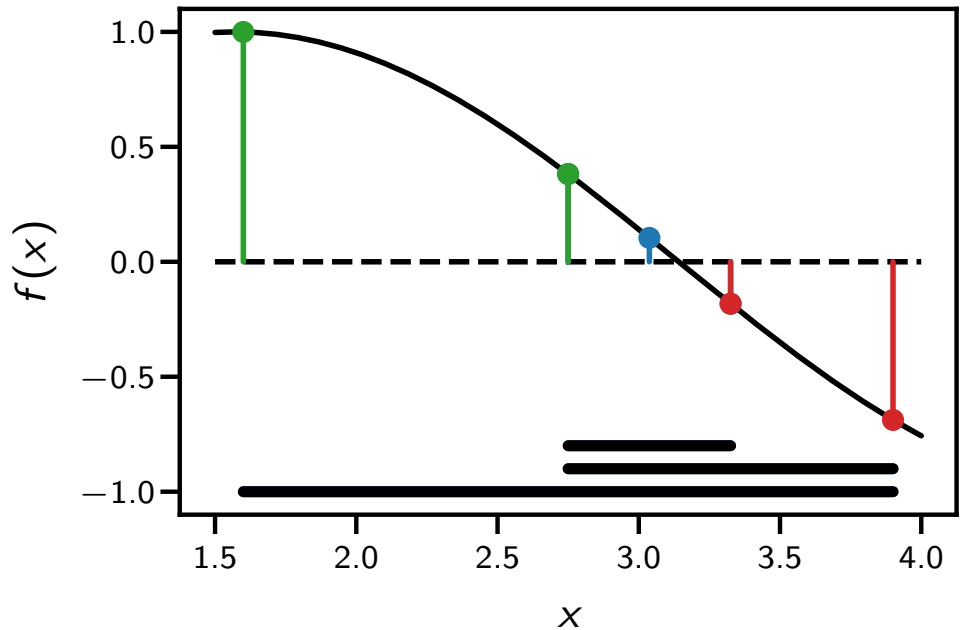
1: Start with any bracketing interval $[a_0, b_0]$.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute $m_k = \frac{a_k + b_k}{2}$.
4: Update $[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, m_k] & \text{if } \operatorname{sign}\big(f(a_k)\big) \neq \operatorname{sign}\big(f(m_k)\big), \\ [m_k, b_k] & \text{otherwise.} \end{cases}$
5: **end for**



Nonlinear Equations

Algorithm Bisection method

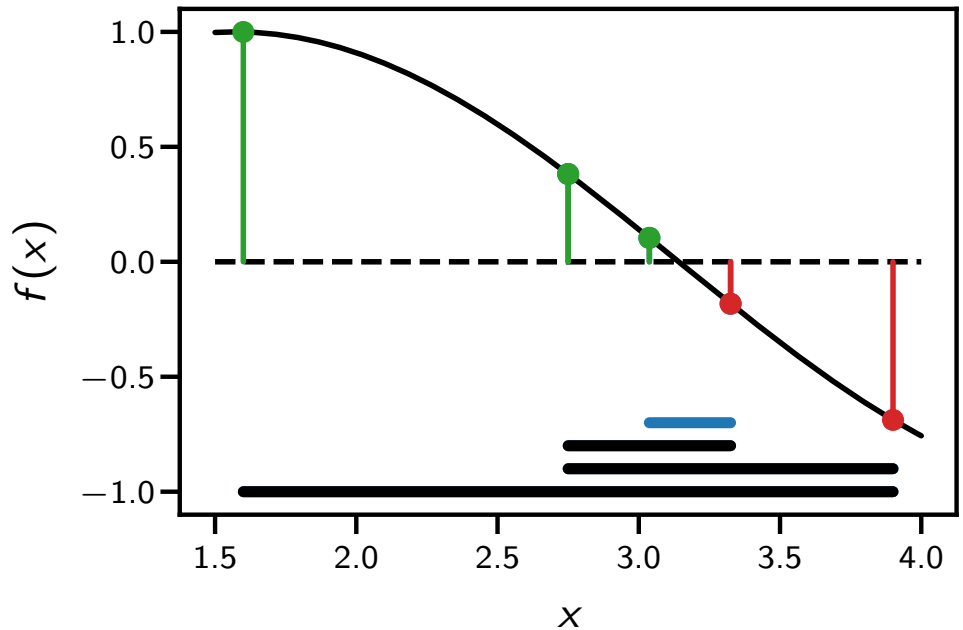
1: Start with any bracketing interval $[a_0, b_0]$.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute $m_k = \frac{a_k + b_k}{2}$.
4: Update $[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, m_k] & \text{if } \operatorname{sign}\big(f(a_k)\big) \neq \operatorname{sign}\big(f(m_k)\big), \\ [m_k, b_k] & \text{otherwise.} \end{cases}$
5: **end for**



Nonlinear Equations

Algorithm Bisection method

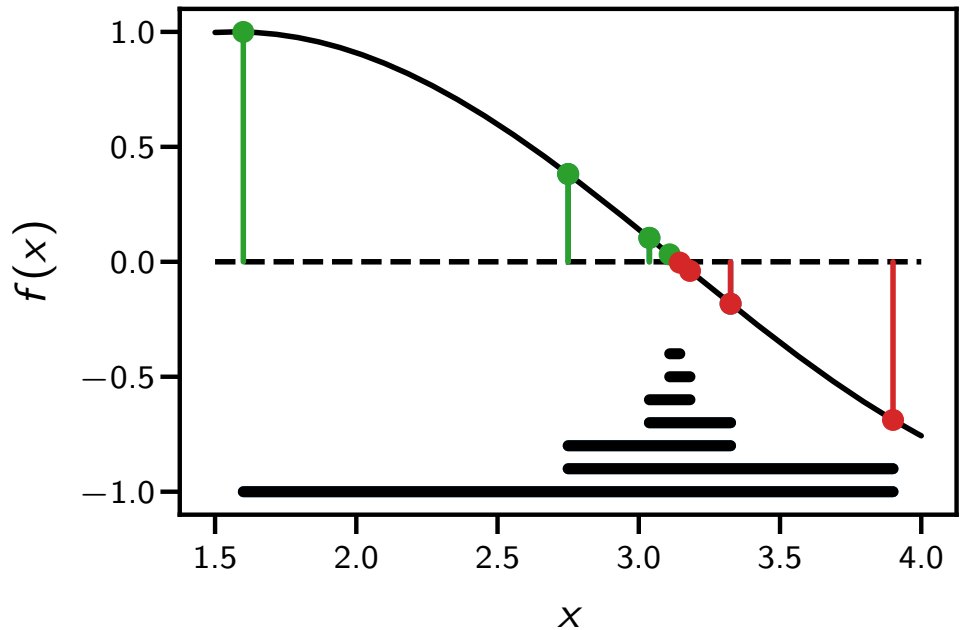
1: Start with any bracketing interval $[a_0, b_0]$.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute $m_k = \frac{a_k + b_k}{2}$.
4: Update $[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, m_k] & \text{if } \operatorname{sign}\big(f(a_k)\big) \neq \operatorname{sign}\big(f(m_k)\big), \\ [m_k, b_k] & \text{otherwise.} \end{cases}$
5: **end for**



Nonlinear Equations

Algorithm Bisection method

1: Start with any bracketing interval $[a_0, b_0]$.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute $m_k = \frac{a_k + b_k}{2}$.
4: Update $[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, m_k] & \text{if } \operatorname{sign}\big(f(a_k)\big) \neq \operatorname{sign}\big(f(m_k)\big), \\ [m_k, b_k] & \text{otherwise.} \end{cases}$
5: **end for**



Nonlinear Equations

Thm: Convergence of bisection method

Denote by $[a_k, b_k]$ the search interval after k bisection steps. We have

$$|b_k - a_k| = 2^{-k} |b_0 - a_0|,$$

i.e. the bisection method is exponentially convergent with rate 2.

