Click here to view the PDF version.

```julia
begin
    using Plots
    using PlutoUI
    using PlutoTeachingTools
    using LaTeXStrings
    using Printf
    using ForwardDiff
    using LinearAlgebra
    using HypertextLiteral: @htl, @htl_str
end
```

Table of Contents

Root finding and fixed-point problems

We saw in the introduction that problems where one wants to find a **fixed point** or a **root** (**zero**) of a function arise naturally in scientific questions. Moreover if the underlying function is more complicated than just a simple polynomial, it becomes **quickly challenging to find such a fixed point** using pen and paper. In this chapter we develop numerical methods for solving such problems.

We start by defining these problems mathematically. In the past you certainly met **root finding problems** where one seeks to find a scalar x such that a function $f : \mathbb{R} \to \mathbb{R}$ is zero, i.e. $f(x) = 0$.

The vector-valued analogue to this problem is that one has a family of functions $f_1, f_2, \ldots, f_n$ each depending on multiple variables $x_1, x_2, \ldots, x_n$ and one seeks to simultaniously satisfy the system of equations

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

Introducing the compact vector notation

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

we can define the vector version of the root-finding problem:

Definition: Rootfinding problem

Given a continuous vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^n$ find a vector $\mathbf{x}_* \in \mathbb{R}^n$ such that $f(\mathbf{x}_*) = \mathbf{0} \in \mathbb{R}^n$. Such a value $\mathbf{x}_*$ is called a **root** of f.

Example: Intersecting circle and parabola

Consider the problem of the plot below, where we seek to find the marked intersection points of a circle and a parabola. These intersection points need to satisfy both the equation for a parabola ($x_2 = 2x_1^2$) and the equation for the circle with radius 1 (i.e. $x_1^2 + x_2^2 = 1$). Rearringing the equations to

$$\begin{aligned} 0 &= 1 - x_1^2 - x_2^2 \\ 0 &= 2x_1^2 - x_2 \end{aligned}$$

thus defines the root finding problem $f(\mathbf{x}) = \mathbf{0}$ with $f : \mathbb{R}^2 \to \mathbb{R}^2$ with

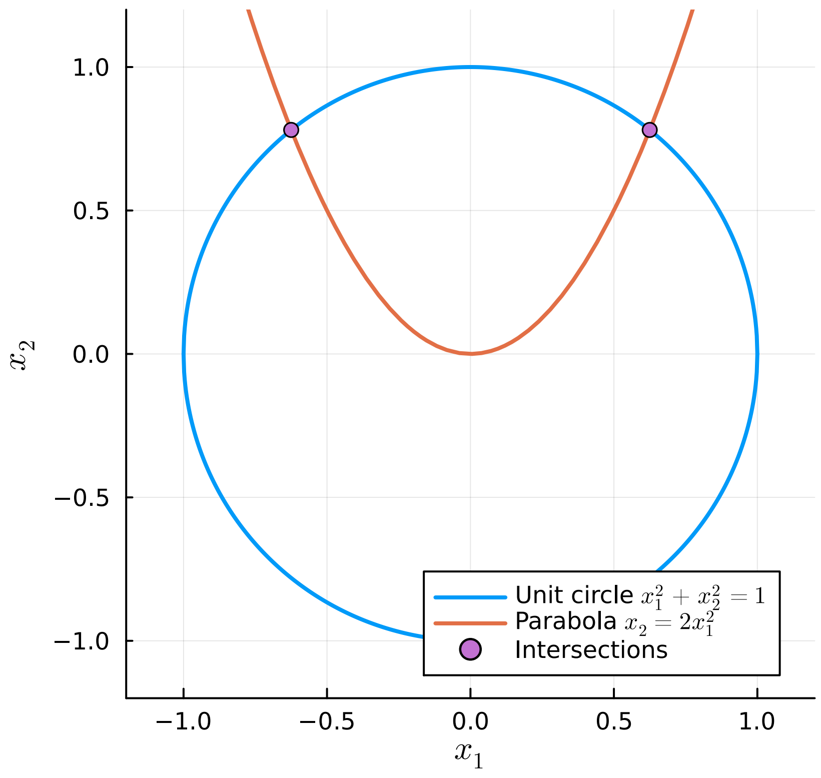
$$\begin{cases} f_1(x_1, x_2) = 1 - x_1^2 - x_2^2 \\ f_2(x_1, x_2) = 2x_1^2 - x_2 \end{cases}.$$

In this example we can still find the solution analytically. E.g. insert the parabola equation $x_2 = 2x_1^2$ into the circle equation $x_1^2 + x_2^2 = 1$ to yield $x_2^2 + \frac{1}{2}x_2 - 1 = 0$ which can be solved for x_2 to yield

$$x_{2,*} = \frac{\sqrt{17} - 1}{4} \approx 0.781 \qquad x_{1,*} = \sqrt{\frac{1}{2} x_{2,*}} \approx 0.624$$

Not the nicest values to compute by hand ...

```julia
begin
    x2_ast = (sqrt(17)-1)/4
    x1_ast = sqrt( x2_ast/2 )
end;
```



An equivalent and sometimes more intuitive way to think about solving equations $f(\mathbf{x}) = \mathbf{0}$ is to recast them as a fixed point-problem:

Definition: Fixed-point problem

Given a continous function $g : \mathbb{R}^n \to \mathbb{R}^n$ a point $\mathbf{x}_*$, such that $g(\mathbf{x}_*) = \mathbf{x}_*$, 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 $f(\mathbf{x}) = g(\mathbf{x}) - \mathbf{x}$.

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

Example: Intersecting circle and parabola (continued)

We continue our example of a circle intersecting a parabola. Recall that the two equations to satisfy are $x_2 = 2x_1^2$ (parabola) and $x_1^2 + x_2^2 = 1$ (circle). We restrict ourselves to the upper-right quadrant, i.e. we will assume $x_1 \geq 0$ and $x_2 \geq 0$.

- Rearranging the circle equation to $x_1 = \sqrt{1 - x_2^2}$ thus leads to the fixed-point problem

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \mathbf{x} = g_D(\mathbf{x}) \qquad \text{with} \qquad g_D(\mathbf{x}) = \begin{pmatrix} \sqrt{1 - x_2^2} \\ 2x_1^2 \end{pmatrix}.$$

- Rearranging instead the parabola equation to $x_1 = \sqrt{\frac{x_2}{2}}$ and the circle equation to $x_2 = \sqrt{1 - x_1^2}$ yields the problem

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \mathbf{x} = g_C(\mathbf{x}) \qquad \text{with} \qquad g_C(\mathbf{x}) = \begin{pmatrix} \sqrt{\frac{x_2}{2}} \\ \sqrt{1 - x_1^2} \end{pmatrix}.$$

Exercise

Verify that the fixed-point problems $\mathbf{x} = g_D(\mathbf{x})$ and $\mathbf{x} = g_C(\mathbf{x})$ are equivalent to the root-finding problem $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

Fixed-point iterations

In the introduction when discussing the diode model, we already mentioned fixed-point iterations as one way of solving the fixed-point problem.

Algorihm 1: Fixed-point iteration

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

$$\mathbf{x}^{(k+1)} = g(\mathbf{x}^{(k)}) \qquad \text{for } k = 1, 2, \ldots.$$

until a **stopping criterion** is reached, see text below.

If $\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}_*$ for the sequence $\mathbf{x}^{(k)}$ generated by Algorithm 1, then $g(\mathbf{x}_*) = \mathbf{x}_*$, i.e. $\mathbf{x}_*$ is a fixed point of g. A typical stopping criterion is to check if the vector norm between subsequent iterates, i.e. $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|$ drops below a certain tolerance. Here $\|x\|$ denotes the Euclidean norm $\|x\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$. We will discuss in the Convergence analysis section why this is a good choice.

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

$$f(\mathbf{x}_*) = 0 \qquad \Longleftrightarrow \qquad \mathbf{x}_* = g(\mathbf{x}_*).$$

On g we then apply fixed-point iteration.

We saw one example how to achieve this rewriting in the discussion of the "Intersecting circle and parabola" example in the section above, where we first defined a root-finding problem and then two equivalent fixed-point problems for the same task.

We already noted in the introduction that these **fixed-point iterations do not always converge**.

