

[Click here to view the PDF version.](#)

```
1 begin
2   using Plots
3   using PlutoUI
4   using PlutoTeachingTools
5   using LaTeXStrings
6   using Printf
7   using ForwardDiff
8   using LinearAlgebra
9   using HypertextLiteral: @html, @html_str
10 end
```

## ☰ Table of Contents

### Root finding and fixed-point problems ⇄

Revisiting the diode model ⇄

Fixed-point iterations ⇄

Visualising fixed-point iterations ⇄

Convergence analysis ⇄

Stopping criteria and residual ⇄

Convergence order ⇄

Optional: Bisection method ⇄

Newton's method ⇄

Achieving higher-order convergence ⇄

Construction of Newton's method ⇄

Graphical interpretation ⇄

Convergence analysis ⇄

Implementation ⇄

Lessons to learn from fixed-point iterations ⇄

Optional: Secant method ⇄

Non-linear equation systems ⇄

# Root finding and fixed-point problems



We saw in the introduction that problems where one wants to find the **root** or **zero** of a function  $f$  arise rather naturally in scientific questions. Moreover as soon as  $f$  is more complicated than just a simple polynomial, it becomes **quickly challenging to find its roots** using pen and paper. We thus have to develop numerical methods for solving such problems.

Let us first define the problem formally:

## Definition: Rootfinding problem

Given a continuous scalar function  $f : [a, b] \rightarrow \mathbb{R}$  find a value  $x_* \in [a, b]$  such that

$$f(x_*) = 0.$$

Such a value  $x_*$  is called a **root** of  $f$ .

In fact an equivalent and sometimes more intuitive way to think about solving equations  $f(x) = 0$  is to recast them as a fixed point-problem:

## Definition: Fixed-point problem

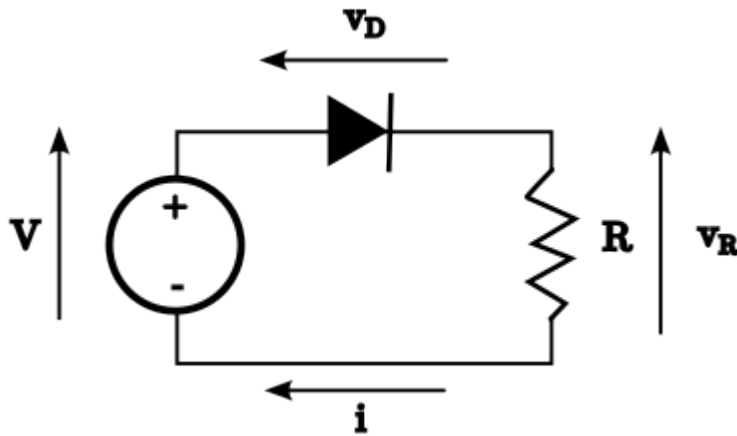
Given a continuous function  $g : [a, b] \rightarrow \mathbb{R}$  a point  $x_*$ , such that  $g(x_*) = x_*$ , is called a **fixed point** of  $g$ . The problem of finding a fixed point of  $g$  is equivalent to finding a root of the function  $f(x) = g(x) - x$ .

Such transformations can also be reversed, i.e. given an  $f$  for root finding, we could also seek a fixed-point of  $g(x) = x + f(x)$ . Moreover the transformations are not unique. Indeed, for any  $c \neq 0$  finding a fixed-point  $x_*$  of  $g(x) = x + cf(x)$  implies  $f(x_*) = 0$ .

To start of our investigation we will explore different ways of numerically solving our circuit problem.

# Revisiting the diode model ⇔

Recall the circuit diagram of the diode model



where we had the relationships

$$\begin{aligned} V &= v_D + v_R \\ v_R &= Ri \\ i &= i_0(e^{v_D/v_0} - 1) \end{aligned} \quad \text{(Shockley diode model)} \quad (1)$$

and wanted to solve for the diode voltage  $v_D$ . We consider two iterative methods to find it, motivated from the physical setting.

## Method 1

- In many physical settings one often already has a good general idea about what  $v_D$  should be like. In this case a reasonable starting point would be to assume that the diode is open and thus the voltage across the diode is zero, thus we set  $v_D^{(0)} = 0$ .
- Given the voltage  $v_D^{(0)} = 0$  across the diode, we find the voltage across the resistor as  $v_R^{(0)} = V - v_D^{(0)}$  and thus the current

$$i^{(0)} = v_R^{(0)} / R = \frac{V - v_D^{(0)}}{R}$$

- Since the current in all elements of the circuit is the same, we can estimate the voltage across the diode by reversing the Shockley diode model:

$$v_D^{(1)} = v_0 \log \left( \frac{V - v_D}{Ri_0} + 1 \right)$$

We hope for this new estimation  $v_D^{(1)}$  to be better than the initial one  $v_D^{(0)}$ .

- Finally we close the loop and repeat the process from top to bottom using  $v_D^{(1)}$  instead of  $v_D^{(0)}$ , thus obtaining  $v_D^{(2)}$ , etc. The resulting sequence  $v_D^{(0)}, v_D^{(1)}, v_D^{(2)}$  we hope to converge to the true diode voltage, that is

$$\lim_{k \rightarrow \infty} v_D^{(k)} = v_D \quad (\text{hopefully})$$

Mathematically, the voltage estimate obtained in the  $k$ -th iteration is given by

$$v_D^{(k+1)} = g_{\log}(v_D^{(k)}) \quad \text{for } k = 1, 2, \dots,$$

where

$$g_{\log}(x) = v_0 \log \left( \frac{V - x}{R i_0} + 1 \right). \quad (2)$$

If the method converges to a value  $v_D$ , then  $v_D = g_{\log}(v_D)$ , i.e. we found a fixed point of  $g_{\log}$ .

Let's see if this actually works for our example. We select the parameters:

- $i_0 = 1.0$
- $v_0 = 0.1$
- $R = 1.0$
- $V = 1.0$

and define the fixed-point map

```
glog (generic function with 1 method)
1 function glog(vD)
2     v0 * log((V - vD) / (R * i0) + 1)
3 end
```

finally we iterate 10 steps, recording along the way the produced iterates

```

1 let
2   vD = 0.1
3   for i in 1:10
4     vD = glog(vD) # Call fixed-point map
5     @printf "iter %2i: vD = %.15f\n" i vD # Print formatted data
6   end
7 end

```

```

iter  1: vD = 0.064185388617239
iter  2: vD = 0.066052822568595
iter  3: vD = 0.065956308405801
iter  4: vD = 0.065961298808626
iter  5: vD = 0.065961040778816
iter  6: vD = 0.065961054120317
iter  7: vD = 0.065961053430491
iter  8: vD = 0.065961053466159
iter  9: vD = 0.065961053464315
iter 10: vD = 0.065961053464410

```

Hooray! After 10 steps about 12 significant figures have stabilised and we thus expect the error to be smaller than  $10^{-12}$ .

But setting up such iterations is in fact not unique:

## Method 2

- Assume again a  $v_D^{(0)}$  is provided by physical intuition.
- Using the Shockley relation we obtain the current through the diode as  $i^{(0)} = i_0 \left( e^{v_D^{(0)}/v_0} - 1 \right)$ , which again equals the current through the resistor.
- The voltage across the resistor is thus  $v_R^{(0)} = R i^{(0)}$ , from which we can compute an updated voltage estimation across the diode as  $v_D^{(1)} = V - v_R^{(0)}$

Again closing the loop and iterating multiple times, we obtain at the  $k$ -th iteration

$$v_D^{(k+1)} = g_{\text{exp}}(v_D^{(k)}) \quad \text{for } k = 1, 2, \dots,$$

where

$$g_{\text{exp}}(x) = V - R i_0 \left( e^{x/v_0} - 1 \right). \quad (3)$$

Let's test this method as well:

gexp (generic function with 1 method)

```
1 function gexp(vD)
2     V - R * i0 * (exp(vD/v0)-1)
3 end
```

```
1 let
2     vD = 0.1
3     for i in 1:10
4         vD = gexp(vD) # Call fixed-point map
5         @printf "iter %2i:  vD = %.15f\n" i vD # Print formatted data
6     end
7 end
```

```
iter 1:  vD = -0.718281828459045
iter 2:  vD = 1.999240475732571
iter 3:  vD = -481494204.686199128627777
iter 4:  vD = 2.000000000000000
iter 5:  vD = -485165193.409790277481079
iter 6:  vD = 2.000000000000000
iter 7:  vD = -485165193.409790277481079
iter 8:  vD = 2.000000000000000
iter 9:  vD = -485165193.409790277481079
iter 10: vD = 2.000000000000000
```

Even though the method seems equally plausible as method 1 on first sight, it clearly performs worse and does not converge at all. For finding the voltage  $v_D$  across the diode it is thus not useful.

Let us now formalise these methods mathematically in order to analyse them more carefully.

## Fixed-point iterations ⇔

As discussed in the previously the equations of the diode model (1) lead to the non-linear problem

$$f(v_D) = R i_0 \left( e^{v_D/v_0} - 1 \right) + v_D - V, \quad (4)$$

where its root is the desired diode voltage. In both **Method 1** and **Method 2** we effectively rewrote this equation into a different fixed-point problem:

$$\begin{aligned} g_{\log}(v_D) &= v_D & g_{\log}(x) &= v_0 \log \left( \frac{V - x}{R i_0} + 1 \right) \\ g_{\exp}(v_D) &= v_D & g_{\exp}(x) &= V - R i_0 \left( e^{x/v_0} - 1 \right) \end{aligned}$$

To solve these fixed-point problems we then applied

### Algorithm: Fixed-point iteration

Given a fixed-point map  $g$  and initial guess  $x^{(0)}$ , iterate

$$x^{(k+1)} = g(x^{(k)}) \quad \text{for } k = 1, 2, \dots$$

until a stopping criterion is reached.

If  $\lim_{k \rightarrow \infty} x^{(k)} = x_*$  for the sequence  $x^{(k)}$  generated by Algorithm 1, then  $g(x_*) = x_*$ , i.e.  $x_*$  is a fixed point of  $g$ . At this point we do not yet specify what is a good stopping criterion. We will return to this point in the *Convergence analysis* section.

If we are faced with a fixed point problem (finding  $x_*$  such that  $g(x_*) = x_*$ ) then Algorithm 1 can be directly applied. However, to **apply the fixed-point method** to a **root-finding problem** (seek  $x_*$  s.t.  $f(x_*) = 0$ ) we first need to **rewrite the non-linear equation  $f(x) = 0$**  into a fixed-point problem, i.e. identify a suitable  $g$ , such that if  $x_*$  is a root of  $f$ , then

$$f(x_*) = 0 \quad \Longleftrightarrow \quad x_* = g(x_*).$$

On  $g$  we then apply fixed-point iteration.