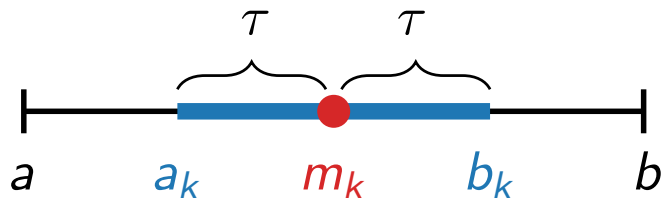
Proof. Obvious.

Termination criterion for the bisection method

The bisection method is our first example of an iterative algorithm, i.e. an algorithm which starts from an initial guess and iteratively refines this guess until it is "good enough".

Providing a precise criterion for when a guess is "good enough" is difficult for many iterative algorithms, but for the bisection method it is trivial:

If we want to know the root x up to an error of at most τ, then we can stop bisecting once $|b_k - a_k| \leq 2\tau$ and return $x \approx m_k = \frac{a_k + b_k}{2}$.



Nonlinear Equations

Termination criterion for the bisection method (continued)

Since $|b_k - a_k| = 2^{-k}\,|b_0 - a_0|$, this condition is satisfied after a number of steps k given by

$$|b_k - a_k| = 2^{-k}\,|b_0 - a_0| \leq 2\tau \qquad \Longleftrightarrow \qquad k \geq \log_2\left(\frac{|b_0 - a_0|}{\tau}\right) - 1.$$

Bisection thus allows us to determine roots up to an arbitrary accuracy in a finite and a priori computable number of steps.

"A priori" means "before running the algorithm". Its opposite is "a posteriori".

In particular, bisection allows us to determine roots up to machine precision, see `bisection()` and `test_bisection()`.

Nonlinear Equations

Implementation of bisection method

See `bisection()` and `test_bisection()`.

Discussion

We have just seen that the bisection method computes a result with hard error bounds in a finite and predictable number of steps.

These are already very powerful properties, and it gets even better.

Thm: Optimality of the bisection method

No algorithm can reduce a bracketing interval $[a_0, b_0]$ to another bracketing interval $[a_k, b_k]$ with $|b_k - a_k| \leq 2^{-k} |b_0 - a_0|$ using less than k function evaluations for every function $f : \mathbb{R} \to \mathbb{R}$.

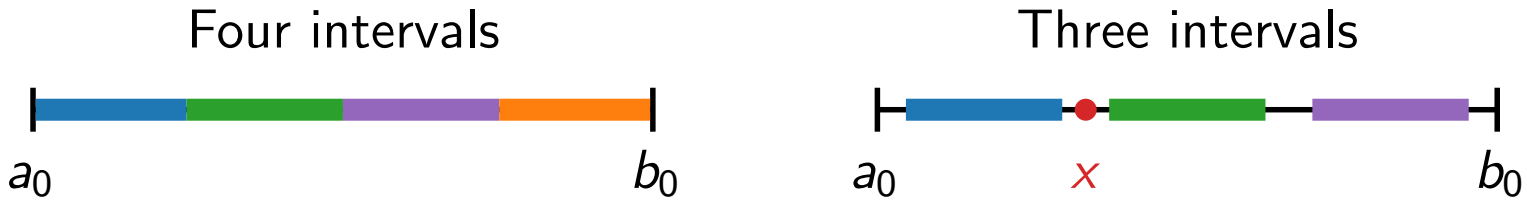
Proof. The bisection method is essentially binary search over the real line; hence the proof of the above statement is essentially the same as the proof regarding the optimality of binary search. The next slide provides the details.

Nonlinear Equations

Proof of optimality of bisection (not examinable).

Let A be an algorithm achieving the desired reduction in $[a_k, b_k]$.
We observe:

1. A must be able to return at least 2^k different intervals $[a_k, b_k]$: if not, there are points $x \in [a_0, b_0]$ which are not contained in any of the output intervals and hence the algorithm must be wrong for functions with roots at these points.



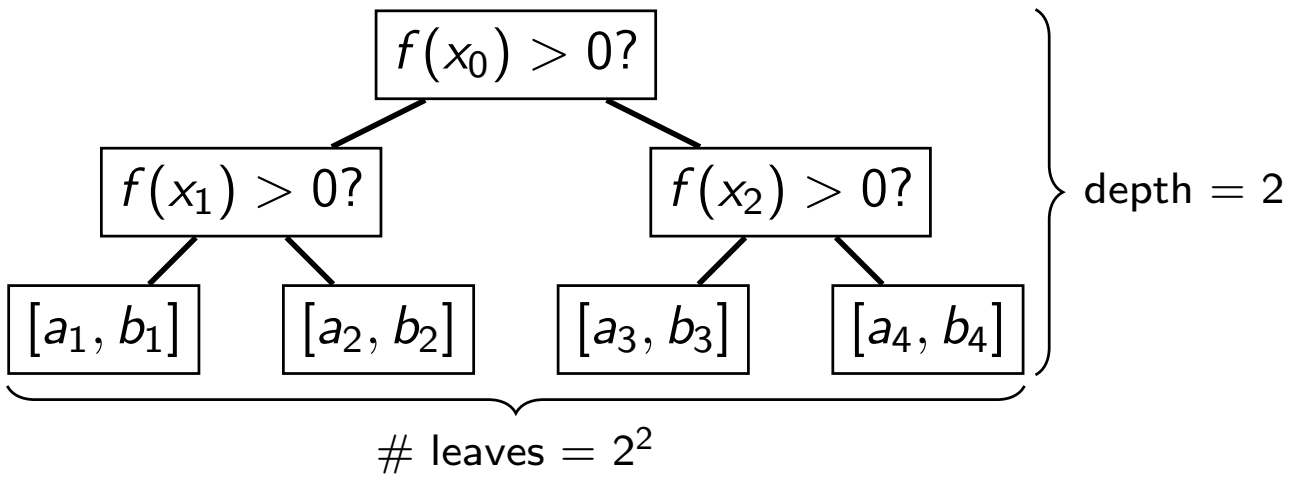
2. Unless we have some regularity assumptions on $f(x)$ (e.g. Lipschitz continuity), every evaluation of $f(x)$ can tell us only whether the root is to the left or right of the evaluation point and nothing more.

Nonlinear Equations

Proof of optimality of bisection (not examinable, continued).

Item 2 implies that we can visualise A as a binary tree where each node represents one evaluation of $f(x)$, and Item 1 means that this tree must have at least 2^k leaves.

The depth of such a tree must be at least k, i.e. there must be at least one leaf which requires k function evaluation to be reached.



Nonlinear Equations

Discussion: Root-finding algorithms

Put differently, the bisection optimality theorem says that for any root-finding algorithm A, we can find some function $f(x)$ such that A takes at least as long as the bisection method when applied to $f(x)$.

Moreover, the proof indicates that if a root-finding algorithm A is faster than bisection for some functions $f(x)$, then at least one of the following conditions must be true.

- A is slower or wrong for other functions $f(x)$ (i.e. some leaves are further from the root, or some leaves are missing).
- A assumes that more information than just point values of $f(x)$ is available (i.e. we can ask questions other than $f(x_k) > 0$ to decide which branches to pursue).

Many root-finding algorithms have been discussed in the literature, and each one of them represents a particular compromise between best- and worst-case performance on the one hand, and assumptions regarding additional information about $f(x)$ on the other hand.

I will discuss only one further root-finding algorithm, namely Newton's method.