Let us investigate this closer for the circle intersection problem of the section above. We noted that the problem can be formulated as two fixed-point problems $\mathbf{x} = g_D(\mathbf{x})$ and $\mathbf{x} = g_C(\mathbf{x})$ involving the functions

$$g_D(\mathbf{x}) = \begin{pmatrix} \sqrt{1 - x_2^2} \\ 2x_1^2 \end{pmatrix} \qquad g_C(\mathbf{x}) = \begin{pmatrix} \sqrt{\frac{x_2}{2}} \\ \sqrt{1 - x_1^2} \end{pmatrix}$$

which we code up as follows:

```
gD (generic function with 1 method)
```

```julia
gD(x) = [sqrt(1 - x[2]^2);
                  2x[1]^2]
```

```
gC (generic function with 1 method)
```

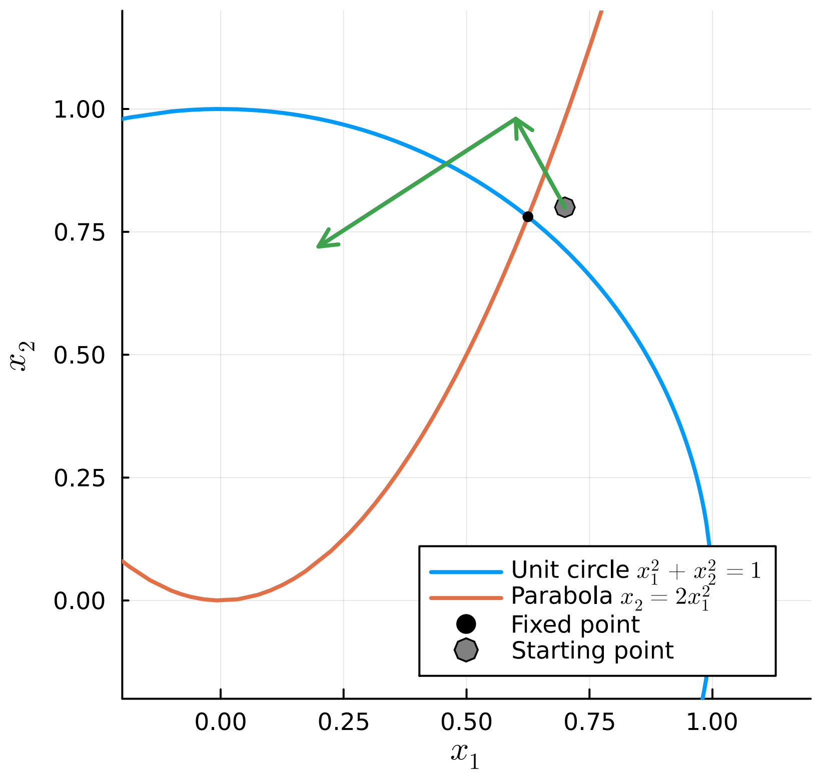
```julia
gC(x) = [     sqrt(x[2]/2);
          sqrt(1 - x[1]^2)]
```

They take as input and provide as output a two-dimensional vector.

We visualise the iterations for `gD` and `gC` by visualising the sequence of points (x_1, x_2) they generate. In the graphics below both the selection of points, the starting point as well as the number of iterations can be selected.

- Fixed-point function `gfix = ` gD
- Starting point `xinit = ` [0.7, 0.8]

- Number of iterations `n_iter = ` 2



We notice:

- While g_C converges (i.e. the arrows get closer and closer to the fixed-point shown in black), g_D diverges.
- While the sequence of points in both cases "circles around" the fixed point. For the converging function g_C the circle spirals inwards, while for g_D the circle spirals outwards.
- Intuitively the fixed-point method therefore converges if it makes the distance between the iteration points and the fixed-point smaller.

We will make this observation more precise in the section on Convergence analysis.

For other ways of visualising fixed-point iterations, see chapter 4.2 of Driscoll, Brown: Fundamentals of Numerical Computation.

Convergence analysis

Scalar functions

To understand the aforementioned behaviour mathematically, we first stay within the setting of a **scalar fixed-point problem**. That is we consider the setting finding a fixed-point $x_* = g(x_*)$ for a differentiable function $g : \mathbb{R} \to \mathbb{R}$.

To understand convergence more rigorously we study the beviour 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. The hope is this enables to find a condition for g that tells us when the fixed point iterations converge (i.e. the error goes to 0 as $k \to \infty$) or not (the error stays non-zeroas $k \to \infty$).

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

$$g(x^{(k)}) = g(x_*) + g'(x_*)(x^{(k)} - x_*) + O((x - x_*)^2) \tag{3}$$

where the Big O notation $O((x - x_*)^2)$ indicates that there are higher-order terms, which we don't show explicitly, but their order is at least $(x - x_*)^2$. See also the discussion in Revision and preliminaries on Taylor approximations and Big O notation.

Sometimes being fully precise in the big O notation will be too distracting. In this case we will use a generic "$O(\text{small})$" to remind ourselves that there are additional terms and we will specify in the surrounding text what this term stands for.

Using (3), the fact that $g(x_*) = x_*$, and the key fixed-point iterations equation, $x^{(k+1)} = g(x^{(k)})$, we obtain

$$\begin{aligned} e^{(k+1)} &= x^{(k+1)} - x_* = g(x^{(k)}) - x_* \\ &\overset{(3)}{=} x_* + g'(x_*)\underbrace{(x^{(k)} - x_*)}_{=e^{(k)}} + O((e^{(k)})^2) - x_* \\ &= g'(x_*)\, e^{(k)} + O((e^{(k)})^2). \end{aligned}$$

Taking moduli on both sides:

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

A related relation also holds between the error in the k-th and $(k-1)$-st iteration, just by subtracting 1 in the iteration count on both sides, i.e.

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

Combining both expressions we obtain

$$|e^{(k+1)}| = |g'(x_*)|^2\, |e^{(k-1)}| + O(\text{small}),$$

where $O(\text{small})$ is a term that is at least quadratic in $|e^{(k)}|$ and at least quadratic in $|e^{(k-1)}|$.

Continuing in a recursive fashion:

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

where the $O(\text{small})$ is at least quadratic in all errors $|e^{(k)}|$ to $|e^{(0)}|$.

From this we conclude:

- If $|g'(x_*)| > 1$, then $|g'(x_*)|^k$ grows to infinity as the number of iteration k grows, therefore the error $|e^{(k+1)}|$ has to grow: **the fixed-point iterations diverge.**
- Furthermore if $|g'(x_*)| < 1$ and if the terms hidden in $O(\text{small})$ can be neglected, then $|e^{(k+1)}|$ is getting smaller and smaller as the number of iterations k increases: **the fixed-point iterations converge**.
- Now we ask under which conditions the terms in $O(\text{small})$ can be neglected. This is indeed the case if $|e^{(0)}|$ is sufficiently small, i.e. **if our starting point $x^{(0)}$ is sufficiently close to the fixed point x_***.
- To see this, assume $|g'(x_*)| < 1$. As a result $|e^{(1)}| < |g'(x_*)|\, |e^{(0)}| + O(|e^{(0)}|^2)$ by (∗) for $k = 1$. Therefore if $|e^{(0)}|$ is sufficiently small, then $|e^{(0)}|^2 = |x_* - x^{(0)}|^2$ is small compared to $|g'(x_*)|\, |e^{(0)}|$, meaning that the terms hidden in $O(|e^{(0)}|^2)$ can be neglected. As a result $|e^{(1)}|^2$ is even smaller than $|e^{(0)}|^2$, such that also the terms in $O(|e^{(1)}|^2)$ can be neglected and so forth. We conclude that all terms at least quadratic in the error term $|e^{(k)}|$ to $|e^{(0)}|$ can be neglected, i.e. that the entire set of terms $O(\text{small})$ is neglibile.
- Note that for $|g'(x_*)| = 1$ our theory does not allow us to conclude neither convergence, nor divergence.

We summarise in a Theorem:

Theorem 0 (scalar version of Theorem 1)

Let $g : \mathbb{R} \to \mathbb{R}$ be a once differentiable function and $x_* \in \mathbb{R}$ 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. $\lim_{k\to\infty} x^{(k)} = x_*$

We will generalise this theorem to the vector case in the following secition.

Higher dimensions

We now consider the generalisation of the above argument to the multi-dimensional setting, i.e. finding a fixed-point $\mathbf{x}_* = g(\mathbf{x}_*) \in \mathbb{R}^n$ of a function $g : \mathbb{R} \to \mathbb{R}$. To make a similar argument to the scalar case, we need to consider again the Talyor expansion of $g(\mathbf{x}^{(k)}) = g(\mathbf{x}_* + \mathbf{e}^{(k)})$ around $\mathbf{x}_*$, where as before $\mathbf{e}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}_*$.

First recall that for a vector-valued function the Taylor expansion to first order reads

$$g(\mathbf{y}) = g(\mathbf{x}) + \mathbf{J}_g(\mathbf{x})\,(\mathbf{y} - \mathbf{x}) + O(\text{small}) \tag{4}$$

In this we used "$O(\text{small})$" to denote that we suppressed higher-order terms and the **Jacobian matrix** $\mathbf{J}_g(\mathbf{x}) \in \mathbb{R}^{n\times n}$ is the collection of all partial derivatives of g *evaluated at* $\mathbf{x}$, i.e.

$$\mathbf{J}_g = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial g_n}{\partial x_1} & \frac{\partial g_n}{\partial x_2} & \cdots & \frac{\partial g_n}{\partial x_n} \end{pmatrix}.$$

See also the discussion on multi-dimensional Talyor approximations in Revision and preliminaries. Note that the Jacobian (just like any derivative) is a function of an independent variable (here $\mathbf{x}$).