We will see an example of this rewriting in the next section.

## Visualising fixed-point iterations $\Leftrightarrow$

Before we proceed to a closer mathematical analysis of fixed-point methods and ultimately to understand the difference between employing  $g_{\log}$  and  $g_{\exp}$ , let us first consider a simpler case, where it is easier to obtain a visual understanding.

We consider solving non-linear equation  $f(x) = 0$  with

$$f(x) = x + \log(x + 1) - 2$$

In this one can easily construct four equivalent fixed-point equations  $x = g_i(x)$  with  $i = 1, 2, 3, 4$ , namely

```
1 begin
2    $g_1(x) = x - (1/2) * (\log(x + 1) + x - 2)$ 
3    $g_2(x) = 2 - \log(x + 1)$ 
4    $g_3(x) = \exp(2-x) - 1$ 
5    $g_4(x) = (1/2) * x * (\log(x+1) + x)$ 
6 end;
```

### Example: Rewriting $f(x)=0$ as a fixed point problem

We are given the problem to solve  $f(x) = 0$  with  $f(x) = x + \log(x + 1) - 2$ . We show how to construct  $g_3$  and  $g_4$ .

First  $g_3$ . At convergence we have that

$$\begin{aligned} 0 &= f(x) = x + \log(x + 1) - 2 && \text{(add } 2 - x) \\ \Leftrightarrow 2 - x &= \log(x + 1) && \text{(take exp)} \\ \Leftrightarrow \exp(2 - x) &= x + 1 && \text{(subtract 1)} \\ \Leftrightarrow \exp(2 - x) - 1 &= x \end{aligned}$$

This is now a fixed-point problem in  $x$  where we seek a fixed point of the function  $g_3(x) = \exp(2 - x) - 1$ , which is just the left-hand side of the expression.

Now  $g_4$ . Again starting from

$$\begin{aligned} 0 &= f(x) = x + \log(x + 1) - 2 && \text{(add 2)} \\ \Leftrightarrow 2 &= x + \log(x + 1) && \text{(divide by 2)} \\ \Leftrightarrow 1 &= \frac{1}{2}(x + \log(x + 1)) && \text{(multiply by } x) \\ \Leftrightarrow x &= \underbrace{\frac{x}{2}(x + \log(x + 1))}_{g_4(x)} \end{aligned}$$

where again we get a problem  $x = g_4(x)$ , i.e. finding a fixed point of  $g_4$ .

To visualise the fixed-point iterations, we choose yet another understanding of fixed point problems:

### Observation: Fixed point problems are about curve intersections

We can understand a fixed-point problem, where we seek an  $x_*$  such that  $g(x_*) = x_*$  as the problem of finding the **intersection of the curve**  $y = g(x)$  with the **line**  $y = x$ . In other words we want to find a point  $(x, y)$  such that  $y = g(x)$  and at the same time  $x = y$ .


The following plot uses this observation to visualise fixed-point iterations. The idea is to think of the fixed-point iterations in the following way:

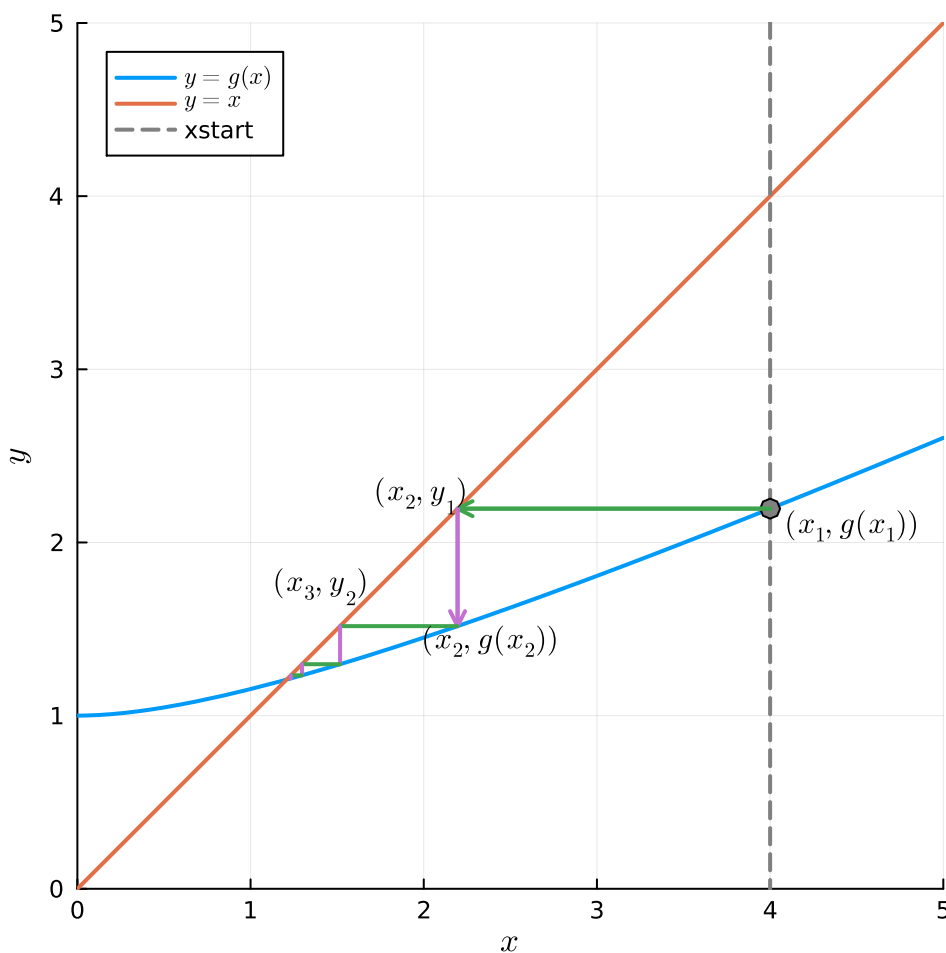
- We start at some  $x_1 = \text{'xstart'}$ . Then we use the condition  $y = g(x)$  to arrive at the first, grey point, that is  $(x_1, y_1) = (x_1, g(x_1))$ .



- From there we use the condition  $x = y$ , i.e.  $x_2 = y_1$  to arrive at the point  $(x_2, y_1) = (y_1, y_1)$  (green arrow)
- Next, we again use  $y = g(x)$  and obtain  $y_2 = g(x_2)$ , thus the point  $(x_2, g(x_2))$  (purple arrow).
- From there we continue setting  $x_3 = y_2$  to get  $(x_3, y_2)$
- And we continue analogously until we hopefully converge.

Use the slider and the drop-down menu to switch between the fixed-point functions and the starting point to see if this works.

- xstart =  4.0
- g =
- Show labels: ☒
- Show gradient: ☐



We notice that that  $g_1$  and  $g_2$  converge, while  $g_3$  and  $g_4$  do not.

Their distinctive feature becomes clear when we also show the gradient of  $g$  at the fixed point (click the checkbox above).

Convergence occurs when the slope is between  $-1$  and  $1$ , i.e. the graph sits between the regions spanned by the orange lines of slope  $1$  and  $-1$  through the fixed point.

## Convergence analysis $\Leftrightarrow$

We consider the setting finding a fixed-point  $x_* = g(x_*)$  for a differentiable function  $g$ . Our hypothesis is that the fixed-point method converges if  $g'(x_*) \in (-1, 1)$ , while it diverges when  $|g'(x_*)| > 1$ .

To establish this result more rigorously we study the behaviour of the error sequence

$$e^{(k)} = x^{(k)} - x_*$$

The goal is to find an expression that relates  $e^{(k+1)}$  with  $e^{(k)}$  and in this way *recursively* to  $e^{(0)}$ , the error of the initial guess.

First we note  $x^{(k)} = x_* + e^{(k)}$  and we develop the Taylor expansion of  $g$  around  $x_*$ :

$$g(x^{(k)}) = \underbrace{g(x_*)}_{=x_*} + g'(x_*)(x^{(k)} - x_*) + R(x^{(k)}) \quad (5)$$

where

$$R(x) = O(|x - x_*|^2) \quad \text{as } x \rightarrow x_*,$$

which means that there exist positive constants  $M, \delta > 0$  such that

$$|R(x)| \leq M|x - x_*|^2 \quad \forall 0 < |x - x_*| < \delta,$$

i.e. that  $R(x)$  stays bounded by the next term of the Taylor expansion (up to a multiplicative constant). The notation  $O(|x - x_*|^2)$  is thus a mathematically precise way of saying that there are more terms that we don't show but their order is at most  $|x - x_*|^2$ . See also the discussion in [Revision and preliminaries](#) on Taylor approximations.

Using (5) and the key fixed-point iterations equation,  $x^{(k+1)} = g(x^{(k)})$  we obtain

$$\begin{aligned}
e^{(k+1)} &= x^{(k+1)} - x_* = g(x^{(k)}) - x_* \\
&\stackrel{(5)}{=} x_* + g'(x_*) \underbrace{(x^{(k)} - x_*)}_{=e^{(k)}} + O(|e^{(k)}|^2) - x_* \\
&= g'(x_*) e^{(k)} + O(|e^{(k)}|^2).
\end{aligned}$$

Taking moduli on both sides:

$$|e^{(k+1)}| = |g'(x_*)| |e^{(k)}| + O(|e^{(k)}|^2)$$

We employ this relation now in a recursive argument. Assume we choose a good initial guess, then  $x^{(0)}$  is close enough to  $x_*$ , such that  $O(|e^{(0)}|^2)$  is negligible compared to  $|g'(x_*)| |e^{(0)}|$ . Similarly, provided that the iteration makes progress,  $O(|e^{(1)}|^2)$  is in turn smaller than  $|g'(x_*)| |e^{(1)}|$  and so forth. Therefore

$$\begin{aligned}
|e^{(k+1)}| &= |g'(x_*)| |e^{(k)}| + O(\text{small}) \\
&= |g'(x_*)|^2 |e^{(k-1)}| + O(\text{small}) \\
&= \dots \\
&= |g'(x_*)|^{k+1} |e^{(0)}| + O(\text{small})
\end{aligned}$$

In other words as  $k \rightarrow \infty$ , i.e. the iteration progresses,  $|e^{(k+1)}|$  approaches zero if  $|g'(x_*)| < 1$ , exactly as we concluded from the plot. Our argument proves the following

### Theorem 1

Let  $g : [a, b] \rightarrow \mathbb{R}$  be a function of class  $C^1$  [1] and  $x_* \in [a, b]$  be a fixed point of  $g$ . If  $|g'(x_*)| < 1$ , then there exists an  $\varepsilon > 0$ , such that for all  $x^{(0)} \in [x_* - \varepsilon, x_* + \varepsilon]$

- the fixed-point iterations  $x^{(k+1)} = g(x^{(k)})$  converge to  $x_*$ , i.e.  $\lim_{k \rightarrow \infty} x^{(k)} = x_*$
- Moreover the **convergence rate** (formal definition below) is given by

$$\lim_{k \rightarrow \infty} \frac{|x^{(k+1)} - x_*|}{|x^{(k)} - x_*|} = |g'(x_*)|,$$

i.e. the smaller the gradient, the faster the convergence.