Nonlinear Equations

Newton's method

Let us assume that we have some initial guess $x_0 \in \mathbb{R}$ for the root of a differentiable function $f(x)$.

According to Taylor's theorem, we then have

$$f(x) \approx f(x_0) + f'(x_0)\,(x - x_0);$$

hence it makes sense to hope that $x_1 \in \mathbb{R}$ determined such that

$$f(x_0) + f'(x_0)\,(x_1 - x_0) = 0 \qquad \Longleftrightarrow \qquad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

will be a better estimate for the root of $f(x)$.

Newton's method consists in repeating this process indefinitely, i.e. Newton's method constructs a sequence $(x_k)_{k=0}^{\infty}$ defined recursively by

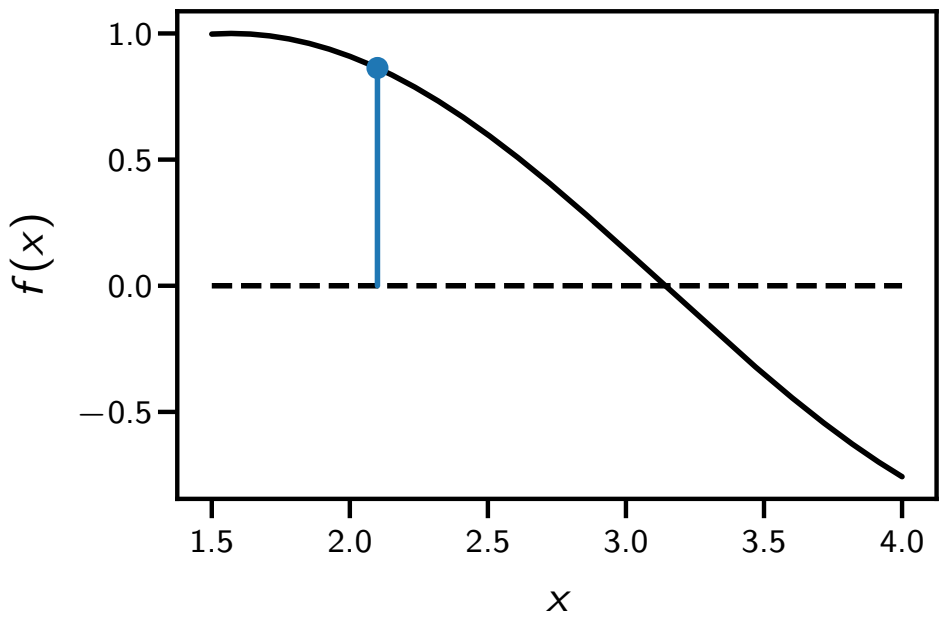
$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}.$$

Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \sin(x), \quad x_0 = 2.1$$

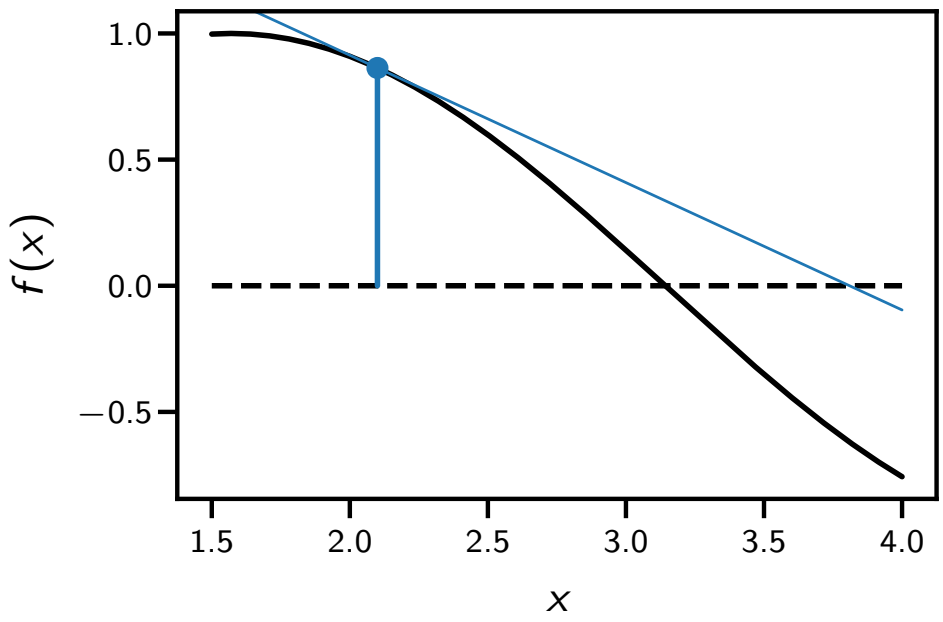


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \sin(x), \quad x_0 = 2.1$$

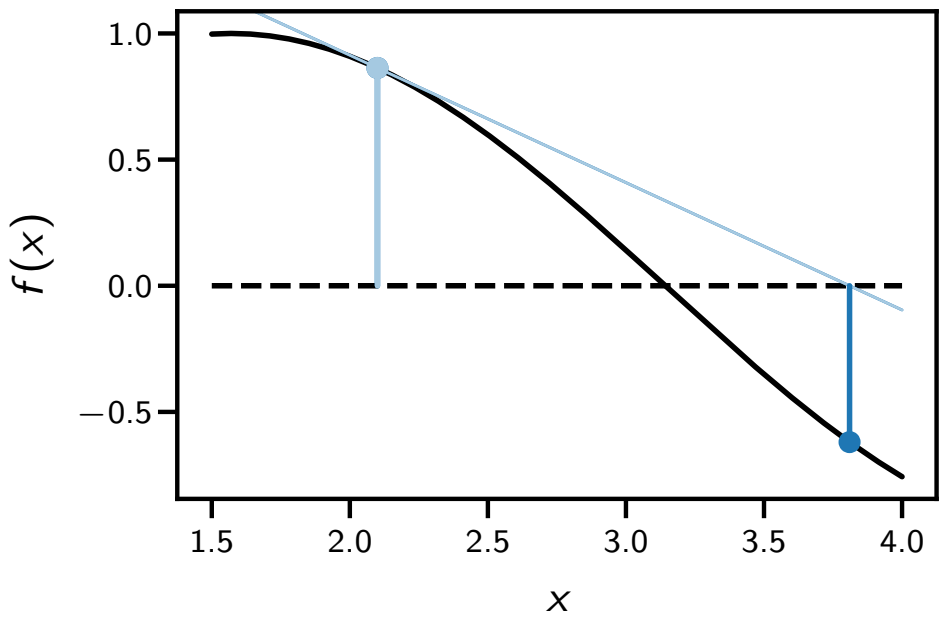


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \sin(x), \quad x_0 = 2.1$$

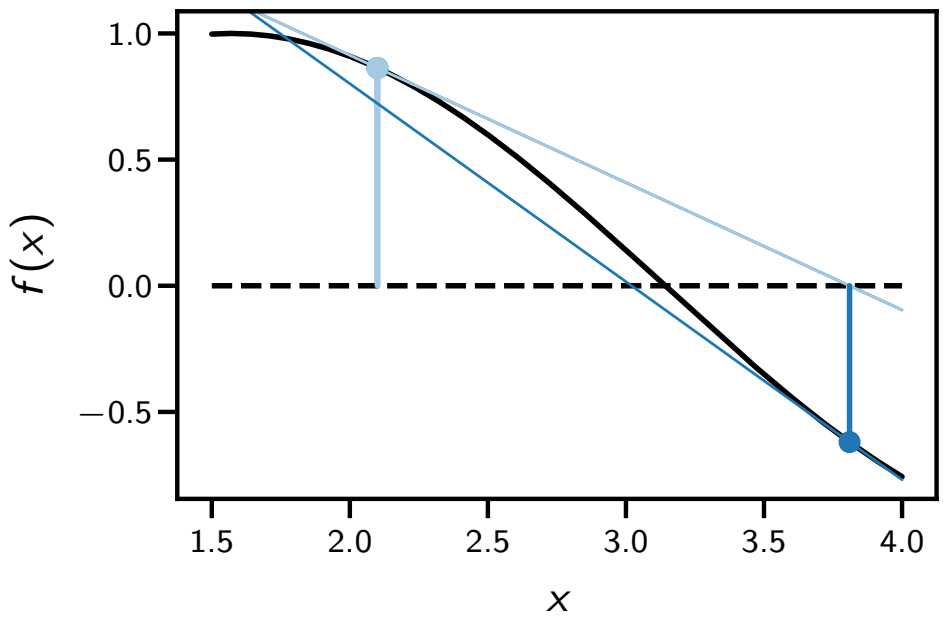


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \sin(x), \quad x_0 = 2.1$$

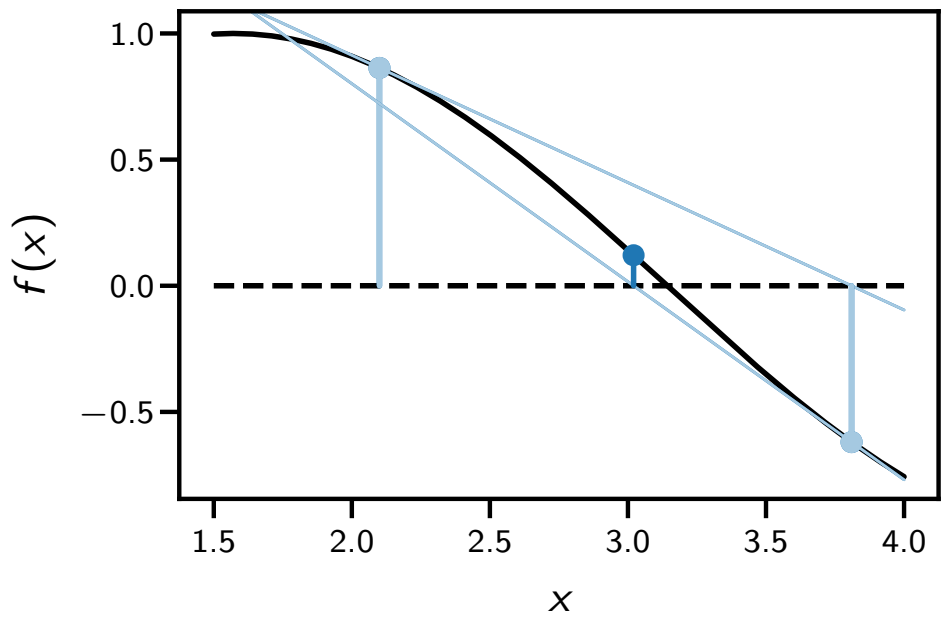


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \sin(x), \quad x_0 = 2.1$$

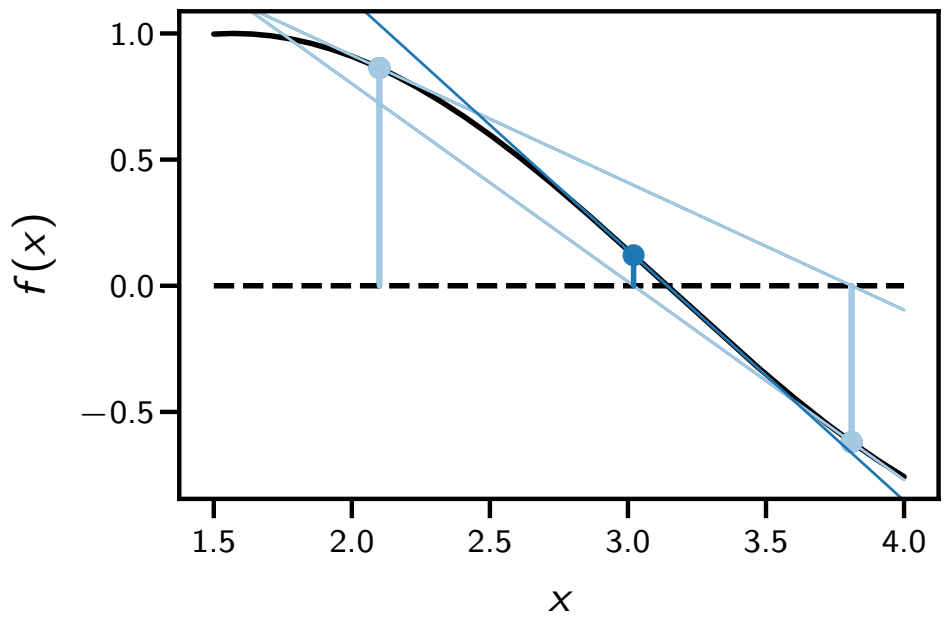


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \sin(x), \quad x_0 = 2.1$$

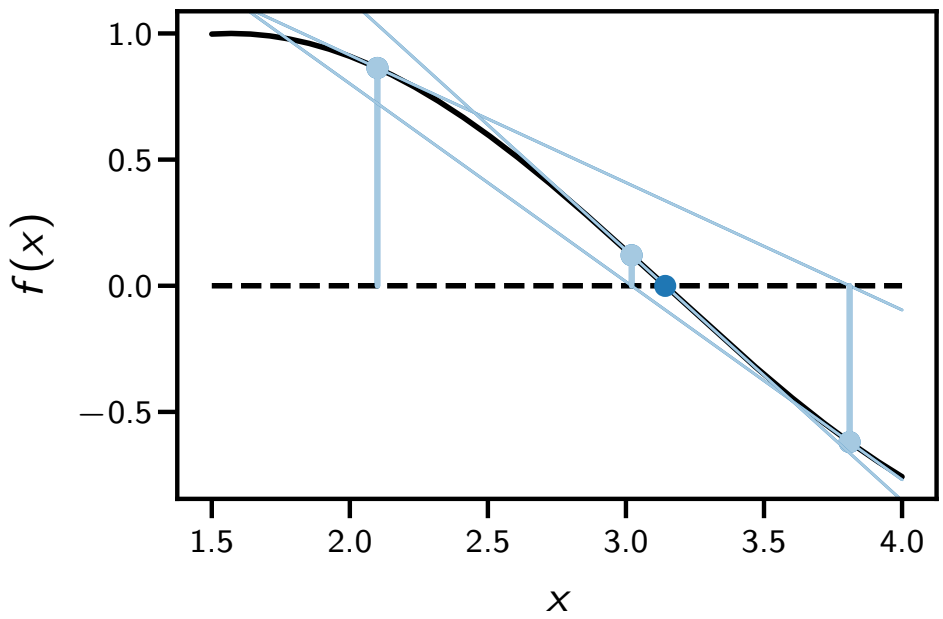


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \sin(x), \quad x_0 = 2.1$$

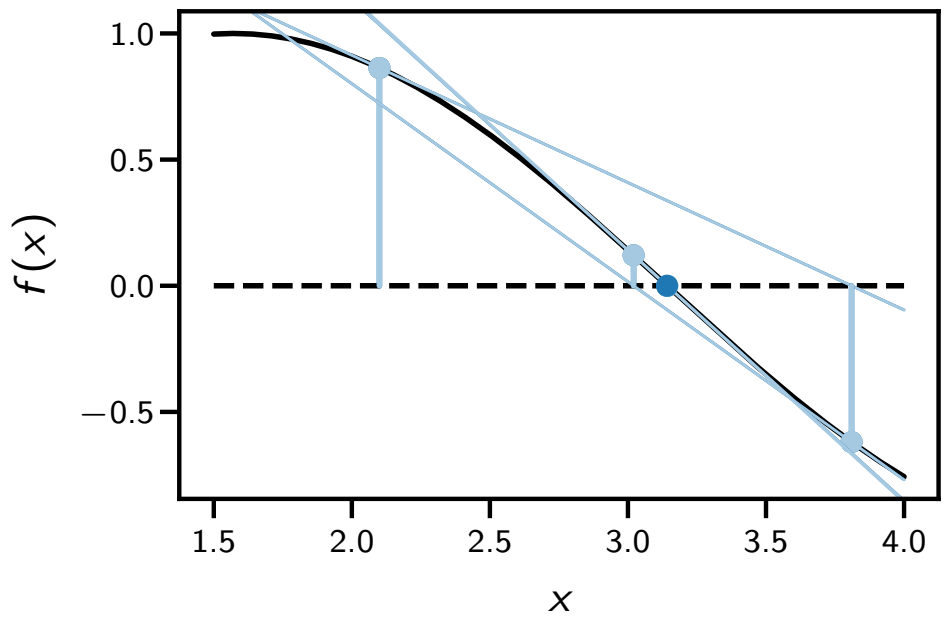


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \sin(x), \quad x_0 = 2.1$$



Nonlinear Equations

Thm: Error recursion for Newton's method

Assume $f : \mathbb{R} \to \mathbb{R}$ is twice differentiable and has a root $x^\star \in \mathbb{R}$ such that $f'(x^\star) \neq 0$, and denote by

$$x_k \in \mathbb{R}, \qquad x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

two consecutive Newton iterates. Then,

$$x_{k+1} - x^\star = O\Big(|x_k - x^\star|^2\Big) \qquad \text{for } x_k \to x^\star.$$

Proof. Subtracting the root $x^\star$ on both sides of the Newton iteration formula and Taylor-expanding $f(x)$ around $x^\star$, we obtain

$$\begin{aligned} x_{k+1} - x^\star &= x_k - x^\star - \tfrac{f(x_k)}{f'(x_k)} \\ &= x_k - x^\star - \frac{f'(x^\star)\,(x_k - x^\star) + \frac{1}{2} f''(x^\star)\,(x_k - x^\star)^2 + O\big(|\cdot|^3\big)}{f'(x^\star) + f''(x^\star)\,(x_k - x^\star) + O\big(|\cdot|^2\big)} \\ &= \frac{1}{2}\frac{f''(x^\star)}{f'(x^\star)}\,(x_k - x^\star)^2 + O\big(|\cdot|^3\big). \end{aligned}$$

A rigorous argument for why we can drop the $O(|\cdot|)$ term in the denominator requires a few lines of calculations. I omit the details here since you can easily work them out yourself based on your findings in Assignment 1, Task 2.

Nonlinear Equations

Discussion: Convergence of Newton's method

The $x_{k+1} - x^\star = O\big((x_k - x^\star)^2\big)$ error recursion for Newton's method indicates extremely rapid convergence: we have

$$|x_1 - x^\star| = O\big(|x_0 - x^\star|^2\big), \qquad |x_2 - x^\star| = O\big(|x_0 - x^\star|^4\big),$$
$$|x_3 - x^\star| = O\big(|x_0 - x^\star|^8\big), \qquad |x_4 - x^\star| = O\big(|x_0 - x^\star|^{16}\big);$$

thus if the initial error satisfies $|x_0 - x^\star| \approx 10^{-1}$, then after just four iterations we obtain

$$|x_4 - x^\star| = O\big(10^{-16}\big) = O\big(\texttt{eps(Float64)}\big).$$

In comparison, the number of steps k required by the bisection method to achieve the same error reduction is given by

$$10^{-16} \le 2^{-k}\, 10^{-1} \qquad \Longleftrightarrow \qquad k = \log_2(10^{15}) \approx 50,$$

i.e. Newton converges in about 12x fewer iterations than bisection.