Since the Jacobian very much plays the role of a generalised derivative of a multidimensional function g, we will sometimes also borrow the notation g' from the scalar case to denote the Jacobian, i.e. we sometimes write $g'(\mathbf{x}) = \mathbf{J}_g(\mathbf{x})$.

Intersecting circle and parabola (continued)

As an example we compute the Jacobian of the two fixed-point maps that arose in the study of the curve intersection problem between circle and parabola, which we considered above, namely

$$g_D(\mathbf{x}) = \begin{pmatrix} \sqrt{1-x_2^2} \\ 2x_1^2 \end{pmatrix} \qquad g_C(\mathbf{x}) = \begin{pmatrix} \sqrt{\frac{x_2}{2}} \\ \sqrt{1-x_1^2} \end{pmatrix}.$$

For the first we find

$$\mathbf{J}_{g_D}(\mathbf{x}) = \begin{pmatrix} \frac{\partial}{\partial x_1}\sqrt{1-x_2^2} & \frac{\partial}{\partial x_2}\sqrt{1-x_2^2} \\ \frac{\partial}{\partial x_1}2x_1^2 & \frac{\partial}{\partial x_2}2x_1^2 \end{pmatrix} = \begin{pmatrix} 0 & \frac{-x_2}{\sqrt{1-x_2^2}} \\ 4x_1 & 0 \end{pmatrix}$$

and for the second

$$\mathbf{J}_{g_C}(\mathbf{x}) = \begin{pmatrix} \frac{\partial}{\partial x_1}\sqrt{\frac{x_2}{2}} & \frac{\partial}{\partial x_2}\sqrt{\frac{x_2}{2}} \\ \frac{\partial}{\partial x_1}\sqrt{1-x_1^2} & \frac{\partial}{\partial x_2}\sqrt{1-x_1^2} \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{4\sqrt{\frac{x_2}{2}}} \\ \frac{-x_1}{\sqrt{1-x_1^2}} & 0 \end{pmatrix}$$

Now, using the Taylor expansion of (4) with $\mathbf{y} = \mathbf{x}_* + \mathbf{e}^{(k)}$ and $\mathbf{x} = \mathbf{x}_*$ we obtain analogously to the scalar case for the **error in the fixed-point iterations**

$$\begin{aligned} \mathbf{e}^{(k+1)} &= g(\mathbf{x}_* + \mathbf{e}^{(k)}) - \mathbf{x}_* \\ &\overset{(4)}{=} g(\mathbf{x}_*) + \mathbf{J}_g(\mathbf{x}_*)\,\mathbf{e}^{(k)} - \mathbf{x}_* + O(\text{small}) \\ &= \mathbf{J}_g(\mathbf{x}_*)\,\mathbf{e}^{(k)} + O(\text{small}). \end{aligned} \tag{5}$$

i.e. we see the Jacobian indeed play the role of the derivative of $\mathbf{g}$.

The above expression again relates the error in step $k+1$ to the error in step k. Following the analogy from the scalar case we now want to understand how the size of this error varies between iterations. We need one further ingredient, namely the generalisation of the modulus of the gradient $|g'(x_*)|$ to higher dimensions, where notably $\mathbf{J}_g$ is now an $n \times n$ matrix.

We need the

Definition: Matrix norm

Given $\mathbf{M} \in \mathbb{R}^{m\times n}$ a real matrix (not necessarily square), we define the matrix norm of $\mathbf{M}$ as

$$\|\mathbf{M}\| = \max_{\substack{\mathbf{v}\in\mathbb{R}^n \\ \mathbf{v}\neq\mathbf{0}}} \frac{\|\mathbf{M}\mathbf{v}\|}{\|\mathbf{v}\|}.$$

The definition of the matrix norm implies in particular that

$$\|\mathbf{M}\mathbf{v}\| \leq \|\mathbf{M}\| \, \|\mathbf{v}\| \quad \forall \mathbf{v} \in \mathbb{R}^n.$$

We now take vector norms on either side of (5) and make use of this last inequality to obtain to first order

$$\|\mathbf{e}^{(k+1)}\| \leq \|\mathbf{J}_g(\mathbf{x}_*)\| \, \left\|\mathbf{e}^{(k)}\right\| + O(\text{small})$$

Under the assumption that our initial guess $\mathbf{x}^{(0)}$ is sufficiently close to $\mathbf{x}_*$ we can again follow a recursive argument to obtain

$$\|\mathbf{e}^{(k+1)}\| \leq \|\mathbf{J}_g(\mathbf{x}_*)\|^{k+1} \, \|\mathbf{e}^{(0)}\| + O(\text{small})$$

where $O(\text{small})$ is a small term that we do not make more precise for simplicity.

Similar to the one-dimensional case we conclude:

- If $\|\mathbf{J}_g(\mathbf{x}_*)\| < 1$ and the initial guess $\mathbf{x}^{(0)}$ is sufficiently close to the final fixed point $\mathbf{x}_*$, then as $k \to \infty$, i.e. as the iteration progresses, the error norm $\|\mathbf{e}^{(k+1)}\|$ approaches zero: **the fixed-point iterations converge**.

However, in contrast to the one-dimensional setting **we cannot conclude divergence** if $\|\mathbf{J}_g(\mathbf{x}_*)\| > 1$. This is because we only obtain an *inequality* relating $\|\mathbf{e}^{(k+1)}\|$ to $\|\mathbf{e}^{(0)}\|$, whereas in the one-dimensional case this equation was an equality.

The following theorem summarises our argument

Theorem 1

Let $g : \mathbb{R}^n \to \mathbb{R}^n$ be a once differentiable function and $\mathbf{x}_* \in \mathbb{R}^n$ be a fixed point of g. If $\|\mathbf{J}_g(\mathbf{x}_*)\| < 1$, then there exists an $\varepsilon > 0$, such that for all $x^{(0)}$ with $\|\mathbf{x}^{(0)} - \mathbf{x}_*\| < \varepsilon$

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

$$\lim_{k\to\infty} \frac{\|\mathbf{x}^{(k+1)} - \mathbf{x}_*\|}{\|\mathbf{x}^{(k)} - \mathbf{x}_*\|} = \|\mathbf{J}_g(\mathbf{x}_*)\|,$$

i.e. the smaller the norm of the Jacobian, the faster the convergence.

We notice the central role the Jacobian matrix norm plays for both determining if the iterations converge and at which rate. It will therefore be important for us to compute such matrix norms. We will provide more details on this point in the final sections of Direct methods for linear systems. For now we only note the key result that

For any matrix $\mathbf{M} \in \mathbb{R}^{m\times n}$

$$\|\mathbf{M}\| = \sqrt{\lambda_\text{max}^\text{abs}(\mathbf{M}^T\mathbf{M})}.$$

where

$$\lambda_\text{max}^\text{abs}(\mathbf{M^T M}) = \max_{i=1,\ldots,n} |\lambda_i(\mathbf{M^T M})| \qquad \textbf{with } \lambda_i(\mathbf{M}^T\mathbf{M}) \textbf{ eigenvalues of } \mathbf{M}^T\mathbf{M}$$

is the *largest absolute* eigenvalue of the matrix $\mathbf{M}^T\mathbf{M}$.

Examples

To develop some intuition about $\lambda_\text{max}^\text{abs}$ we consider the diagonal matrices:

$$\begin{aligned}
\mathbf{A} &= \begin{pmatrix} 5 & 0 \\ 0 & 10 \end{pmatrix} & \qquad \lambda_\text{max}^\text{abs}(\mathbf{A}) &= 10 \\
\mathbf{B} &= \begin{pmatrix} -5 & 0 \\ 0 & 10 \end{pmatrix} & \qquad \lambda_\text{max}^\text{abs}(\mathbf{B}) &= 10 \\
\mathbf{C} &= \begin{pmatrix} -500 & 0 \\ 0 & 10 \end{pmatrix} & \qquad \lambda_\text{max}^\text{abs}(\mathbf{C}) &= 500
\end{aligned}$$

With this in place we go back to our circle example:

Intersecting circle and parabola (continued)

Based on Theorem 1 we now want to understand why searching the intersection of circle and parabola using fixed-point iterations on the map g_C is successful while the iterations using g_D fails. We already computed the respective Jacobians as

$$\mathbf{J}_{g_C}(\mathbf{x}) = \begin{pmatrix} 0 & \frac{1}{4\sqrt{\frac{x_2}{2}}} \\ \frac{-x_1}{\sqrt{1-x_1^2}} & 0 \end{pmatrix} \qquad \mathbf{J}_{g_D}(\mathbf{x}) = \begin{pmatrix} 0 & \frac{-x_2}{\sqrt{1-x_2^2}} \\ 4x_1 & 0 \end{pmatrix}$$