[1]:

differentiable with continuous first derivative.

## Exercise

Verify the theorem for the fixed-point problems we considered so far, i.e. show that

- The map  $g_{\log}$  has a gradient modulus less than 1 at the fixed point while  $g_{\exp}$  has one larger 1.
- similarly verify analytically the convergence of the fixed-point iterations of  $g_1$  and  $g_2$  and the divergence of  $g_3$  and  $g_4$ .

## Stopping criteria and residual $\Leftrightarrow$

Let us come back to the question at which point to stop the fixed-point iteration algorithm. Let  $\epsilon$  denote the tolerance to which we want to determine  $x_*$ , i.e. we would like to stop the iterations as soon as the error is smaller,  $|x^{(k)} - x_*| < \epsilon$ .

Since  $x_*$  is not known, this expression cannot be exactly computed during the iterations. We thus need to seek an alternative approach. In a given step  $x^{(k)}$  during the iteration we have likely not yet achieved our goal, i.e.  $g(x^{(k)}) \neq x^{(k)}$ . A natural idea is thus to consider exactly the discrepancy

$$r^{(k)} = g(x^{(k)}) - x^{(k)},$$

the so-called **residual**. A natural stopping criterion is thus

### Algorithm: Fixed-point iteration stopping criterion

$$|r^{(k)}| = |g(x^{(k)}) - x^{(k)}| < \epsilon.$$

Employing this stopping criteria, the algorithm to find a fixed point of the function  $g$  becomes

fixed\_point\_iterations\_simple (generic function with 1 method)

```
1 function fixed_point_iterations_simple(g, xstart; tol=1e-6)
2     # g:      Fixed-point function
3     # xstart: Initial guess
4     # tol:    Tolerance
5
6     rk = Inf
7     xk = xstart
8     k = 0
9     while abs(rk) ≥ tol
10         xk+1 = g(xk)
11         rk = xk+1 - xk
12
13         k = k + 1 # Update k
14         xk = xk+1 # Update xk accordingly
15     end
16
17     # Return results as a named tuple
18     (; fixed_point=xk, residual=rk, n_iter=k)
19 end
```

We apply this function:

```
res_simple = ▶(fixed_point = 1.20794, residual = -7.54952e-15, n_iter = 27)
1 res_simple = fixed_point_iterations_simple(g1, 4.0; tol=1e-14)
```

Since res\_simple is now a named tuple, we can access its individual fields to e.g. get the fixed point

1.2079400315693258

```
1 res_simple.fixed_point
```

or the number of iterations required

27

```
1 res_simple.n_iter
```

In practice it is often useful to **also include a cap on the maximal number of iterations** (for cases where the algorithm does not converge) and to **record a history** of the visited points, which is useful for later analysis.

fixed\_point\_iterations (generic function with 1 method)

```
1 function fixed_point_iterations(g, xstart; tol=1e-6, maxiter=100)
2     # g:      Fixed-point function
3     # xstart: Initial guess
4     # tol:    Tolerance
5
6     history_x = [xstart]
7     history_r = empty(history_x)
8
9     rk = Inf      # For initial pass in while loop
10    xk = xstart  # Starting point of iterations
11    k = 0
12    while k < maxiter && abs(rk) ≥ tol
13        xk+1 = g(xk)
14        rk = xk+1 - xk
15        push!(history_r, rk)
16
17        k = k + 1 # Update k
18        xk = xk+1 # Update xk accordingly
19        push!(history_x, xk) # Push next point to the history
20    end
21
22    # Return results as a named tuple
23    (; fixed_point=xk, residual=rk, n_iter=k, history_x, history_r)
24 end
```

► (fixed\_point = 1.20794, residual = -7.54952e-15, n\_iter = 27, history\_x = [4.0, 2.19528, 1.!

```
1 fixed_point_iterations(g1, 4.0; tol=1e-14)
```

### Additional remarks on the residual

- The residual is in general only an **error indicator**. This means that **there is no guarantee** that  $|g(x^{(k)}) - x^{(k)}| < \epsilon$  always implies  $|x^{(k)} - x_*| < \epsilon$ .
- In fact we can derive the **residual-error relationship** (see derivation below)

$$|x^{(k)} - x_*| = \frac{1}{|1 - g'(\xi^{(k)})|} |r^{(k)}|. \quad (6)$$

for some  $\xi^{(k)} \in [x_*, x^{(k)}]$ . Note, that this is just a **conceptual expression** as determining  $\xi^{(k)}$  is in general *as hard* as finding  $x_*$ . But it will be useful in some theoretical arguments.

- For converging iterations  $x^{(k)} \rightarrow x_*$  as  $k \rightarrow \infty$ . Therefore the interval  $[x_*, x^{(k)}]$  gets smaller and smaller, such that necessarily  $\xi^{(k)} \rightarrow x_*$  and  $g'(\xi^{(k)}) \rightarrow g'(x_*)$  as  $k \rightarrow \infty$ . We note it is the **gradient at the fixed point**,  $g'(x_*)$ , which determines **how reliable our error indicator** is.

- In particular if  $|g'(x_*)|$  is close to 1, then the denominator of the prefactor may blow up and the **residual criterion**  $|r^{(k)}| < \epsilon$  may well **stop the iterations too early**. That is the **actual error**  $|x^{(k)} - x_*|$  may still be **way larger than the residual**  $|r^{(k)}|$  and thus way larger than our desired accuracy  $\epsilon$ .
- In contrast if  $|g'(x_*)| = 0$  then  $|r^{(k)}| < \epsilon$  is an excellent stopping criterion as  $|x^{(k)} - x_*| = |r^{(k)}|$  as  $k \rightarrow \infty$ .

### ► Derivation of the residual-error relationship

#### General principle: Residual

Note, that the **residual is a general terminology**, which is not only applied to such an error indicator in the context of fixed-point iterations, but used in general for iterative procedures. The idea is that the **residual provides the discrepancy from having fully solved the problem** and is thus a natural error indicator. The **functional form**, however, is **different for each type of iterative procedure** as we will see.

## Convergence order $\Leftrightarrow$

When performing numerical methods one is usually not only interested whether an iteration converges, but also how quickly, i.e. how the error approaches zero.

#### Definition: Convergence order and rate

A sequence  $x^{(k)}$ , which converges to  $x_*$ , is said to have **convergence order**  $q$  and **convergence rate**  $C$  when there exists a  $q > 1$  and  $C > 0$ , such that

$$\lim_{k \rightarrow \infty} \frac{|x^{(k+1)} - x_*|}{|x^{(k)} - x_*|^q} = C \quad (7)$$

If  $q = 1$  (convergence order 1) we additionally require  $0 < C < 1$ .

Some remarks:

- Convergence order  $q = 1$  is also called **linear convergence**, any convergence  $q > 1$  is usually called **superlinear convergence**. In particular  $q = 2$  is called **quadratic convergence**
- These names become more apparent if one considers a logarithmic scale. Suppose for simplicity that  $q = 1$  and all ratios in (7) are equal to  $C$  (perfect linear convergence), then

$|x^{(k+1)} - x_*| = \alpha C^k$  where  $\alpha$  is some constant. Taking the logs we get

$$\log |x^{(k)} - x_*| = k \log(C) + \log(\alpha)$$

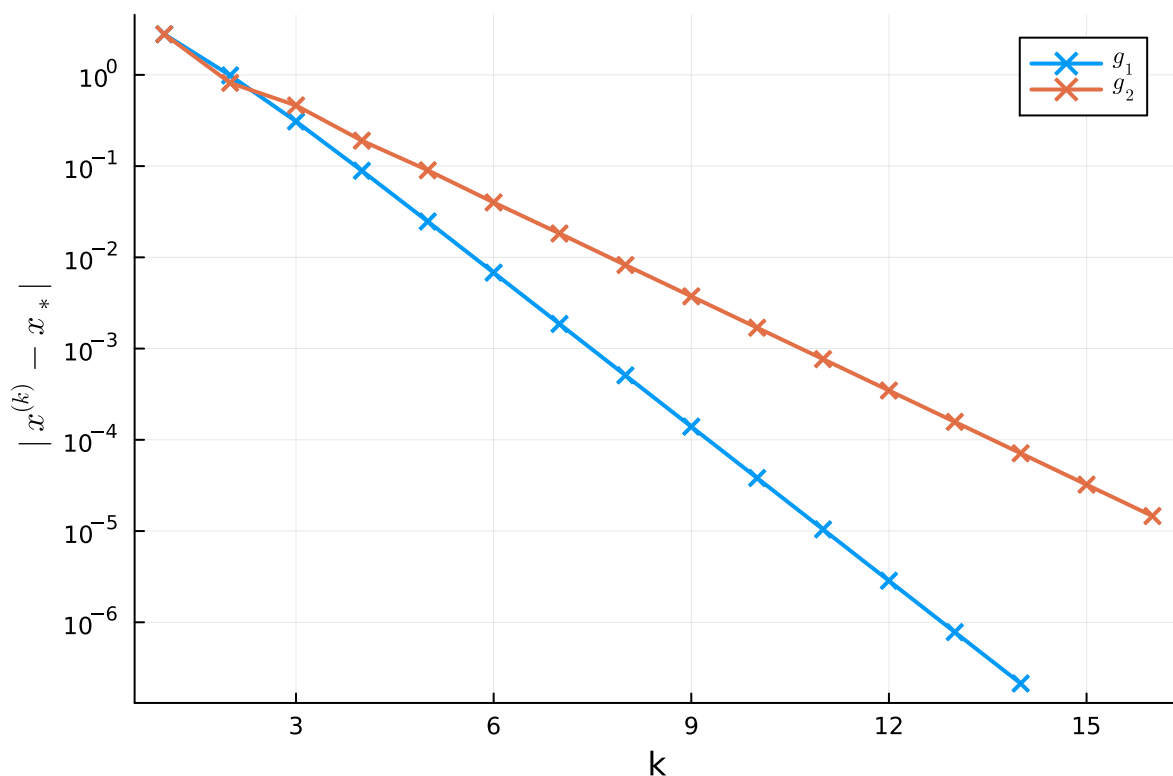
which is a straight line.