It is important to distinguish number of iterations from runtime here: Newton evaluates both $f(x_k)$ and $f'(x_k)$ in each iteration while bisection evaluates only $f(m_k)$; hence if $f'(x_k)$ is as expensive to compute as $f(x_k)$, then Newton's method is only 6x faster than bisection under the given assumptions.

Nonlinear Equations

Discussion: Convergence of Newton's method (continued)

Another way to get a feeling for the speed of convergence of Newton's method is to observe that $x_{k+1} - x^\star = O\big((x_k - x^\star)^2\big)$ roughly means that

$$x_k \text{ has } n \text{ correct digits} \quad \Longrightarrow \quad x_{k+1} \text{ has } 2n \text{ correct digits}$$

Terminology: x_k has n correct digits $\iff$ $|x_k - x^\star| \approx 10^{-n}$.

In comparison, the analogous statement for the bisection method is

$$m_k \text{ has } n \text{ correct digits} \quad \Longrightarrow \quad m_{k+3} \text{ has } n+1 \text{ correct digits.}$$

These observations are illustrated in `bisection_convergence()` and `newton_convergence()`.

Terminology

- $x_{k+1} - x^\star = O\big((x_k - x^\star)\big)$ is called *quadratic convergence*.
- $|x_{k+1} - x^\star| \leq r\,|x_k - x^\star|$ for some $r < 1$ is called *linear convergence*.

Linear convergence is the same as exponential convergence. I will mostly say "exponential convergence" since linear convergence could be confused with $O(k^{-1})$.

Nonlinear Equations

Newton's method and roots of multiplicity > 1

Recall from slide 16 that the lowest-order term in the Newton error recursion formula $x_{k+1} - x^\star = O\big(|x_k - x^\star|^2\big)$ is

$$x_{k+1} - x_\star = \frac{1}{2}\,\frac{f''(x^\star)}{f'(x^\star)}\,(x_k - x^\star)^2 + O\big(|\cdot|^3\big).$$

This explains why we had to assume $f'(x^\star) \neq 0$ in the theorem on slide 16: if $f'(x^\star) = 0$, then the above formula would not make sense.

I will next discuss what happens if $f'(x^\star) = 0$. In order to do so, it is useful to introduce the following terminology.

Def: Multiplicity of roots

Let $x^\star$ be a root of an infinitely differentiable function $f(x)$. The smallest integer $m \geq 1$ such that $f^{(m)}(x^\star) \neq 0$ is called the *multiplicity* of $x^\star$.

Example: $x^\star = 0$ is a root of multiplicity 2 of $f(x) = x^2$ since

$$f(x^\star) = (x^\star)^2 = 0, \qquad f'(x^\star) = 2x^\star = 0, \qquad f''(x^\star) = 2.$$

Nonlinear Equations

Newton's method and roots of multiplicity > 1 (continued)

If $x^\star$ is a root of multiplicity m, then the lowest-order term in the Newton error recursion formula becomes

$$\begin{aligned} x_{k+1} - x^\star &= x_k - x^\star - \frac{0 + \frac{1}{m!} f^{(m)}(x^\star)\,(x_k - x^\star)^m + O\big(|\cdot|^{m+1}\big)}{0 + \frac{1}{(m-1)!} f^{(m)}(x^\star)\,(x_k - x^\star)^{m-1} + O\big(|\cdot|^{m}\big)} \\ &= \Big(1 - \frac{1}{m}\Big)\,(x_k - x^\star) + O\big(|\cdot|^2\big). \end{aligned}$$

This shows that Newton's method converges only linearly when applied to roots of multiplicities > 1, and the rate of convergence $\big(1 - \frac{1}{m}\big)$ is as good as the bisection method for $m = 2$ and worse for $m > 2$.