According to Theorem 1 to understand the convergence properties we need to investigate $\|\mathbf{J}_{g_C}(\mathbf{x}_*)\|$ and $\|\mathbf{J}_{g_D}(\mathbf{x}_*)\|$ and for this compute respectively the largest eigenvalues of

$$\mathbf{J}_{g_C}(\mathbf{x}_*)^T\mathbf{J}_{g_C}(\mathbf{x}_*) = \begin{pmatrix} \frac{1}{8x_{2,*}} & 0 \\ 0 & \frac{x_{1,*}^2}{1-x_{1,*}^2} \end{pmatrix} \qquad \mathbf{J}_{g_D}(\mathbf{x}_*)^T\mathbf{J}_{g_D}(\mathbf{x}_*) = \begin{pmatrix} \frac{x_{2,*}^2}{1-x_{2,*}^2} & 0 \\ 0 & 16x_{1,*}^2 \end{pmatrix}$$

Since these are diagonal matrices their eigenvalues can be directly read off. In particular we compute numerically:

```
JgC = 2×2 Matrix{Float64}:
        0.0        0.400121
       -0.800243   0.0
```

```julia
JgC = [0                           1 / (4sqrt(x2_ast/2));
       -x1_ast / sqrt(1-x1_ast^2)                    0]
```

```
2×2 Matrix{Float64}:
 0.160097  0.0
 0.0       0.640388
```

```julia
JgC*JgC'
```

Therefore $\|\mathbf{J}_{g_C}(\mathbf{x}_*)\| \approx \sqrt{0.640} \approx 0.800 < 1$

```
JgD = 2×2 Matrix{Float64}:
       0.0       -1.24962
       2.49924    0.0
```

```julia
JgD = [0          -x2_ast / sqrt(1-x2_ast^2);
       4x1_ast                            0]
```

```
2×2 Matrix{Float64}:
 1.56155  0.0
 0.0      6.24621
```

```julia
JgD*JgD'
```

and $\|\mathbf{J}_{g_D}(\mathbf{x}_*)\| \approx \sqrt{6.25} \approx 2.50 > 1$.

Provided we start sufficiently close to $\mathbf{x}_*$ fixed-point iterations for g_C therefore converge while fixed-point iterations for g_D diverge.

```
# TODO(md"Table contrast scalar versus multi-dimensional case in a table.")
```

Stopping criteria and residual

Let us come back to the question at which point to stop the fixed-point iteration algorithm. Let ϵ denote the tolerance to which we want to determine $\mathbf{x}_*$, i.e. we would like to stop the iterations as soon as the error is smaller, $\|\mathbf{x}^{(k)} - \mathbf{x}_*\| < \epsilon$. Since $\mathbf{x}_*$ is not known, this expression cannot be computed during the iterations. We thus seek an alternative approach.

In a given step $\mathbf{x}^{(k)}$ we have in general not yet achieved our goal, i.e. $g(\mathbf{x}^{(k)}) \neq \mathbf{x}^{(k)}$. An idea is thus to consider exactly the descrepancy

$$\mathbf{r}^{(k)} = g(\mathbf{x}^{(k)}) - \mathbf{x}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)},$$

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

Algorithm: Fixed-point iteration stopping criterion

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

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

```
fixed_point_iterations_simple (generic function with 1 method)
```

```julia
function fixed_point_iterations_simple(g, xstart; tol=1e-6)
    # g:      Fixed-point function
    # xstart: Initial guess
    # tol:    Tolerance

    rᵏ = Inf
    xᵏ = xstart
    k  = 0
    while norm(rᵏ) ≥ tol
        xᵏ⁺¹ = g(xᵏ)
        rᵏ   = xᵏ⁺¹ - xᵏ

        k  = k + 1  # Update k
        xᵏ = xᵏ⁺¹   # Update xᵏ accordingly
    end

    # Return results as a named tuple
    (; fixed_point=xᵏ, residual=rᵏ, n_iter=k)
end
```

We apply this function:

```
res_simple =
▸ (fixed_point = [0.624811, 0.780776], residual = [-1.11022e-16, 8.32667e-15], n_iter = 57)
```

```julia
res_simple = fixed_point_iterations_simple(gC, [0.4, 0.3]; tol=1e-14)
```

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

```
▸ [0.624811, 0.780776]
```

```julia
res_simple.fixed_point
```

or the number of iterations required

```
57
```

```julia
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)
```

```julia
function fixed_point_iterations(g, xstart; tol=1e-6, maxiter=100)
    # g:      Fixed-point function
    # xstart: Initial guess
    # tol:    Tolerance

    history_x = [xstart]
    history_r = empty(history_x)

    rᵏ = Inf      # For initial pass in while loop
    xᵏ = xstart   # Starting point of iterations
    k  = 0
    while k < maxiter && norm(rᵏ) ≥ tol
        xᵏ⁺¹ = g(xᵏ)
        rᵏ   = xᵏ⁺¹ - xᵏ
        push!(history_r, rᵏ)

        k  = k + 1  # Update k
        xᵏ = xᵏ⁺¹   # Update xᵏ accordingly
        push!(history_x, xᵏ)  # Push next point to the history
    end

    # Return results as a named tuple
    (; fixed_point=xᵏ, residual=rᵏ, n_iter=k, history_x, history_r)
end
```

```
▸ (fixed_point = [0.624811, 0.780776], residual = [-1.11022e-16, 8.32667e-15], n_iter = 57)
```

```julia
fixed_point_iterations_simple(gC, [0.4, 0.3]; 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 $\|g(\mathbf{x}^{(k)}) - \mathbf{x}^{(k)}\| < \epsilon$ **always implies** $\|\mathbf{x}^{(k)} - \mathbf{x}_*\| < \epsilon$.
- To explain this, let us consider the **scalar case** $g : \mathbb{R} \to \mathbb{R}$. In this setting we can derive the **residual-error relationship** *(see derivation below)*

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

for some $\xi^{(k)} \in [x_*, x^{(k)}]$. Note, that this is just a **conceptional 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 $x^{(k)} \to x_*$ as $k \to \infty$. Therefore the interval $[x_*, x^{(k)}]$ gets smaller and smaller, such that necessarily $\xi^{(k)} \to x_*$ and $g'(\xi^{(k)}) \to g'(x_*)$ as $k \to \infty$. 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 ϵ.
- In contracst if $|g'(x_*)| = 0$ than $|r^{(k)}| < \epsilon$ is an excellent stopping criterion as $|x^{(k)} - x_*| = |r^{(k)}|$ as $k \to \infty$.
- For the **multi-dimensional case** *(see detailed discussion below)* the conclusion is similarly that **the residual is a reliable error indicator** if **no eigenvalue of the Jacobian $\mathbf{J}_g$ is close to 1**. See below for a more detailed discussion.

▶ **Derivation of the residual-error relationship for the scalar case**

▶ **In-depth discussiong of the multi-dimensional case**

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

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

$$\lim_{k\to\infty} \frac{\|\mathbf{x}^{(k+1)} - \mathbf{x}_*\|}{\|\mathbf{x}^{(k)} - \mathbf{x}_*\|^q} = C \tag{7}$$

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

  $$\|\mathbf{x}^{(k+1)} - \mathbf{x}_*\| = C\|\mathbf{x}^{(k)} - \mathbf{x}_*\| = C^2\|\mathbf{x}^{(k-1)} - \mathbf{x}_*\| = \alpha\, C^k$$

  where α is some constant. Taking the logs we get

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

  which is a straight line.

Visual inspection: Log of error

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

So let us inspect graphically the convergence of the fixed-point map g_1 in comparison to another fixed-point map g_2 defined as

```
g₁ (generic function with 1 method)
```

```julia
g₁(x) = x - (1/2) * (log(x + 1) + x - 2)
```

```
g₂ (generic function with 1 method)
```

```julia
g₂(x) = 2 - log(x + 1)
```

Both converge to the same fixed point.