## Visual inspection: Log of error $\Leftrightarrow$

The last remark provides an idea how to **visually inspect the convergence order**, namely by plotting the error  $|x^{(k)} - x_*|$  on a logscale. For our fixed point iterations, the (hopefully) converging sequence is exactly generated by the relationship  $x^{(k+1)} = g(x^{(k)})$ .

So let us inspect the convergence of  $g_1$  and  $g_2$  graphically:





```

1 let
2   fp = 1.2079400315693 # Approximate fixed point
3   p = plot(; yaxis=:log, xlabel="k", ylabel=L"|x^{(k)} - x_{ast}|")
4
5   results_g1 = fixed_point_iterations(g1, 4.0; maxiter=15)
6   errors_g1 = [abs(xn - fp) for xn in results_g1.history_x]
7   plot!(p, errors_g1, label=L"g_1", mark=:x, lw=2)
8
9   results_g2 = fixed_point_iterations(g2, 4.0, maxiter=15)
10  errors_g2 = [abs(xn - fp) for xn in results_g2.history_x]
11  plot!(p, errors_g2, label=L"g_2", mark=:x, lw=2)
12
13  yticks!(p, 10.0 .^ (-6:0))
14 end

```

So clearly both  $g_1$  and  $g_2$  converge linearly, but  $g_1$  has a larger convergence rate.

## Visual inspection: Residual ratio $\Leftrightarrow$

One caveat with this analysis is that we cheated a little by assuming that we already *know* the solution. An alternative approach is to **build upon our residual-error relationship** (6), i.e.

$$|x^{(k)} - x_*| = \frac{1}{|1 - g'(\xi^{(k)})|} r^{(k)}.$$

and investigate the limit of the **residual ratio**

$$\lim_{k \rightarrow \infty} \frac{|r^{(k+1)}|}{|r^{(k)}|^q} = \lim_{k \rightarrow \infty} \frac{|1 - g'(\xi^{(k+1)})|}{|1 - g'(\xi^{(k)})|^q} \frac{|x^{(k+1)} - x_*|}{|x^{(k)} - x_*|^q} \stackrel{(7)}{=} \frac{1}{|1 - g'(x_*)|^{q-1}} C.$$

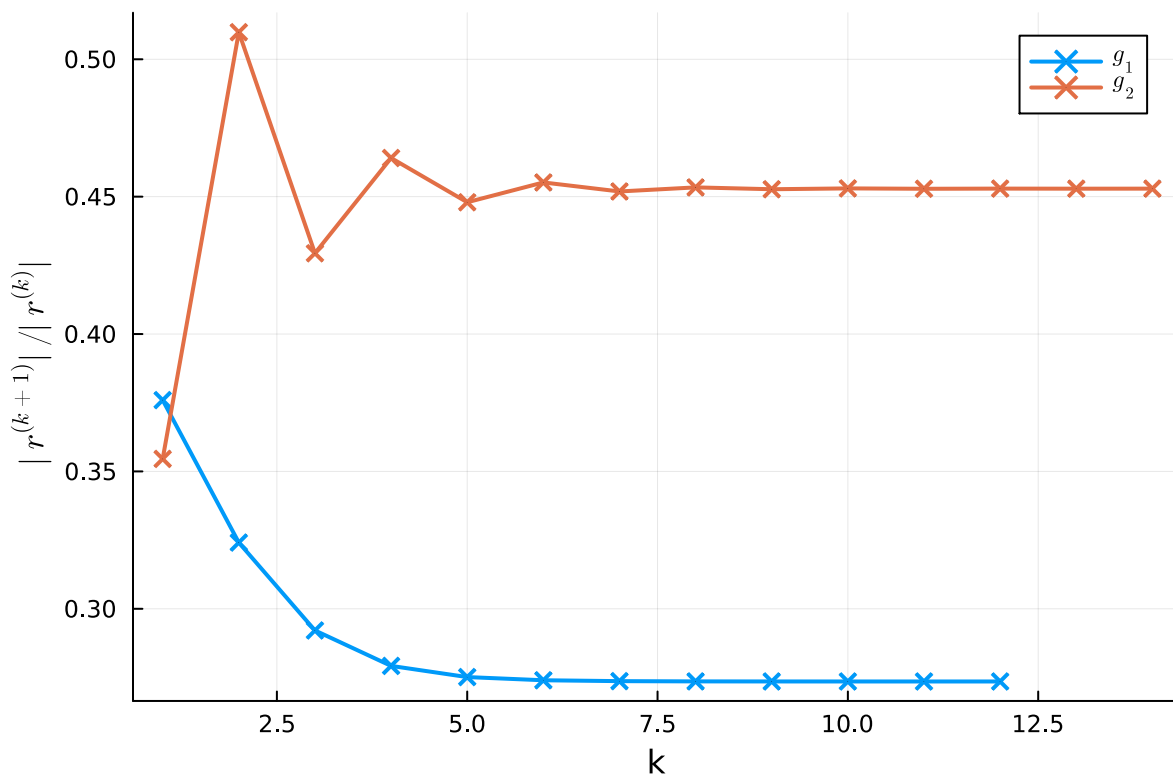
In other words if the residual ratios approach a constant for a chosen  $q$ , then we have  $q$ -th order convergence.

In particular for **linear order** ( $q = 1$ ) we have

$$\lim_{k \rightarrow \infty} \frac{|r^{(k+1)}|}{|r^{(k)}|} = C,$$

i.e. that the **residual ratio should approach a constant** as  $k \rightarrow \infty$ , which is the **convergence rate**.

This is a condition we can check also *without* knowing the solution:



```

1 let
2   p = plot(xlabel="k", ylabel=L"|r^{(k+1)}| / |r^{(k)}|")
3
4   results_g1 = fixed_point_iterations(g1, 4.0; maxiter=15)
5   residuals_g1 = results_g1.history_r
6   ratios_g1 = [abs(residuals_g1[i+1] / residuals_g1[i])
7               for i in 1:length(residuals_g1)-1]
8   plot!(p, ratios_g1, label=L"g_1", mark=:x, lw=2)
9
10  results_g2 = fixed_point_iterations(g2, 4.0, maxiter=15)
11  residuals_g2 = results_g2.history_r
12  ratios_g2 = [abs(residuals_g2[i+1] / residuals_g2[i])
13              for i in 1:length(residuals_g2)-1]
14  plot!(p, ratios_g2, label=L"g_2", mark=:x, lw=2)
15 end

```

Clearly in both cases these ratios become approximately constant as  $k$  gets larger.

## Observations

- For **linear convergence**, the **error reduces** in each iteration by a **constant factor**
- Looking at the **error norm** or **residual ratio** on a **log-scale** as the iteration proceeds is often extremely insightful to understand the convergence behaviour (and debug implementation bugs)!

## Optional: Bisection method ⇌

Fixed-point iterations are a very useful tool to solve non-linear equations. However, their condition for convergence, namely  $g'(x_*)$  is very hard to verify *a priori*, i.e. before even attempting a numerical solution of the problem we have at hand.

We will now develop a simple algorithm for root finding, which has the appealing feature, that under an easily verifiable condition, it is guaranteed to converge.

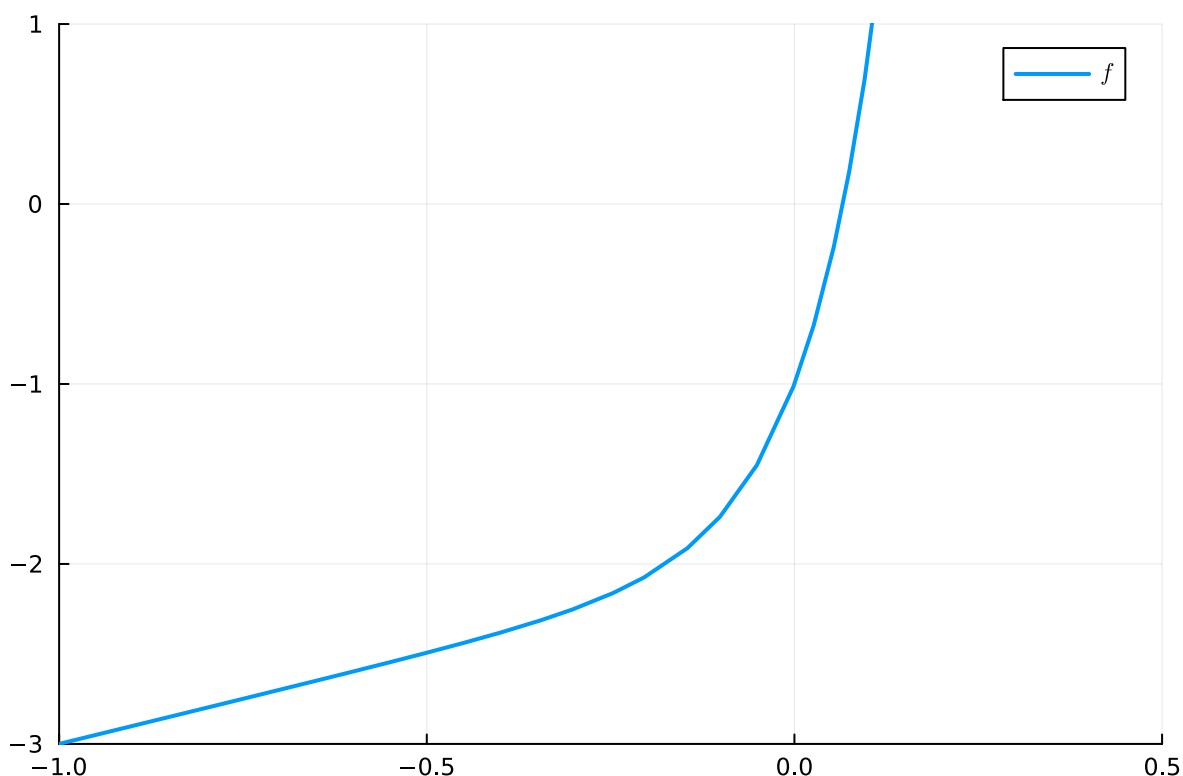
For this let us revisit our diode circuit problem. the non-linear equation to solve was

$$f(x) = R i_0 \left( e^{x/v_0} - 1 \right) + x - V.$$

We will now directly attempt to find a root  $f(x) = 0$  of this equation.

f (generic function with 1 method)

```
1 f(x) = R*i0*(exp(x/v0) - 1) + x - V
```



```
1 plot(f, ylims=(-3, 1), xlims=(-1, 0.5), label=L"f", lw=2)
```

This function has clearly a pronounced root near zero, where it changes its sign. This observation is put on more rigorous footing by the following

## Theorem 2

Let  $f$  be a continuous function defined on  $[a, b]$  with  $f(a)f(b) < 0$  (i.e. the function takes different signs at the boundary of the interval). Then there exists a  $x_* \in [a, b]$  such that  $f(x_*) = 0$ .