This effect is illustrated in `newton_convergence_slow()`.

Remark

Roots of multiplicities $m > 1$ do not occur very frequently in applications. In the following, I will therefore focus on the case $f'(x^\star) \neq 0$, and I may occasionally not point out this assumption very clearly to avoid unnecessary distractions.

Nonlinear Equations

Guaranteed convergence for good enough initial guesses

Note that both cases

$$x_{k+1} - x^\star = \begin{cases} O\big(|x_k - x^\star|^2\big) & (\text{multiplicity } m = 1) \\ \big(1 - \frac{1}{m}\big)\,(x_k - x^\star) + O(|\cdot|^2) & (\text{multiplicity } m > 1) \end{cases}$$

imply that Newton's method converges for all initial guesses x_0 close enough to a root $x^\star$.

This result is important because we shall see next that Newton's method does not converge for all inputs $\big(f(x), x_0\big)$.

Nonlinear Equations

The drawback of beating bisection

Recall from slide 13 that any root-finding algorithm outperforming the bisection method must necessarily suffer from other drawbacks.

I will next present an abstract argument which shows that in the case of Newton's method, the drawback is that Newton's method may fail to converge for some inputs $\big(f(x), x_0\big)$.

We will see later that the possibility of non-convergence leads to significant complications when applying Newton's method to real-world problems. The point of the below argument is therefore to point out that these complications are due to fundamental mathematical limitations, not lack of ingenuity of mathematicians developing root-finding algorithms.

Nonlinear Equations

The drawback of beating bisection (continued)

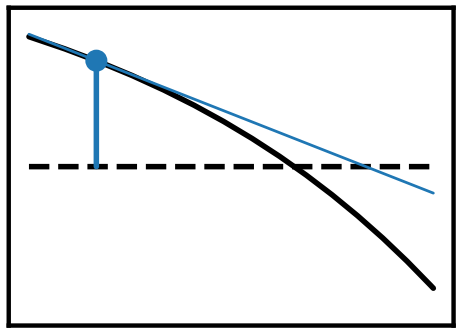
Let us now dive into the aforementioned argument.

According to our findings from slide 13, a root-finding algorithm can only outperform bisection if

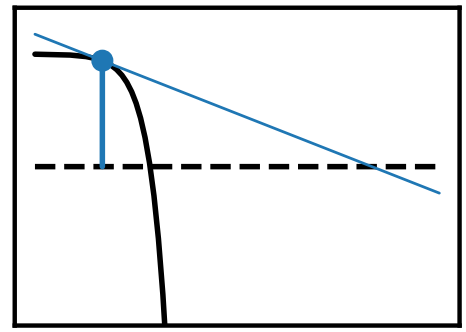
1. it exploits more information than only function values $f(x)$, or
2. it performs worse than bisection on some inputs.

Newton's method clearly exploits extra information about $f(x)$ in the form of derivatives, but it is not enough to explain why Newton's method beats bisection. For example, knowing that $f(x_k)$ is large and $f'(x_k)$ is small *suggests* that any roots of $f(x)$ must be far from x_k, but it does not *guarantee* that there cannot be a root arbitrarily close to x_k.

Expection



Possible reality



Nonlinear Equations

The drawback of beating bisection (continued)

Newton's method therefore still suffers from the problem that any function evaluation gives us at most binary information regarding the location of the roots, and hence the decision tree associated with Newton's method must be at least as deep as the decision tree of the bisection method.

More precisely, Newton's method does not even extract binary information from the function evaluations since it is not based on bracketing intervals.

We therefore conclude that Newton's method must perform worse than bisection on some inputs. But the $x_{k+1} - x^\star = O\big((x_k - x^\star)^2\big)$ convergence estimate from slide 16 tells us that Newton converges much faster than bisection if it converges; so the only way for Newton to perform worse than bisection is if it does not converge at all.

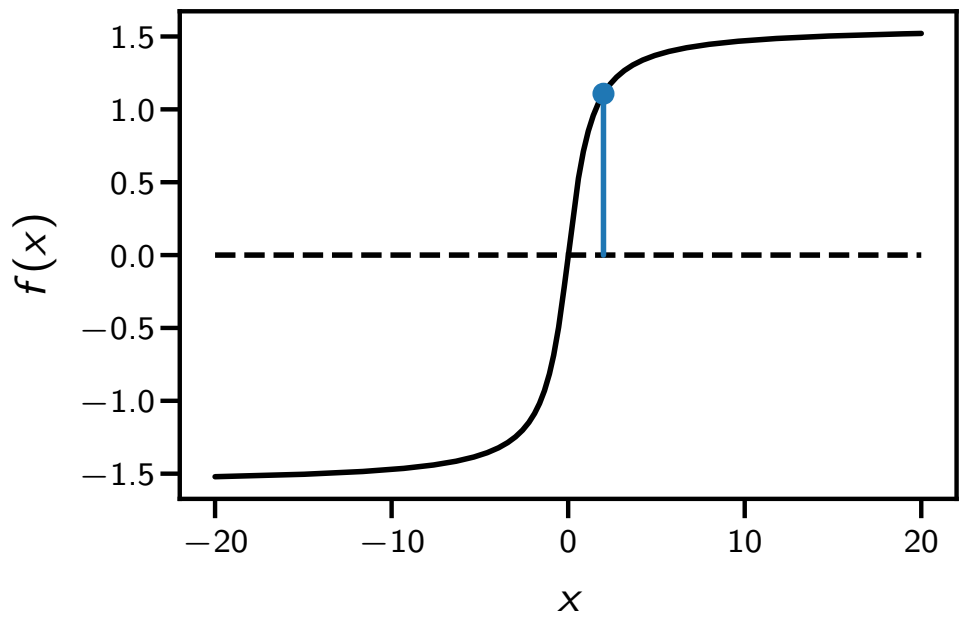
The following slide present a concrete example of an input $\big(f(x), x_0\big)$ which leads to divergence of Newton's method.

Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \text{atan}(x), \quad x_0 = 2.0$$

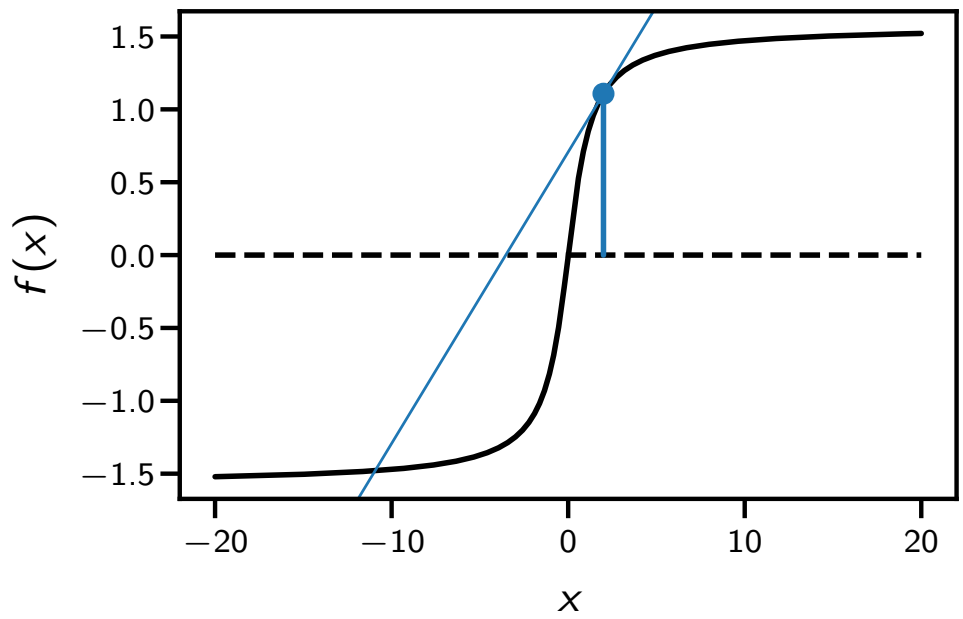


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$f(x) = \text{atan}(x), \quad x_0 = 2.0$