```
fp = 1.2079400315693
```

```julia
fp = 1.2079400315693  # Approximate fixed point
```

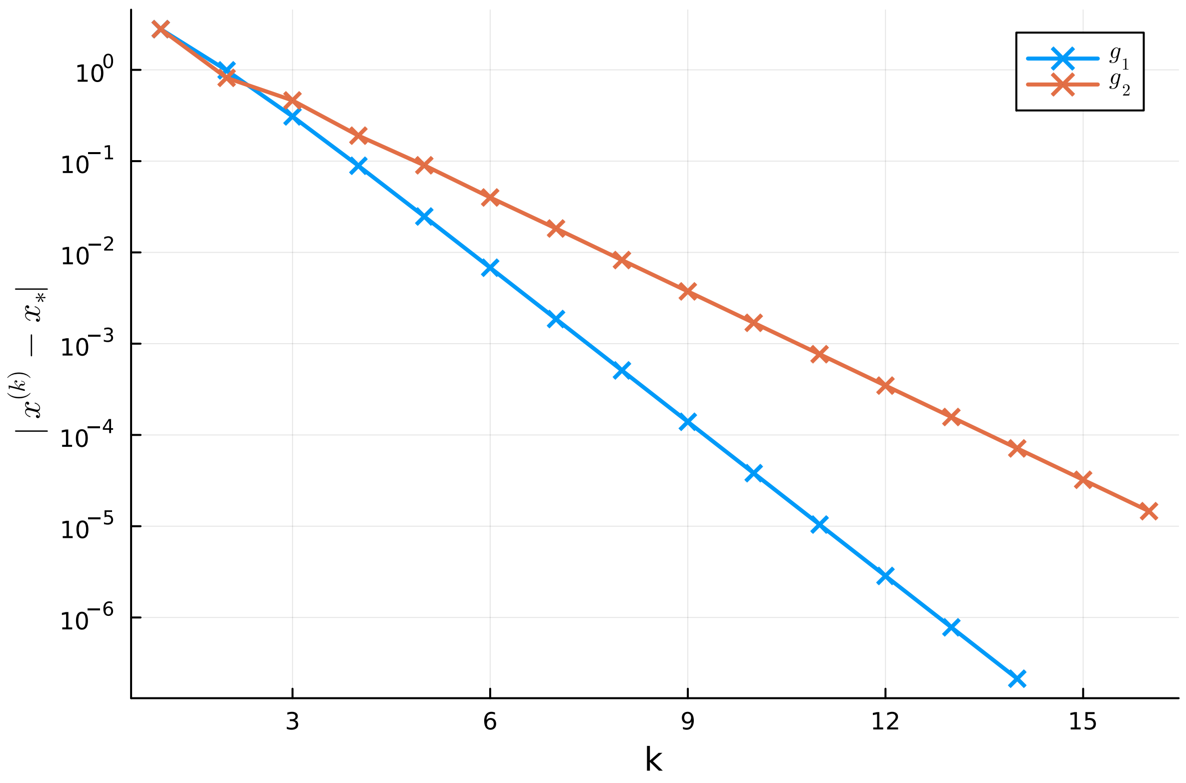
```
1.6653345369377348e-14
```

```julia
g₁(fp) - fp
```

```
3.352873534367973e-14
```

```julia
g₂(fp) - fp
```

We notice:



```julia
let
    fp = 1.2079400315693  # Approximate fixed point
    p = plot(; yaxis=:log, xlabel="k", ylabel=L"\|x^{(k)} - x_\ast|")

    results_g₁ = fixed_point_iterations(g₁, 4.0; maxiter=15)
    errors_g₁ = [abs(xn - fp) for xn in results_g₁.history_x]
    plot!(p, errors_g₁, label=L"g_1", mark=:x, lw=2)

    results_g₂ = fixed_point_iterations(g₂, 4.0, maxiter=15)
    errors_g₂ = [abs(xn - fp) for xn in results_g₂.history_x]
    plot!(p, errors_g₂, label=L"g_2", mark=:x, lw=2)

    yticks!(p, 10.0 .^ (-6:0))
end
```

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

Visual inspection: Residual ratio

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**, i.e. for the scalar case (6)

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

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

$$\lim_{k\to\infty} \frac{|r^{(k+1)}|}{|r^{(k)}|^q} = \lim_{k\to\infty} \frac{|1-g'(\xi^{(k+1)})|}{|1-g'(\xi^{(k)})|^q} \frac{|x^{(k+1)}-x_*|}{|x^{(k)}-x_*|^q} \overset{(5)}{=} \frac{1}{|1-g'(x_*)|^{q-1}} C.$$

In other words if the residual ratios approach a constant for a chosen q, then we have q-th order convergence. For the **multi-dimensional case** we would equivalently study the ratio of the norms

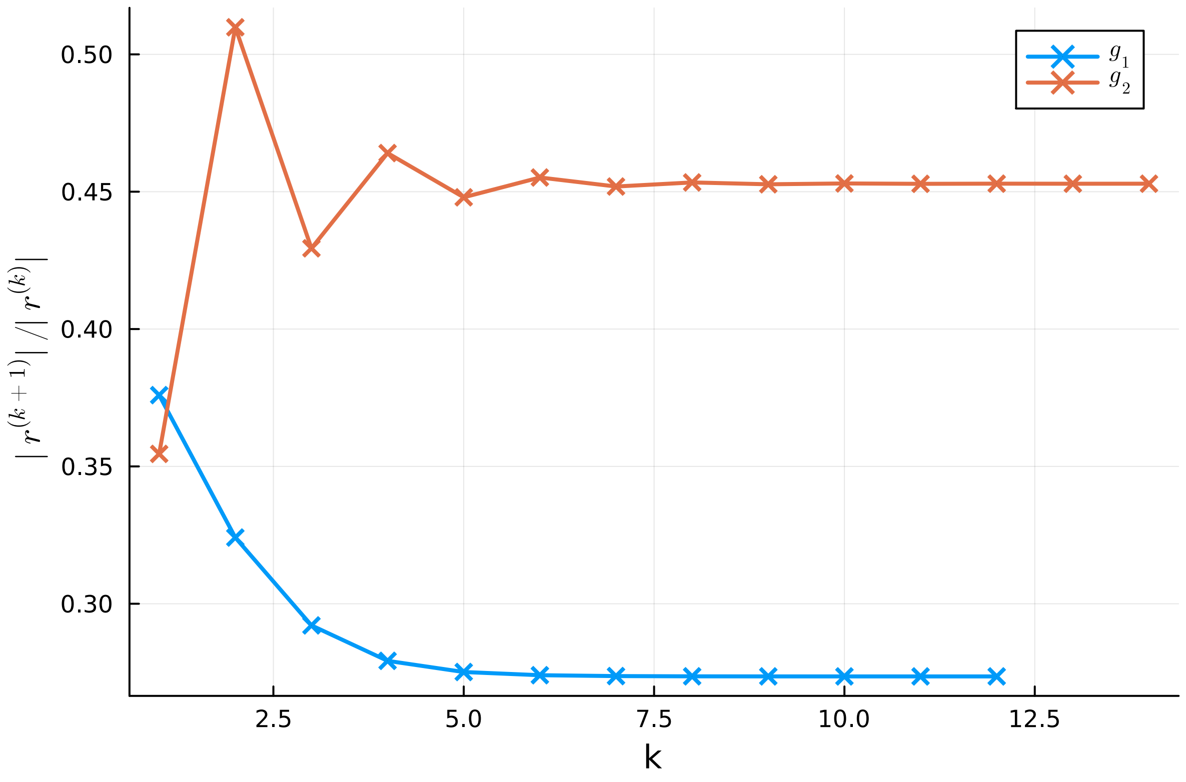
$$\lim_{k\to\infty} \frac{\left\|\mathbf{r}^{(k+1)}\right\|}{\left\|\mathbf{r}^{(k)}\right\|^q}.$$

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

$$\lim_{k\to\infty} \frac{\left\|\mathbf{r}^{(k+1)}\right\|}{\left\|\mathbf{r}^{(k)}\right\|} = C,$$

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

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



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 verifyable condition, it is guaranteed to converge.

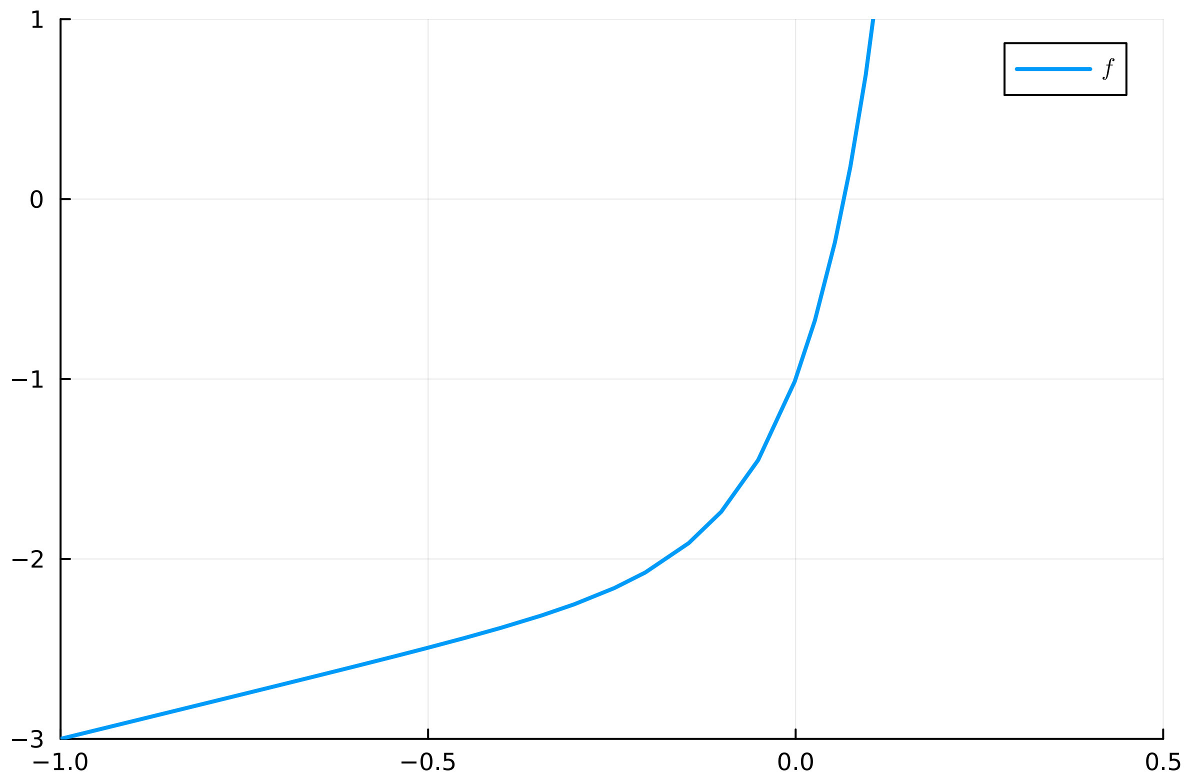
Going back to our diode circuit problem, we find that the non-linear, scalar fixed-point problem $g_\text{exp}(v_D) = v_D$ can be written as the root-finding problem $f(x) = 0$ for the function

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

We select the parameters and define f