This is a direct consequence of the intermediate value theorem, which we now want to exploit to find a root numerically.

Suppose  $f$  on  $[a, b]$  satisfies the conditions of the theorem  $f(a)f(b) < 0$  and let us consider the midpoint of the interval  $x_m = \frac{a+b}{2}$ . There are three possible cases:

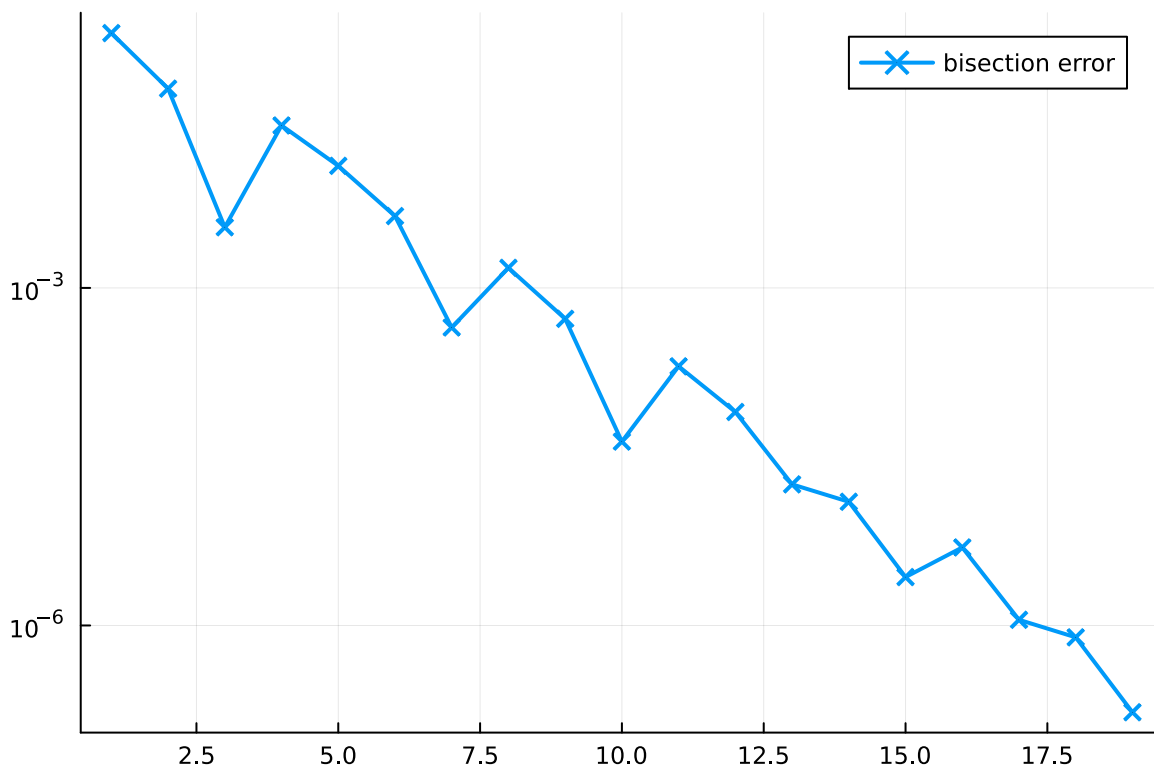
- If  $f(x_m)f(a) < 0$ , then there is a root in  $[a, x_m]$
- If  $f(x_m)f(b) < 0$ , then root is in  $[x_m, b]$
- If  $f(x_m) = 0$ , we found a root.

By simply repeating this procedure (now on the *smaller* interval  $[a, x_m]$  or  $[x_m, b]$ ) we obtain the **bisection method**:

bisection\_method (generic function with 1 method)

```
1 function bisection_method(f, a, b; tol=1e-6)
2     @assert a ≤ b
3     @assert f(a) * f(b) < 0 # Otherwise the assumptions are not true
4
5     # Initialise
6     k = 0
7     xk = (a + b) / 2
8
9     history_x = Float64[] # Empty Array, but only for Float64 numbers
10    while abs(b - a) / 2 ≥ tol
11        k = k + 1
12        if f(xk) * f(a) < 0
13            b = xk # New interval [a, xk]
14        else
15            a = xk # New interval [xk, b]
16        end
17
18        xk = (a + b) / 2
19        push!(history_x, xk)
20    end
21
22    (; root=xk, history_x, n_iter=k)
23 end
```

Its convergence plot again suggests a linear convergence:



```

1 let
2   # First get an almost exact root
3   reference = bisection_method(f, -0.5, 0.5; tol=1e-12)
4
5   # Now run again to plot convergence
6   result = bisection_method(f, -0.5, 0.5)
7   errors = [abs(xn - reference.root) for xn in result.history_x]
8
9   plot(errors, label="bisection error", mark=:x, lw=2, yaxis=:log)
10 end

```

In our suggested implementation we choose to stop either when the length of the interval  $[a, b]$  drops below the desired tolerance or when a maximal number of iterations is reached. However, for the bisection method we could even determine *a priori* how many iterations to perform as we will now demonstrate.

The bisection method starts from an interval  $I^{(0)} = [a, b]$ , which has width  $|I^{(0)}| = b - a$ . In each iteration we split the interval into two parts of equal size, therefore the interval in iteration  $k$  has size  $|I^{(k)}| = \frac{b-a}{2^k}$ . By construction the root  $x_* \in I^{(k)}$ . Our estimate for the root is always the midpoint of  $I^{(k)}$ . Therefore the error in the  $k$ -th step is bounded by

$$|x^{(k)} - x_*| \leq \frac{1}{2} |I^{(k)}| = \left(\frac{1}{2}\right)^{k+1} (b - a) \quad (7)$$

Suppose now we want to achieve an error less than  $\epsilon$ , i.e.  $|\mathbf{x}^{(k)} - \mathbf{x}_*| \leq \epsilon$ . According to (7) it is sufficient to achieve

$$\left(\frac{1}{2}\right)^{k+1} (b - a) \leq \epsilon \quad (8)$$

in order to have an error below the threshold  $\epsilon$ . Rearranging (8) leads to

$$k > \underbrace{\log_2 \left( \frac{b - a}{\epsilon} \right) - 1}_{=K},$$

which thus provides a lower bound on the number of iterations we need to achieve convergence to  $\epsilon$ . Note that  $K$  can be computed *a priori* before even starting the bisection algorithm. Moreover since (7) and (8) are *guaranteed* bounds, iterating for at least  $K$  iterations guarantees that an error below  $\epsilon$  is obtained. In comparison to our residual-based stopping criterion for the fixed point iterations, this is a much stronger result.

From our analysis we can characterise the bisection method as follows:

- If we can find an interval  $[a, b]$  where a given function  $f$  changes sign, then the bisection method almost certainly converges to a root. [2]
- We can precisely control the error up to the point where we know *a priori* how many iterations are needed.
- However, the algorithm cannot be employed if such an interval  $[a, b]$  cannot be found.

### Exercise

Proove the linear convergence of the bisection method. What is the convergence rate  $C$ ?

[2]:

Our analysis does not include the effect of finite-precision floating point arithmetic, which in theory and in practice can inhibit convergence for some tricky cases.

## Newton's method ⇨

### Achieving higher-order convergence ⇨

So far we only constructed methods with linear convergence order. In this subsection we first want to understand what is required to achieve quadratic or higher-order convergence and then use this to construct a second-order method.

Let us return to the fixed-point iterations  $x^{(k+1)} = g(x^{(k)})$  and revisit the **Taylor expansion** (5) of the **fixed-point map**  $g$ . Continuing the expansion to **second order** we notice for the error

$$\begin{aligned} x^{(k+1)} - x_* &= g(x^{(k)}) - x_* \\ &= g'(x_*) (x^{(k)} - x_*) + \frac{1}{2} g''(x_*) (x^{(k)} - x_*)^2 + O(|x^{(k)} - x_*|^3). \end{aligned} \quad (9)$$

**Assume** now that  $g'(x_*) = 0$ , such that

$$x^{(k+1)} - x_* = \frac{1}{2} g''(x_*) (x^{(k)} - x_*)^2 + O(|x^{(k)} - x_*|^3).$$

By neglecting the small terms and rearranging we observe

$$\frac{x^{(k+1)} - x_*}{(x^{(k)} - x_*)^2} \simeq \frac{1}{2} g''(x_*)$$

when  $x^{(k)}$  is close to  $x_*$ . Comparing with the condition of order- $q$  convergence, i.e. that the limit

$$\lim_{k \rightarrow \infty} \frac{|x^{(k+1)} - x_*|}{|x^{(k)} - x_*|^q} = C$$

is a constant, we thus would **expect such a fixed-point method** with  $g'(x_*) = 0$  to **give quadratic convergence**. More generally if  $g'(x_*) = g''(x_*) = \dots = g^{(q-1)}(x_*) = 0$  and  $g^{(q)}(x_*) \neq 0$  we obtain a method of order  $q$ . We summarise in a theorem:

#### Theorem 3

Let  $g : [a, b] \rightarrow \mathbb{R}$  be a  $p$  times continuously differentiable fixed-point map with fixed point  $x_* \in [a, b]$ . If

$$g'(x_*) = g''(x_*) = \dots = g^{(q-1)}(x_*) = 0 \quad \text{and} \quad g^{(q)}(x_*) \neq 0$$



then there exists a  $\delta > 0$  such that for all starting points  $x^{(0)} \in [x_* - \delta, x_* + \delta]$  the following holds:

- The **fixed-point iterations**  $x^{(k+1)} = g(x^{(k)})$  converge to  $x_*$  with **convergence order**  $q$ .
- The convergence rate is

$$\lim_{k \rightarrow \infty} \frac{|x^{(k+1)} - x_*|}{|x^{(k)} - x_*|^q} = \frac{1}{q!} |g^{(q)}(x_*)|$$

Recall the residual-error relationship (6)

$$|x^{(k)} - x_*| = \frac{1}{|1 - g'(\xi^{(k)})|} r^{(k)}.$$

A corollary of our arguments is that for superlinear methods we have that  $g'(x_*) = 0$ , such that for  $x^{(k)}$  close to  $x_*$  (and thus  $\xi^{(k)} \simeq x_*$ ) we have that

$$|x^{(k)} - x_*| \simeq r^{(k)}.$$

As a result for superlinear methods a residual-based stopping criterion becomes extremely reliable.