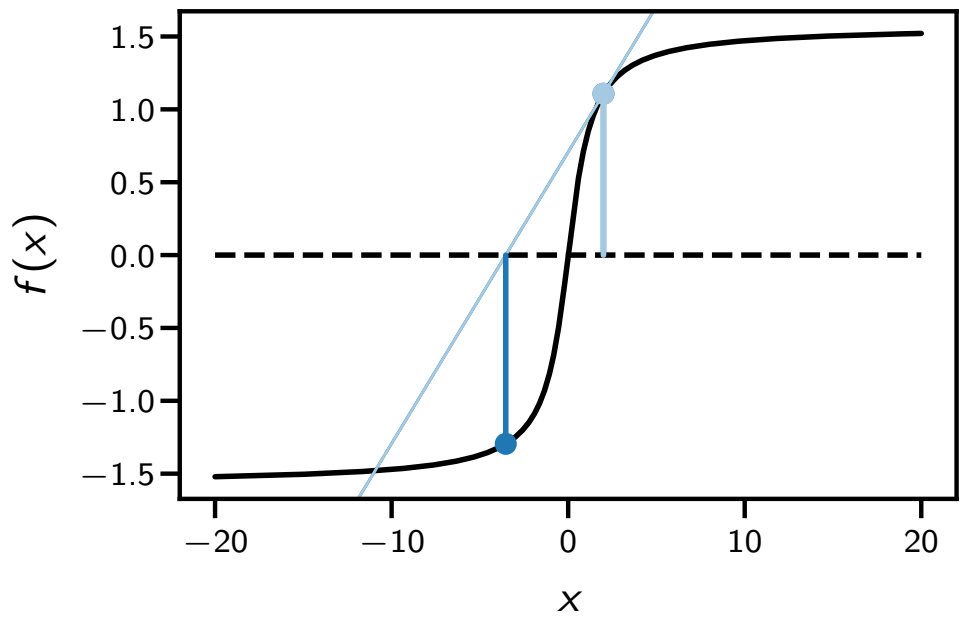


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \operatorname{atan}(x), \quad x_0 = 2.0$$

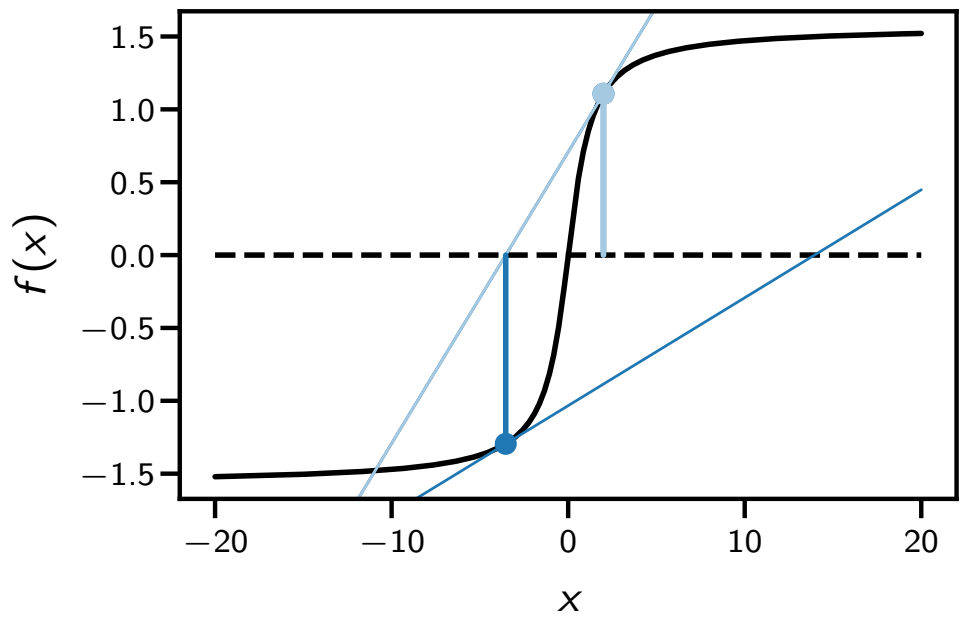


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$f(x) = \text{atan}(x), \quad x_0 = 2.0$

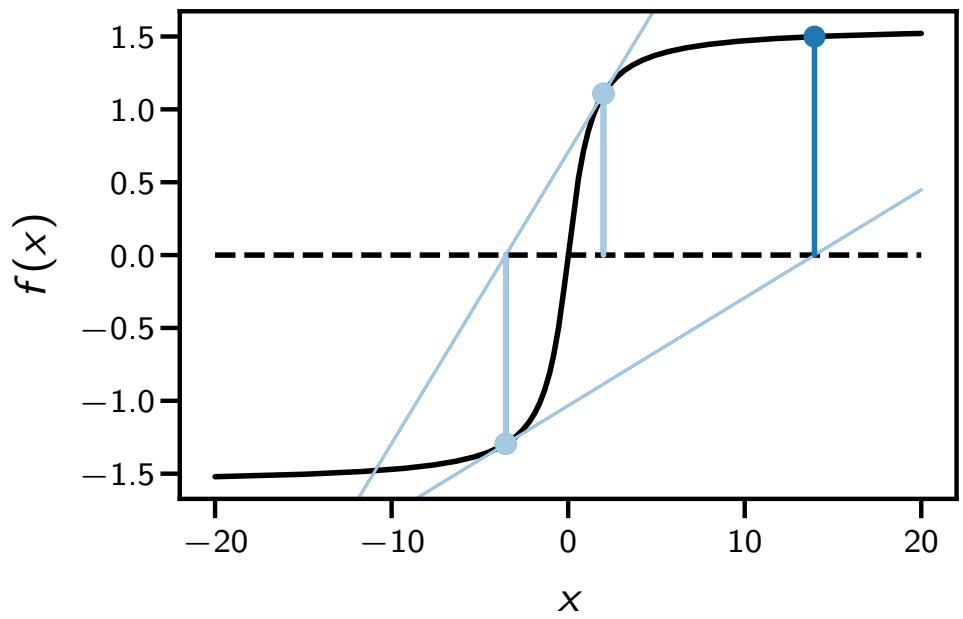


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$$f(x) = \operatorname{atan}(x), \quad x_0 = 2.0$$