```
begin
    i0 = 1.0
    v0 = 0.1
    R = 1.0
    V = 1.0

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

followed by a plot to get an idea of the function:



```julia
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:

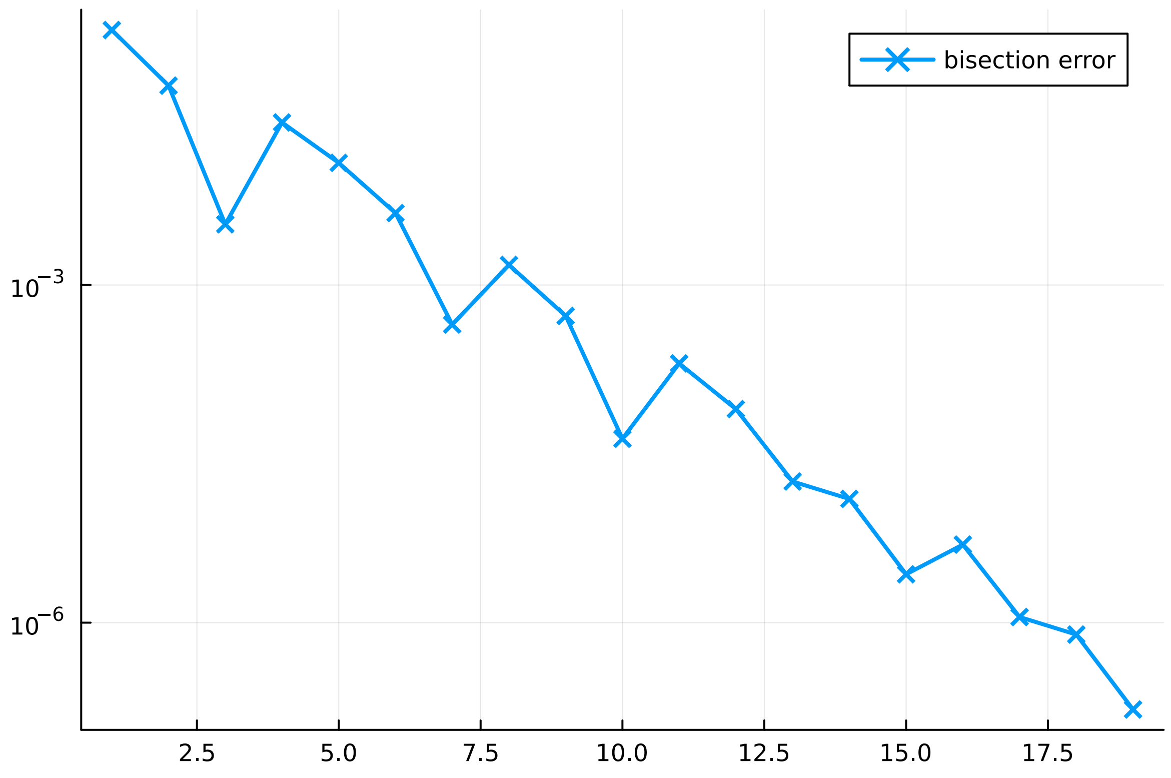
- If $f(x_m)f(a) < 0$, then there is a root in $[a, x_m]$
- If $f(x_m)f(b) < 0$, then root is in $[x_m, b]$
- If $f(x_m) = 0$, we found a root.

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)
```

```julia
function bisection_method(f, a, b; tol=1e-6)
    @assert a ≤ b
    @assert f(a) * f(b) < 0  # Otherwise the assumptions are not true

    # Initialise
    k  = 0
    xᵏ = (a + b) / 2

    history_x = Float64[]  # Empty Array, but only for Float64 numbers
    while abs(b - a) / 2 ≥ tol
        k = k + 1
        if f(xᵏ) * f(a) < 0
            b = xᵏ  # New interval [a, xᵏ]
        else
            a = xᵏ  # New interval [xᵏ, b]
        end

        xᵏ = (a + b) / 2
        push!(history_x, xᵏ)
    end

    (; root=xᵏ, history_x, n_iter=k)
end
```

Its convergence plot again suggests a linear convergence:



```julia
let
    # First get an almost exact root
    reference = bisection_method(f, -0.5, 0.5; tol=1e-12)

    # Now run again to plot convergence
    result = bisection_method(f, -0.5, 0.5)
    errors  = [abs(xn - reference.root) for xn in result.history_x]

    plot(errors, label="bisection error", mark=:x, lw=2, yaxis=:log)
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

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

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

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

in order to have an error below the threshold ϵ. Rearringing (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 ϵ. 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 ϵ 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 $[a, b]$ wherea 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 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.

We consider the case of a scalar **fixed-point map** $g : \mathbb{R} \to \mathbb{R}$ and revisit the **Taylor expansion** (3) of g, which we used to analyse the convergence of the fixed-point iterations $x^{(k+1)} = g(x^{(k)})$. Continuing the expansion to **second order** we notice for the error

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

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

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

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

Theorem 3

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

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

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

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

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

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

To develop these methods assume that the root of a function $f : \mathbb{R}^n \to \mathbb{R}^n$ is $\mathbf{x}_*$ and we are given a point $\mathbf{x}$, which is close to $\mathbf{x}_*$. We consider a Taylor expansion of f around $\mathbf{x}$ to first order:

$$\mathbf{0} = f(\mathbf{x}_*) = f(\mathbf{x}) + \mathbf{J}_f(\mathbf{x})\,(\mathbf{x}_* - \mathbf{x}) + O(\text{small}). \tag{10}$$

Where we made the condition $f(\mathbf{x}_*) = \mathbf{0}$ explicict.

Assuming that $\det \mathbf{J}_\mathbf{f}(\mathbf{x}) \neq 0$, i.e. that the Jacobian is non-singular, we can rearrange this to

$$\mathbf{0} = \left(\mathbf{J}_\mathbf{f}(\mathbf{x})\right)^{-1} f(\mathbf{x}) + (\mathbf{x}_* - \mathbf{x}) + O(\text{small}).$$

If $\mathbf{x}$ is close to $\mathbf{x}_*$, thus $\mathbf{x} - \mathbf{x}_*$ is small, then the last term $O(\text{small})$, which actually depends on $(\mathbf{x}_* - \mathbf{x})^2$ and higher powers of $(\mathbf{x}_* - \mathbf{x})$ is even smaller.

Neglecting it completely we can further develop this to an approximation for x_* as

$$\mathbf{0} \simeq (\mathbf{J_f}(\mathbf{x}))^{-1} f(\mathbf{x}) + \mathbf{x}_* - \mathbf{x} \qquad \Longrightarrow \qquad \mathbf{x}_* \simeq \mathbf{x} - (\mathbf{J_f}(\mathbf{x}))^{-1} f(\mathbf{x}).$$

If we denote the RHS by $g_{\text{Newton}}(\mathbf{x}) = \mathbf{x} - (\mathbf{J}_f(\mathbf{x}))^{-1} f(\mathbf{x})$, then a root of f is a fixed point of g_{Newton} and vice versa. Performing fixed-point iterations on g_{Newton} is the idea of Newton's method:

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

Given a once differentiable function $f : \mathbb{R}^n \to \mathbb{R}^n$, a starting value $\mathbf{x}^{(0)} \in \mathbb{R}^n$ perform fixed-point iterations

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

on the map

$$g_{\text{Newton}}(\mathbf{x}) = \mathbf{x} - (\mathbf{J}_f(\mathbf{x}))^{-1} f(\mathbf{x}). \tag{11}$$

For scalar problems, i.e. when we consider finding the root of $f : \mathbb{R} \to \mathbb{R}$, we can identify the Jacobian (which formally is a 1×1 matrix) by the single element $\frac{\partial f}{\partial x_1} = f'(x_1)$. Inversion of such a 1×1 matrix is simply division.

Therefore **for scalar problems** we can equivalently write

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

provided that $f'(x) \neq 0$ — which is equivalent to the requirement that $\mathbf{J}_f$ is invertible in the scalar case.

Graphical interpretation

- 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

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.

For simplicity we will only consider the scalar version of Newton's method

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

To apply Theorem 3 we study its derivatives at the fixed point x_*. We obtain

$$\begin{aligned}
g'_{\text{Newton}}(x) &= 1 - \frac{f'(x)\, f'(x) - f''(x)\, f(x)}{\left(f'(x)\right)^2} = \frac{f''(x)\, f(x)}{\left(f'(x)\right)^2} \\
g''_{\text{Newton}}(x) &= \frac{\left(f'(x)\right)^3 f''(x) + f(x) \left(f'(x)\right)^2 f'''(x) - 2 f(x) \left(f''(x)\right)^2 f'(x)}{\left(f'(x)\right)^4}
\end{aligned}$$

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

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

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

A similar line of argument can be formulated for the multi-dimensional case. We summarise

Theorem 4: Convergence of Newton's method

Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a twice differentiable function and $\mathbf{x}_*$ a root of f. If $\mathbf{J}_f(\mathbf{x}_*)$ is invertible (respectively $f'(x_*) \neq 0$ for the scalar case) and the second derivative of f is non-zero at $\mathbf{x}_*$, then Newton's method converges quadratically for every $\mathbf{x}^{(0)}$ sufficiently close to $\mathbf{x}_*$.

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_*.
- We further noted that for the scalar case $f : \mathbb{R} \to \mathbb{R}$ the convergence rate is

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

- Importantly, 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_{Newton} the implementation is straightforward by employing the `fixed_point_iterations` function we already implemented above:

```
newton_fp (generic function with 1 method)
```

```julia
function newton_fp(f, jac, xstart; maxiter=40, tol=1e-6)
    # f: Function of which we seek the roots
    # jac: Function, which evaluates its Jacobian or derivative
    # xstart: Start of the iterations
    # maxiter: Maximal number of iterations
    # tol: Convergence tolerance

    # Define the fixed-point function g_Newton using f and df
    # Here "\" is the operator for solving linear systems, see discussion below.
    g_Newton(x) = x - jac(x) \ f(x)

    # Solve for its fixed point:
    fixed_point_iterations(g_Newton, xstart; tol, maxiter)
end
```

Note, that the formal definition of the fixed-point function

$$g_{\text{Newton}}(\mathbf{x}) = \mathbf{x} - (\mathbf{J}_f(\mathbf{x}))^{-1} f(\mathbf{x})$$

asked us to compute the expression $\mathbf{z}^{(k)} = \left(\mathbf{J}_f(\mathbf{x}^{(k)})\right)^{-1} f(\mathbf{x})$, i.e. the inverse of $\left(\mathbf{J}_f(\mathbf{x}^{(k)})\right)$ followed by a matrix-vector product with the vector $f(\mathbf{x})$. However, the implementation achieves this by invoking the `\` Julia operator, which solves the linear system

$$\mathbf{J}_f(\mathbf{x}^{(k)})\, \mathbf{z}^{(k)} = f(\mathbf{x}) \tag{12}$$

for the vector $\mathbf{z}^{(k)}$. On paper both computations are equivalent, but in practice using the `\` operator is more numerically stable. We will discuss more on solving linear systems in the chapter Direct methods for linear systems.

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

Algorithm 2: Newton's method (conventional)

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

1. Compute the right-hand side $\mathbf{y}^{(k)} = f(\mathbf{x}^{(k)})$ and Jacobian $\mathbf{A}^{(k)} = \mathbf{J}_f(\mathbf{x}^{(k)})$.
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 $\|\mathbf{r}^{(k)}\| < \epsilon$.

Some remarks:

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

$$\mathbf{r}^{(k)} = g_{\text{Newton}}(\mathbf{x}^{(k)}) - \mathbf{x}^{(k)} = \mathbf{x}^{(k)} - (\mathbf{J}_f(\mathbf{x}))^{-1} f(\mathbf{x}) - \mathbf{x}^{(k)} = -(\mathbf{J}_f(\mathbf{x}))^{-1} f(\mathbf{x}),$$

  and that $\mathbf{r}^{(k)} = -\mathbf{z}^{(k)}$ with the $\mathbf{z}^{(k)}$ we employed in equation (12).
- Since in the Newton setting g_{Newton} exhibits quadratic convergence, the **residual-based stopping criterion** $\|\mathbf{r}^{(k)}\| < \epsilon$ is actually **extremely reliable**, see the discussion after Theorem 3.

A corresponding implementation of Algorithm 2 is:

```
newton (generic function with 1 method)
```

```julia
function newton(f, jac, xstart; maxiter=40, tol=1e-8)
    # f: Function of which we seek the roots
    # jac: Function, which evaluates its Jacobian or derivative
    # xstart: Start of the iterations
    # maxiter: Maximal number of iterations
    # tol: Convergence tolerance
    history_x = [float(xstart)]
    history_r = empty(history_x)

    r = Inf     # Dummy to enter the while loop
    x = xstart  # Initial iterate
    k = 0

    # Keep running the loop when the residual norm is beyond the tolerance
    # and we have not yet reached maxiter
    while norm(r) ≥ tol && k < maxiter
        k = k + 1

        y = f(x)      # Function value
        A = jac(x)    # Jacobian
        r = -(A \ y)  # Newton step
        x = x + r     # Form next iterate

        push!(history_r, r)  # Push newton step and
        push!(history_x, x)  # next iterate to history
    end

    (; root=x, n_iter=k, history_x, history_r)
end
```

▶ **Teacher hint: Live coding**

To see how this method performs we compare against plain fixed-point iterations on the function

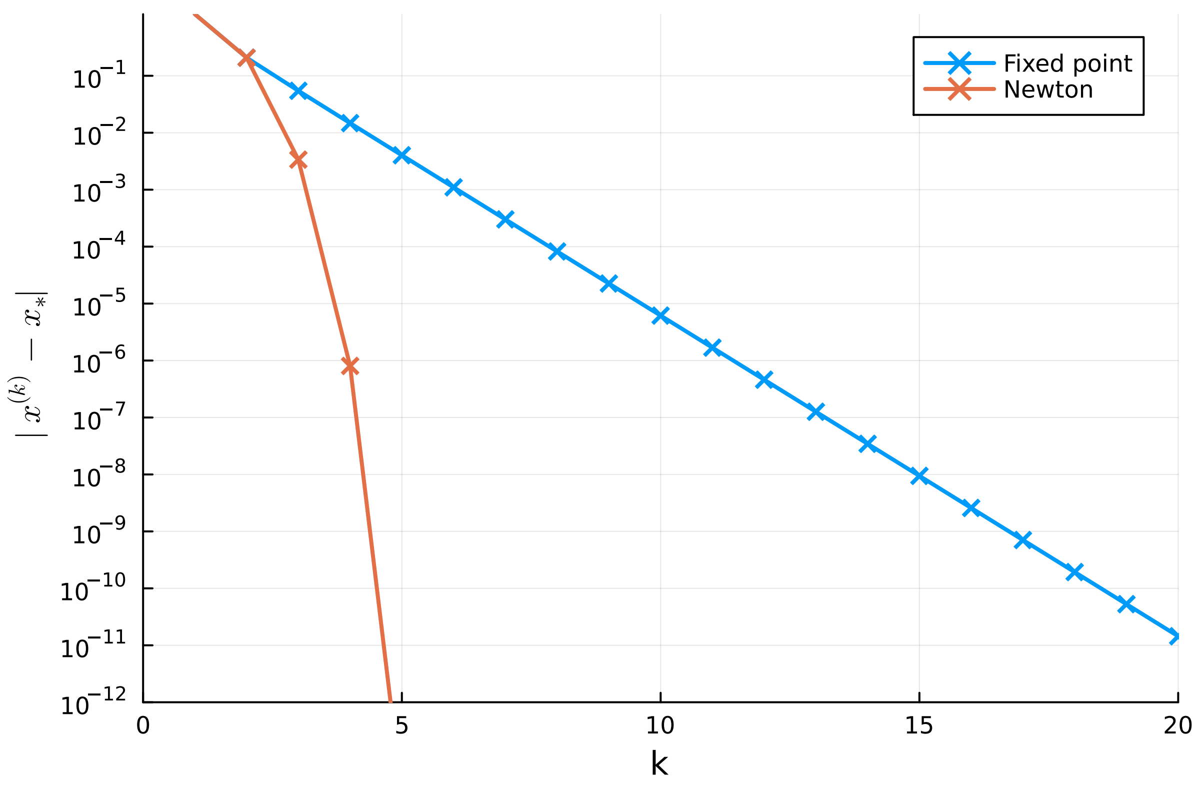
$$g_1(x) = x - \frac{1}{2}(\log(x+1) + x - 2)$$

which we also employed earlier. An equivalent root-finding problem is $f_1(x) = 0$ with

$$f_1(x) = g_1(x) - x = -\frac{1}{2}(\log(x+1) + x - 2).$$