## Construction of Newton's method $\Leftrightarrow$

Newton's method and its variants are the **most common approaches** to **solve non-linear equations**  $f(x) = 0$ .

To develop these methods assume that the fixed-point is  $x_*$  and we are given a point  $x$ , which is close to  $x_*$ . We consider a Taylor expansion of  $f$  around  $x$ :

$$0 = f(x_*) = f(x) + f'(x)(x_* - x) + O((x_* - x)^2)$$

Where we made the condition  $f(x_*) = 0$  has been made explicit. Now assume  $f'(x) \neq 0$  to rearrange this to

$$0 = \frac{f(x)}{f'(x)} + (x_* - x) + O((x - x_*)^2).$$

If  $x$  is close to  $x_*$ , thus  $x - x_*$  is small, then the last  $O((x - x_*)^2)$  term is even smaller. Neglecting it we can further develop this to an approximation for  $x_*$  as

$$0 \simeq \frac{f(x)}{f'(x)} + x_* - x \quad \implies \quad x_* \simeq x - \frac{f(x)}{f'(x)}.$$

If we denote the RHS by  $g_{\text{Newton}}(x) = x - \frac{f(x)}{f'(x)}$ , then a root of  $f$  is a fixed point of  $g_{\text{Newton}}$  and vice versa. Performing fixed-point iterations on  $g_{\text{Newton}}$  is the idea of Newton's method:

### Algorithm 1: Newton's method (fixed-point formulation)

Given a  $C^1$  function  $f$  and an initial guess  $x^{(0)}$  perform fixed-point iterations

$$x^{(k+1)} = g_{\text{Newton}}(x^{(k)})$$

on the map

$$g_{\text{Newton}}(x) = x - \frac{f(x)}{f'(x)}. \quad (10)$$

## Graphical interpretation $\Rightarrow$

- See the chapter on [Newton's method](#) in the MIT computational thinking class.
- See [chapter 4.3](#) of Driscoll, Brown: *Fundamentals of Numerical Computation*.

## Convergence analysis $\Rightarrow$

Our goal is to apply Theorem 3 in order to obtain both the result that Newton's method converges as well as an understanding of its convergence order. We thus study the derivatives of  $g_{\text{Newton}}$  at the fixed point  $x_*$ . We obtain

$$g'_{\text{Newton}}(x) = 1 - \frac{f'(x)f'(x) - f''(x)f(x)}{(f'(x))^2} = \frac{f''(x)f(x)}{(f'(x))^2}$$

$$g''_{\text{Newton}}(x) = \frac{(f'(x))^3 f''(x) + f(x)(f'(x))^2 f'''(x) - 2f(x)(f''(x))^2 f'(x)}{(f'(x))^4}$$

such that under the assumption that  $f'(x_*) \neq 0$  and  $f''(x_*) \neq 0$  we obtain

$$g'_{\text{Newton}}(x_*) = 0$$

$$g''_{\text{Newton}}(x_*) = \frac{f''(x_*)}{f'(x_*)} \neq 0$$

where we used  $f(x_*) = 0$ . We summarise

### Theorem 4: Convergence of Newton's method

Let  $f$  be a twice differentiable ( $C^2$ ) function and  $x_*$  a root of  $f$ . If  $f'(x_*) \neq 0$  and  $f''(x_*) \neq 0$ , then Newton's method converges quadratically for every  $x^{(0)}$  sufficiently close to  $x_*$ . The rate is

$$\lim_{k \rightarrow \infty} \frac{|x^{(k+1)} - x_*|}{|x^{(k)} - x_*|^2} = \frac{1}{2} \left| \frac{f''(x_*)}{f'(x_*)} \right|$$

Some remarks:

- Theorem 4 only makes a *local* convergence statement, i.e. it requires the initial value  $x^{(0)}$  to be close enough to  $x_*$ .
- If  $f'(x_*) = 0$  we can show that Newton's method is only of first order.

## Implementation ⇄

Since in our construction Newton's method (Algorithm 1) is obtained in form of the fixed-point map  $g_{\text{Newton}}$  the implementation is straightforward by employing the `fixed_point_iterations` function we already implemented above:

```
newton_fp (generic function with 1 method)
1 function newton_fp(f, df, xstart; maxiter=40, tol=1e-6)
2     # f: Function of which we seek the roots
3     # df: Function, which evaluates its derivatives
4     # xstart: Start of the iterations
5     # maxiter: Maximal number of iterations
6     # tol: Convergence tolerance
7
8     # Define the fixed-point function g_Newton using f and df
9     g_Newton(x) = x - f(x) / df(x)
10
11     # Solve for its fixed point:
12     fixed_point_iterations(g_Newton, xstart; tol, maxiter)
13 end
```

In this setting where  $g_{\text{Newton}}$  exhibits quadratic convergence, the **residual-based stopping criterion** of `fixed_point_iterations` is actually **extremely reliable**, see the discussion after Theorem 3.

More conventionally one "inlines" the function  $g_{\text{Newton}}$  into the fixed point iterations and expresses the problem as

### Algorithm 2: Newton's method (conventional)

Given a once differentiable function  $f: \mathbb{R} \rightarrow \mathbb{R}$ , a starting value  $x^{(0)}$  and a convergence tolerance  $\epsilon$ , perform for  $k = 1, 2, 3, \dots$ :

1. Compute the residual  $r^{(k)} = -\frac{f(x^{(k)})}{f'(x^{(k)})}$
2. Update  $x^{(k+1)} = x^{(k)} + r^{(k)}$

Loop 1. and 2. until  $|r^{(k)}| < \epsilon$ .

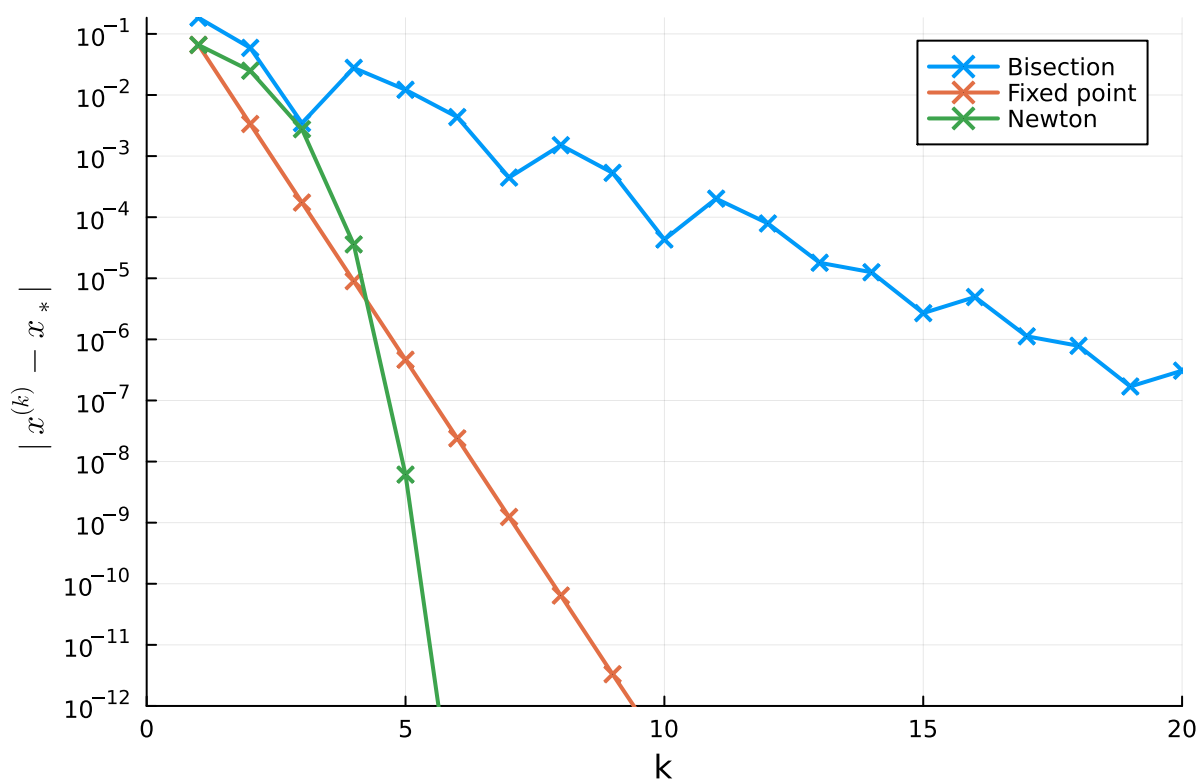
A corresponding implementation of Algorithm 2 is:

```
newton (generic function with 1 method)
1 function newton(f, df, xstart; maxiter=40, tol=1e-6)
2     # f: Function of which we seek the roots
3     # df: Function, which evaluates its derivatives
4     # xstart: Start of the iterations
5     # maxiter: Maximal number of iterations
6     # tol: Convergence tolerance
7
8     history_x = [float(xstart)]
9     history_r = empty(history_x)
10
11     r = Inf      # Dummy to enter the while loop
12     x = xstart  # Initial iterate
13     k = 0
14
15     # Keep running the loop when the residual norm is beyond the tolerance
16     # and we have not yet reached maxiter
17     while norm(r) ≥ tol && k < maxiter
18         k = k + 1
19
20         # Evaluate function, gradient and residual
21         r = - f(x) / df(x)
22
23         # Evaluate next iterate
24         x = x + r
25
26         push!(history_r, r) # Push residual and
27         push!(history_x, x) # next iterate to history
28     end
29
30     (; root=x, n_iter=k, history_x, history_r)
31 end
```

To compare Algorithm 2 to Algorithm 1 note that steps 1 and 2 jointly apply the function  $g_{\text{Newton}}$  and that the residual is defined as

$$r^{(k)} = g_{\text{Newton}}(x^{(k)}) - x^{(k)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} - x^{(k)} = -\frac{f(x^{(k)})}{f'(x^{(k)})}$$