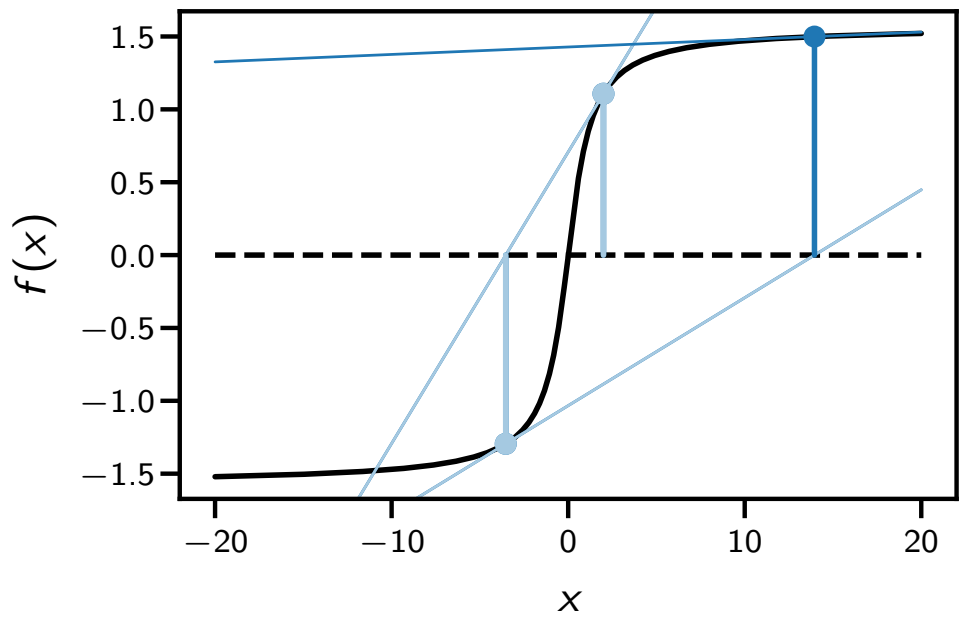


Nonlinear Equations

Algorithm Newton's method

1: Start with any x_0.
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: Compute tangent of $f(x)$ at x_k.
4: Update $x_{k+1} =$ [root of tangent]
5: **end for**

$f(x) = \text{atan}(x), \quad x_0 = 2.0$



Nonlinear Equations

Newton's method in practice

It follows from the fact that Newton's method diverges for some inputs $\big(f(x), x_0\big)$ that at least one of the following conditions must be true whenever we use Newton's method in applications.

1. We have a mathematical argument which shows that Newton's method converges for all inputs $\big(f(x), x_0\big)\big)$ of interest.
2. We have a mechanism for detecting when Newton's method diverges and fall back on another root-finding method (e.g. bisection) when this happens.
3. We accept that Newton's method may run forever and trust on our human intuition to handle such cases.

Option 3 is rarely acceptable because humans are expensive and slow.

Option 2 can often be made to work reasonably well, but it follows from the abstract argument on slide 13 that this strategy cannot beat bisection in terms of worst-case performance.

We therefore conclude that Option 1 is the best of the options above, and it turns out that proving convergence of Newton's method is indeed often possible. However, the arguments for doing so are specific to the particular input $\big(f(x), x_0\big)$, so I will not discuss them here.

Nonlinear Equations

Termination criteria for Newton's method

Recall: Newton's method is an iterative algorithm which takes in an initial guess x_0 and iteratively refines this guess until it is "good enough". We must therefore provide a criterion which allows the computer to recognise such "good enough" guesses and terminate the iteration.

We encountered the same problem already for the bisection method. There, the solution was to terminate the iteration once $|b_k - a_k| \leq 2\tau$ and return $m_k = \frac{b_k + a_k}{2}$, because doing so guarantees $|m_k - x^\star| \leq \tau$.

Unfortunately, there is no termination criterion for Newton's method which can guarantee a similar error bound. Instead, I will discuss several termination criteria, each of which has some merits but also some drawbacks.

Nonlinear Equations

Termination criteria for Newton's method (continued)

- *Vanishing updates*

  The closest we can get to an error bound is to conclude that since $x_{k+1} - x^\star = O\big(|x_k - x^\star|^2\big)$, we should have

  $$|x_{k+1} - x_k| \approx |x^\star - x_k| \tag{1}$$

  and thus we can use the left-hand side as an estimate for the error in x_k. This often works in practice, but 1) we have no guarantee that (1) indeed holds, and 2) this condition may never be satisfied if the iteration diverges.

- *Excessive runtime*

  To protect against divergence, many software packages allow the user to specify an upper bound on the number of iterations and/or functions evaluations such that the iteration is aborted if this upper bound is exceeded. The drawback of this solution is that now your code must be able to handle outputs $x_{k_{\max}}$ which may be far from being a root.

Nonlinear Equations

Termination criteria for Newton's method (continued)

- *Vanishing function values*

  In some applications, we are not interested in guarantees that the distance between x_k and an exact root $x^\star$ is small, but rather that $|f(x_k)|$ is small.

  Example: If x are model parameters and $f(x)$ measures the model error, then we may only need that $|f(x)|$ is small to guarantee that the model makes accurate predictions.

  In such applications, it may be appropriate to terminate the iteration once $|f(x_k)|$ is smaller than some specified threshold $\tau > 0$.

Example

Julia's root-finding package `Roots.jl` allows the user to specify a combination of these criteria; see `?Roots.assess_convergence()`.

Nonlinear Equations

Discussion

Based on the discussion so far, you might conclude that the comparison between the bisection and Newton methods goes as follows.

- Advantage bisection: guaranteed convergence.
- Advantage Newton: sometimes faster than bisection.

This comparison is correct but misses an important point:

- Newton's method applies to functions $f : \mathbb{R}^n \to \mathbb{R}^n$ for any n while bisection only applies if $n = 1$.

This last point makes some claims which require further justification:

1. Why does bisection not apply to $n > 1$ equations?
2. How can we generalise Newton's method to $n > 1$ equations?

Question 1 is easily addressed: bisection does not work in the multidimensional because the notion of bracketing intervals does not generalise to $n > 1$.

Question 2 is addressed on the next slide.

Nonlinear Equations

Newton's method for multidimensional root finding

Generalising Newton's method to $n > 1$ equations is straightforward: we simply replace

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \qquad \text{with} \qquad x_{k+1} = x_k - \nabla f(x_k)^{-1} f(x_k).$$

One can then show using arguments analogous to the above that

$$x_{k+1} - x^\star = O\big((x_k - x^\star)^2\big) \qquad \text{for } x_k \to x^\star$$

is still true assuming that $\nabla f(x^\star)$ is not singular.

Actually doing so is quite technical because we have

$$f(x) \in \mathbb{R}^n, \qquad \nabla f(x) \in \mathbb{R}^{n \times n}, \qquad \nabla^2 f(x) \in \mathbb{R}^{n \times n \times n}$$

and thus generalising the proof on slide 16 to $n > 1$ equations requires some notation for working with "cubes" of numbers which is beyond the scope of this module.

All the other points of the above discussion (guaranteed local convergence, possible divergence for some inputs, termination criteria) can be generalised similarly.

The fact that Newton's method works in any dimension is perhaps the most important reason why this method is in wide-spread use today.

Nonlinear Equations

Summary

- Bisection method:

$$[a_{k+1}, b_{k+1}] = \begin{cases} [a_k, m_k] & \text{if } \operatorname{sign}\big(f(a_k)\big) \neq \operatorname{sign}\big(f(m_k)\big), \\ [m_k, b_k] & \text{otherwise.} \end{cases}$$

  Error recursion: $|b_{k+1} - a_{k+1}| = \frac{1}{2}\,|b_k - a_k|$.

  Good: Simple, reliable and in some sense optimal.

  Bad: Only applies to scalar root finding.

- Newton's method: $x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)}$

  Error recursion: $x_{k+1} - x^\star = O\big((x_k - x^\star)^2\big)$ if $f'(x^\star) \neq 0$.

  Good: Very fast and applies to root finding in any dimension.

  Bad: May fail to converge.

Nonlinear Equations

Recommended exercise

Use Newton's method to compute complex square roots

$$x + iy = \sqrt{u + iv} \qquad \Longleftrightarrow \qquad (x + iy)^2 = u + iv.$$

To do so, you must translate the equation on the right into a system of nonlinear equations $f(x) = 0$ where $f : \mathbb{R}^2 \to \mathbb{R}^2$, manually compute the Jacobian $\nabla f(x)$ and then implement the Newton iteration

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

in Julia (you can use \ to solve the linear system).

A reference implementation of this algorithm is provided in `square_root()`.