```
f₁ (generic function with 1 method)
```

```julia
f₁(x) = g₁(x) - x
```

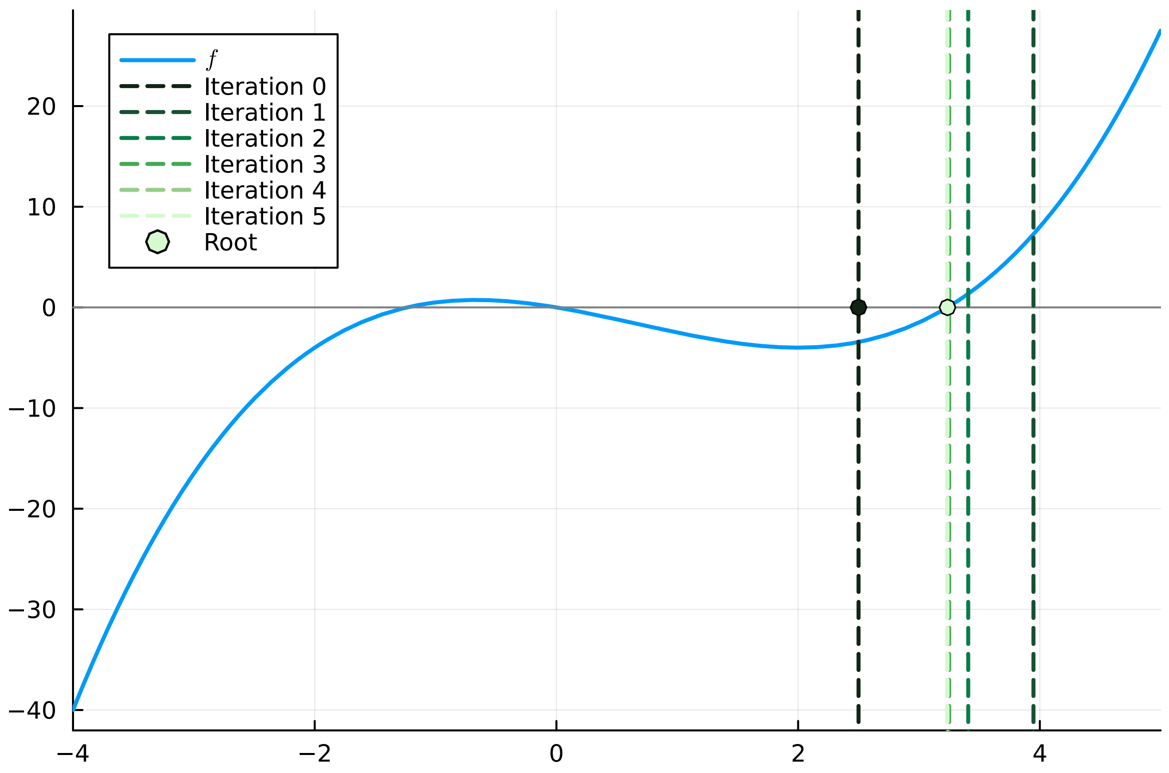


```julia
let
    reference = 1.2079400315693  # Approximate root

    p = plot(yaxis=:log, xlims=(0, 20), ylims=(1e-12, Inf),
             xlabel="k", ylabel=L"|x^{(k)} - x_\ast|")

    # Run fixed-point on g₁
    result = fixed_point_iterations(g₁, 0.0; tol=1e-12)
    errors  = [abs(xn - reference) for xn in result.history_x]
    plot!(p, errors, label="Fixed point", mark=:x, lw=2)

    # For Newton we need the derivative of f.
    # An easy way to obtain this derivative is to use algorithmic differentiation:
    df(x) = ForwardDiff.derivative(f₁, x)

    # With this we run Newton
    result = newton(f₁, df, 0.0; tol=1e-12)
    errors = [abs(xn - reference) for xn in result.history_x]
    plot!(p, errors, label="Newton", mark=:x, lw=2)
    yticks!(p, 10.0 .^ (-12:-1))

    p
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:

- `x0 =` 2.5



Finally, let us investigate on a more involved problem what quadratic convergence means in practice.

We will consider the problem $f(\mathbf{x}) = \mathbf{0}$ with $\mathbf{f} : \mathbb{R}^3 \to \mathbb{R}^3$ given by

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

The Jacobian can be computed as

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

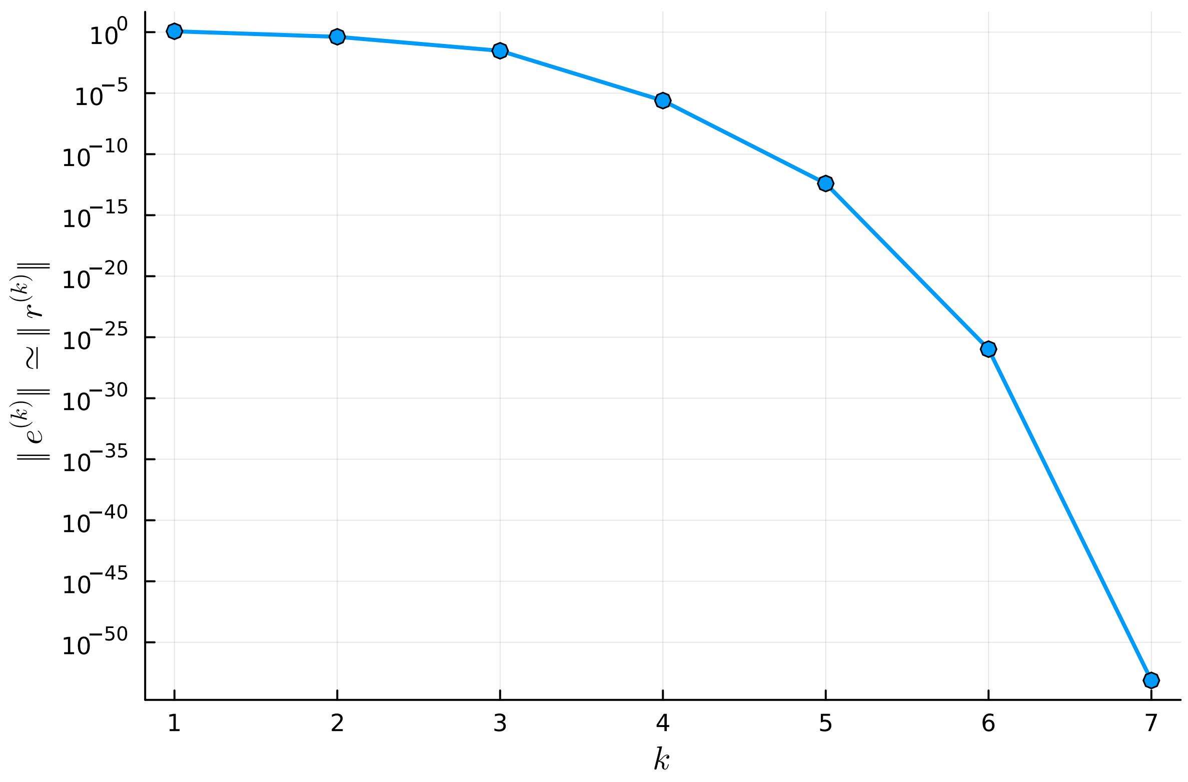
This example can be implemented in Julia as follows:

```
begin
    func(x) = [
        -x[1] * cos(x[2]) - 1,
        x[1] * x[2] + x[3],
        exp(-x[3]) * sin(x[1] + x[2]) + x[1]^2 - x[2]^2
    ]
    jac_func(x) = [
        -cos(x[2])        x[1]*sin(x[2])    0;
        x[2]              x[1]              1;
        exp(-x[3])*cos(x[1]+x[2]) + 2x[1]  exp(-x[3])*cos(x[1]+x[2]) - 2x[2] exp(-
        x[3])*sin(x[1]+x[2])
    ]
end;
```

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 in this example to see more closely what is going on:

```
res = newton(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:



```
plot(norm.(res.history_r); yaxis=:log, label="", xlabel=L"k", ylabel=L"\Vert
e^{(k)} \Vert \simeq \Vert r^{(k)} \Vert", mark=:o, lw=2, yticks=10.0 .^ (0:-5:-60))
```

Notice the *extremely* small error norms of less than 10^{-50} : From our chosen starting point it takes **only 7 iterations** to get the result **accurate to more than 50 digits** !

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.

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

Obervations

- 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.
- The convergence behaviour of Newton can be 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 phaenomenon are Newton fractals.

Optional: Secant method

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

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 $x^{(0)} = x_{\text{start}}$.
2. **Iteration:** For $k = 1, 2, 3, \ldots$ we perform the same

iterative procedure, advancing $x^{(k-1)}$ into $x^{(k)}$.

3. **Check for convergence:** Once $x^{(k)}$ is similar enough to $x^{(k-1)}$ 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 g, i.e. $x^{(k)} = g(x^{(k-1)})$. In this formulation step 3 thus does nothing else than checking whether the iterates $x^{(k)}$ no longer change. Or, put in other words, if we have found a **fixed point** of g.

Observation: Iterative algorithms are fixed-point problems

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

Numerical analysis