To see how this method performs we compare against bisection and the plain fixed-point iterations in  $g_{\log}$  we saw earlier.



```

1 let
2   # First get an almost exact root
3   reference = bisection_method(f, -0.5, 0.5; tol=1e-14)
4
5   p = plot(yaxis=:log, xlims=(0, 20), ylims=(1e-12, Inf),
6           xlabel="k", ylabel=L"|x^{(k)} - x_*|")
7
8   # Run bisection
9   result = bisection_method(f, -0.5, 0.5; tol=1e-12)
10  errors = [abs(xn - reference.root) for xn in result.history_x]
11  plot!(p, errors, label="Bisection", mark=:x, lw=2)
12
13  # Run fixed-point on glog
14  result = fixed_point_iterations(glog, 0.0; tol=1e-12)
15  errors = [abs(xn - reference.root) for xn in result.history_x]
16  plot!(p, errors, label="Fixed point", mark=:x, lw=2)
17
18  # For Newton we need the derivative of f.
19  # An easy way to obtain this derivative is to use algorithmic differentiation:
20  df(x) = ForwardDiff.derivative(f, x)
21
22  # With this we run Newton
23  result = newton(f, df, 0.0; tol=1e-12)
24  errors = [abs(xn - reference.root) for xn in result.history_x]
25  plot!(p, errors, label="Newton", mark=:x, lw=2)
26  yticks!(p, 10.0 .^ (-12:-1))
27

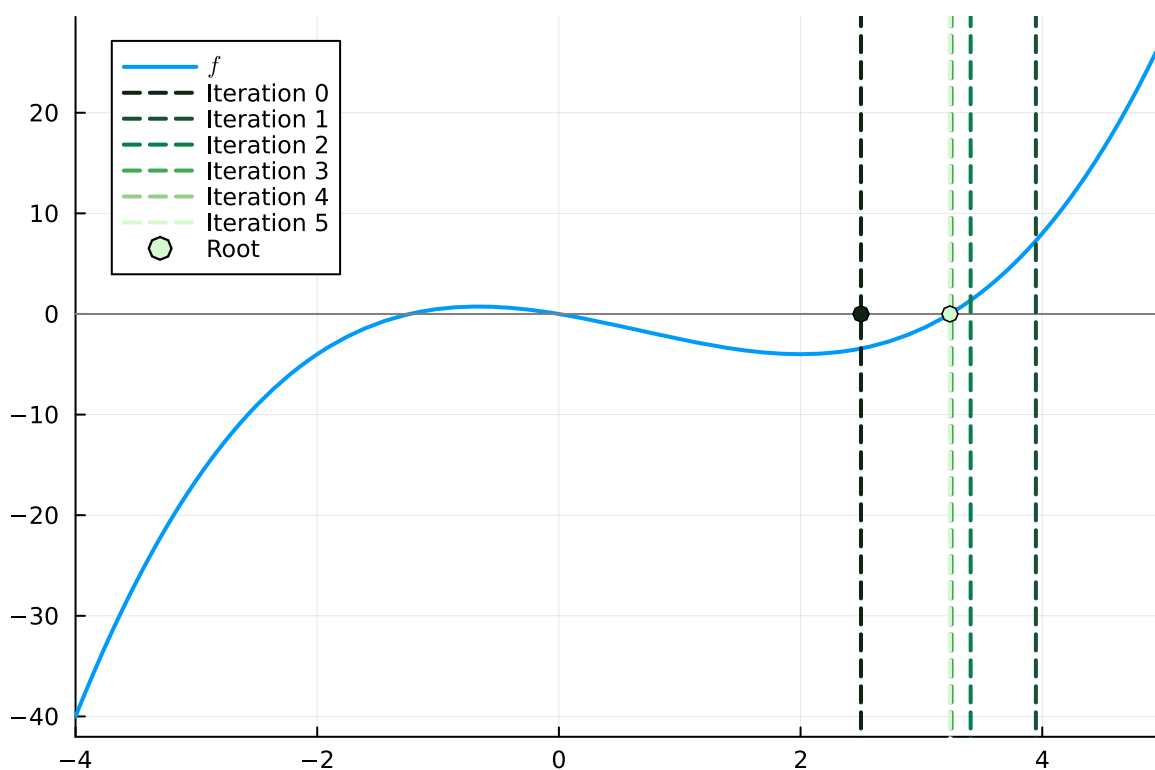
```

```
28 p
29 end
```

On the log-scale of the plot the quadratic convergence behaviour of Newton's method is clearly visible.

Let us investigate a little the stability of this algorithm, especially with respect to the requirement to choose a sufficiently close initial guess:

•  $x_0 =$



Finally, let us investigate what quadratic convergence means in terms of our error estimate, the residuals of the Newton fixed-point map. Since Newton converges fast and thus very quickly maxes out the roughly 16 digits of precision in standard `Float64` numbers, we employ Julia's arbitrary precision `BigFloat` type to see more closely what is going on:

```

1 let
2   # We want to solve  $e^x * x = 2$ , which has a solution near 1.
3   f(x) = x*exp(x) - 2
4   df(x) = (x+1) * exp(x)
5
6   xstart = BigFloat(1.0)
7   result = newton(f, df, xstart; tol=1e-60)
8
9   for (i, r) in enumerate(result.history_r)
10      @printf "%3i %e\n" i r
11   end
12 end

```

```

1 -1.321206e-01
2 -1.509607e-02
3 -1.778470e-04
4 -2.435520e-08
5 -4.566801e-16
6 -1.605657e-31
7 -1.984881e-62

```

We observe that from the starting point only **7** iterations are required to get the result accurate to 61 digits.

## Observations

- For quadratic convergence the error roughly squares in each iteration
- Newton only converges well if the initial guess is chosen sufficiently close to the fixed point.
- Unlike the bisection algorithm the convergence behaviour of Newton is thus sometimes less reliable. In particular if  $f$  has **multiple roots** it is **not guaranteed**, that **Newton converges to the closest root**. A good graphical representation of this phenomenon are Newton fractals.

## Lessons to learn from fixed-point iterations ⇔

In this chapter we discussed fixed point iterations as well as Newton's method (Algorithm 2) as two examples for iterative algorithms.

### General form of iterative algorithms

More generally an iterative algorithm has the form:



1. **Initialisation:** Choose a starting point  $\mathbf{x}^{(0)} = \mathbf{x}_{\text{start}}$ .
2. **Iteration:** For  $k = 1, 2, 3, \dots$  we perform the same iterative procedure, advancing  $\mathbf{x}^{(k-1)}$  into  $\mathbf{x}^{(k)}$ .
3. **Check for convergence:** Once  $\mathbf{x}^{(k)}$  is similar enough to  $\mathbf{x}^{(k-1)}$  we consider the iteration converged and exit the iterations.

Writing the second step more mathematically we can consider it as the application of a function  $\mathbf{g}$ , i.e.  $\mathbf{x}^{(k)} = \mathbf{g}(\mathbf{x}^{(k-1)})$ . In this formulation step 3 thus does nothing else than checking whether the iterates  $\mathbf{x}^{(k)}$  no longer change. Or, put in other words, if we have found a **fixed point** of  $\mathbf{g}$ .

### Observation: Iterative algorithms are fixed-point problems

By identifying the iteration step  $\mathbf{x}^{(k)} = \mathbf{g}(\mathbf{x}^{(k-1)})$  of any iterative algorithm with a function  $\mathbf{g}$  we can view this algorithm as a fixed-point problem, where at convergence a fixed-point of  $\mathbf{g}$  is found.

As a result **any technique we discussed** for understanding *when fixed-point iterations converge* and at *which convergence rate* can be used **in general** to analyse the convergence of *any iterative procedure*.

We will consider this aspect further, for example in Iterative methods for linear systems.

## Optional: Secant method ⇨

See chapter 4.4 of Driscoll, Brown: *Fundamentals of Numerical Computation*.

## Non-linear equation systems ⇨

Many applications are characterised by more than one degree of freedom and more than a single equation to satisfy. Here, we will generalise our discussion and consider a system of equations

$$\begin{cases} f_1(x_1, \dots, x_n) = 0 \\ f_2(x_1, \dots, x_n) = 0 \\ \vdots \\ f_n(x_1, \dots, x_n) = 0 \end{cases}$$

Introducing the compact vector notation

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad \mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(x_1, \dots, x_n) \\ f_2(x_1, \dots, x_n) \\ \vdots \\ f_n(x_1, \dots, x_n) \end{pmatrix},$$

we can define the multi-dimensional version of the root-finding problem:

### Definition: Multidimensional root-finding problem

Given a continuous vector-valued function  $\mathbf{f}: \mathbb{R}^n \rightarrow \mathbb{R}^n$  find a vector  $\mathbf{x}_* \in \mathbb{R}^n$  such that  $\mathbf{f}(\mathbf{x}_*) = \mathbf{0}$ .

Solving such nonlinear multi-dimensional equation systems is much more involved. Even establishing basic mathematical properties, such as the existence or uniqueness of solutions is typically quite difficult, let alone solving such equation systems analytically.

### Running example: Definition

As the running example in this section we will consider the problem  $\mathbf{f}(\mathbf{x}) = \mathbf{0}$  with  $\mathbf{f}: \mathbb{R}^3 \rightarrow \mathbb{R}^3$  given by

$$\begin{cases} f_1(x_1, x_2, x_3) = -x_1 \cos(x_2) - 1 \\ f_2(x_1, x_2, x_3) = x_1 x_2 + x_3 \\ f_3(x_1, x_2, x_3) = e^{-x_3} \sin(x_1 + x_2) + x_1^2 - x_2^2 \end{cases}.$$

This example very much illustrates the previous point of the increased complexity: Even guessing an approximate solution is not obvious, so we proceed to develop a numerical technique. Taking inspiration from Newton's method in 1D we proceed to solve such equation systems **by linearisation**. That is to say we start by developing  $\mathbf{f}$  to first order around some initial point  $\mathbf{x}$ , which we assume to be close enough to the solution  $\mathbf{x}_*$

$$\mathbf{0} = \mathbf{f}(\mathbf{x}_*) = \mathbf{f}(\mathbf{x}) + \mathbf{J}_{\mathbf{f}}(\mathbf{x}) (\mathbf{x}_* - \mathbf{x}) + O(\|\mathbf{x}_* - \mathbf{x}\|^2). \quad (11)$$

In this the **Jacobian matrix**  $\mathbf{J}_{\mathbf{f}}(\mathbf{x}) \in \mathbb{R}^{n \times n}$  is the collection of all partial derivatives of  $\mathbf{f}$ , i.e.

$$\mathbf{J}_f = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}.$$

See also the discussion on multi-dimensional Talyor approximations in [Revision and preliminaries](#).

The Jacobian very much plays the role of a generalised derivative of a multidimensional function  $\mathbf{f}$ . Also not that (just like any derivative) it is a function of the independent variable  $\mathbf{x}$ .

### Running example: Computing the Jacobian

For the running example identified above we can compute the Jacobian as

$$\mathbf{J}_f(\mathbf{x}) = \begin{pmatrix} -\cos(x_2) & x_1 \sin(x_2) & 0 \\ x_2 & x_1 & 1 \\ e^{-x_3} \cos(x_1 + x_2) + 2x_1 & e^{-x_3} \cos(x_1 + x_2) - 2x_2 & -e^{-x_3} \sin(x_1 + x_2) \end{pmatrix}$$

In expansion (11) the terms  $\mathbf{f}(\mathbf{x}) + \mathbf{J}_f(\mathbf{x}) (\mathbf{x}_* - \mathbf{x})$  represent the linear part of  $\mathbf{f}$  around  $\mathbf{x}_*$ .

Assuming that these dominate, i.e. that the remaining term  $O(\|\mathbf{x}_* - \mathbf{x}\|^2)$  is indeed small and can be neglected, we obtain:

$$\mathbf{0} \simeq \mathbf{f}(\mathbf{x}) + \mathbf{J}_f(\mathbf{x}) (\mathbf{x}_* - \mathbf{x})$$

In the same spirit as in the 1D Newton case we want to employ this relation in an iterative scheme, where in iteration  $k$  we have  $\mathbf{x}^{(k)}$  and want to compute an improved iterate  $\mathbf{x}^{(k+1)}$ . Inserting  $\mathbf{x}^{(k)}$  as  $\mathbf{x}$  and  $\mathbf{x}^{(k+1)}$  as  $\mathbf{x}_*$  as in the 1D case, we obtain

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

or

$$\left( \mathbf{J}_f(\mathbf{x}^{(k)}) \right) (\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = -\mathbf{f}(\mathbf{x}^{(k)}). \quad (12)$$

Assuming that  $\det \mathbf{J}_f(\mathbf{x}^{(k)}) \neq 0$ , i.e. that the Jacobian is non-singular, this linear system can be solved and thus  $\mathbf{x}^{(k+1)}$  computed.

Note, that in accordance with the 1D case the **residual in this multi-dimensional version** is  $\mathbf{r}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ , i.e. exactly the solution to the linear system in (12). Similar to the 1D case we will thus employ the stopping criterion  $\|\mathbf{r}^{(k)}\| < \epsilon$ , where  $\|\mathbf{x}\|$  denotes the Euclidean norm  $\|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$ . All combined we obtain the algorithm

### Algorithm 3: Multidimensional Newton's method

Given a once differentiable function  $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , a starting value  $\mathbf{x}^{(0)}$  and a convergence tolerance  $\epsilon$ , perform for  $k = 1, 2, 3, \dots$ :

1. Compute the right-hand side  $\mathbf{y}^{(k)} = \mathbf{f}(\mathbf{x}^{(k)})$  and Jacobian  $\mathbf{A}^{(k)} = \mathbf{J}_{\mathbf{f}}(\mathbf{x}^{(k)})$ .
2. **Newton step:** Solve the linear system  $\mathbf{A}^{(k)} \mathbf{r}^{(k)} = -\mathbf{y}^{(k)}$  for  $\mathbf{r}^{(k)}$ .
3. Update  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{r}^{(k)}$

Loop 1. to 3. until  $\|\mathbf{r}^{(k)}\| < \epsilon$ .

An implementation of this algorithm is given below:

```
newtonsys (generic function with 1 method)
1 function newtonsys(f, jac, xstart; maxiter=40, tol=1e-8)
2     history_x = [float(xstart)]
3     history_r = empty(history_x)
4
5     r = Inf      # Dummy to enter the while loop
6     x = xstart  # Initial iterate
7     k = 0
8     while norm(r) ≥ tol && k < maxiter
9         k = k + 1
10
11         y = f(x)      # Function value
12         A = jac(x)    # Jacobian
13         r = -(A \ y)  # Newton step
14         x = x + r      # Form next iterate
15
16         push!(history_r, r) # Push newton step and
17         push!(history_x, x) # next iterate to history
18     end
19
20     (; root=x, n_iter=k, history_x, history_r)
21 end
```

Note that the linear system  $\mathbf{A}^{(k)} \mathbf{r}^{(k)} = -\mathbf{y}^{(k)}$  is solved in Julia using the backslash operator  $\backslash$ , which employs a numerically more stable algorithm than explicitly computing the inverse  $\text{inv}(\mathbf{A})$

and then applying this to  $y$ . We will discuss these methods in [Direct methods for linear systems](#).

► **Remark: Connection to conventional 1D Newton algorithm (Algorithm 2)**

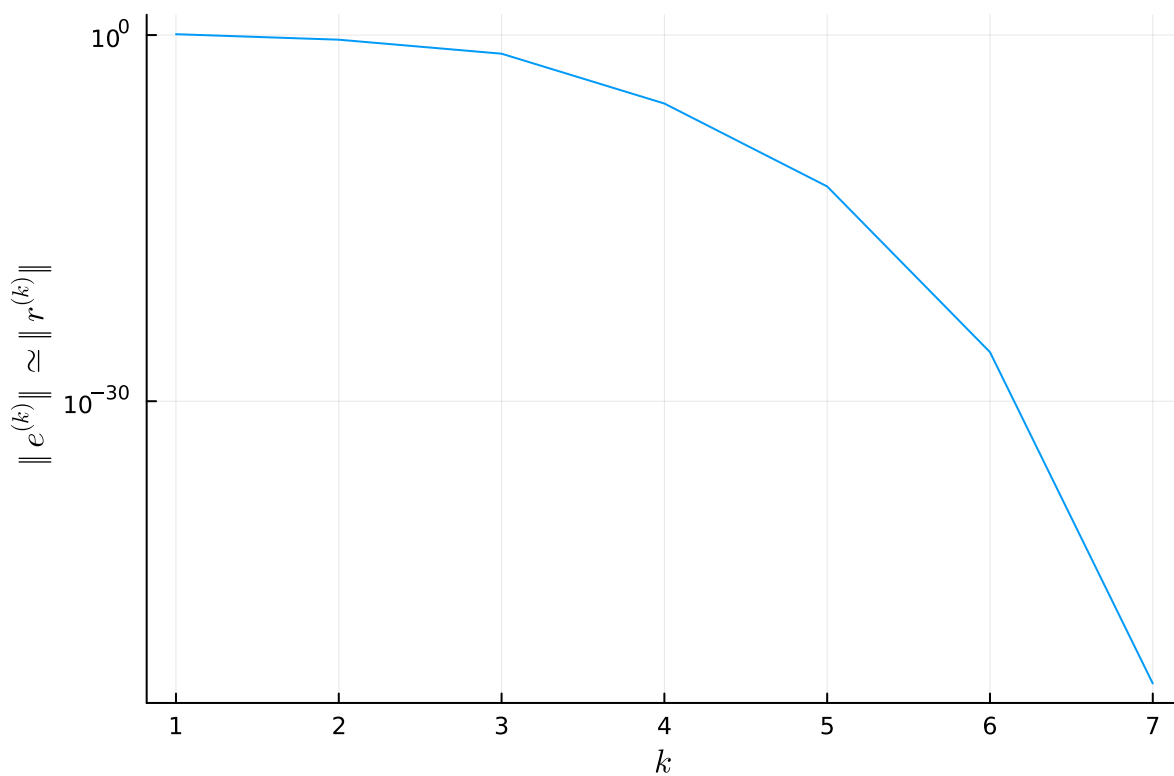
Let's apply `newtonsys` (Algorithm 2) to our running example. First we implement the functions computing  $\mathbf{f}(\mathbf{x})$  and  $\mathbf{J}_f(\mathbf{x})$  for a given  $\mathbf{x}$ .

```
1 begin
2     func(x) = [
3         -x[1] * cos(x[2]) - 1,
4         x[1] * x[2] + x[3],
5         exp(-x[3]) * sin(x[1] + x[2]) + x[1]^2 - x[2]^2
6     ]
7     jac_func(x) = [
8         -cos(x[2])      x[1]*sin(x[2])      0;
9         x[2]            x[1]                1;
10        exp(-x[3])*cos(x[1]+x[2]) + 2x[1]  exp(-x[3])*cos(x[1]+x[2]) - 2x[2] exp(-
11        x[3])*sin(x[1]+x[2])
12    ]
13 end;
```

Since we want to estimate the convergence order we again run the Newton solver using arbitrary precision floating-point numbers by using Julia's `BigFloat` number type:

```
1 res = newtonsys(func, jac_func, BigFloat.([1.5, -1.5, 5]), tol=1e-50);
```

Plotting the residual norm (our estimate of the error) in a log-plot gives a strong indication this is again quadratic convergence:



```
1 plot(norm.(res.history_r); yaxis=:log, label="", xlabel=L"k", ylabel=L"\|e^{(k)}\| \simeq \|r^{(k)}\|")
```

Using the residual norms stored in the Newton result, we can now also look at the ratios

$$\frac{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|}{\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|^q} = \frac{\|\mathbf{r}^{(k)}\|}{\|\mathbf{r}^{(k-1)}\|^q}$$

of two consecutive increments. Recall that for a  $q$ -th order convergence these should converge to a constant.

```

1 let
2   for q in (1, 2, 3)
3     println("# Checking order q=$q")
4     for k in 2:length(res.history_r)
5       ratio = norm(res.history_r[k]) / norm(res.history_r[k-1])^q
6       @printf "%i %.5f\n" k ratio
7     end
8     println()
9   end
10 end

```

```

# Checking order q=1
2 0.35411
3 0.07138
4 0.00008
5 0.00000
6 0.00000
7 0.00000

# Checking order q=2
2 0.30117
3 0.17146
4 0.00278
5 0.06559
6 0.06755
7 0.06755

# Checking order q=3
2 0.25615
3 0.41182
4 0.09348
5 26726.63724
6 171033902015.23834
7 6410486721954533487072506.39898

```

As can be see the most constant is the sequence corresponding to  $q = 2$ , such that we conclude that the method converges quadratically.

## Numerical analysis

1. Introduction
2. The Julia programming language
3. Revision and preliminaries
4. Root finding and fixed-point problems
5. Interpolation
6. Direct methods for linear systems
7. Iterative methods for linear systems
8. Eigenvalue problems

- 9. Numerical integration
- 10. Numerical differentiation
- 11. Initial value problems
- 12. Boundary value problems