Click here to view the PDF version.

```
# Block installing Julia packages needed to run this notebook.
begin
    using Plots
    using PlutoUI
    using PlutoTeachingTools
    using LaTeXStrings
    using Symbolics
    using NLsolve
    using HypertextLiteral: @htl, @htl_str
    using Printf
end
```

Table of Contents

Introduction

Opening remarks

Slides of the presentation on Moodle: These summarise the following paragraphs and the content of the course in a short presentation.

In modern practice **every scientist employs computers**. Some use them to symbolically derive equations for complex theories, others perform simulations, yet others analyse and visualise experimental results. In all these cases we encounter **mathematical problems**, which we need to **solve using numerical techniques**. The point of this lecture is to learn how to **formalise such numerical procedures** mathematically, **implement them** using the Julia programming language and **analyse using pen and paper** why some methods work better, some worse and some not at all.

You might think learning about this is a bit of an overkill and probably not very useful for your future studies. But let us look at a few examples, which give an overview what we will study in more depth throughout the course.

How this course works

Context of the course:

- In this lecture we focus very much on **teaching you the high-level ideas** of key numerical algorithms. We keep proofs to a minimal level, where they are easy or help understanding, but **not all technicalities can be avoided**.
- The **role of this lecture** in the study curriculum is to **provide a broad overview of many numerical techniques**, which you will need in future classes. Unfortunately in this lecture **we have little time to go into more interesting applications**. While we do employ examples to motivate our mathematical study, these examples therefore can appear constructed or only loosely connected to the rest of your studies.

How to get the most out of this course:

- **All content** of the lecture is in the **online lecture notes**. Note that these are a **living document**. Expect them to improve over the semester based on errors we find or based on questions that come up during the lecture.

- The **additional PDF documents** are provided as references. Their content is similar and they can serve you as additional learning resources.
- The **blackboard derivations** is only **meant to slow me down** and give you the chance to follow the more technical parts of the derivations and ask questions about them. In principle **everything I say is also written down in the lecture notes**. It is up to you whether you **just want to listen**, you **want to copy all discussion** on the blackboard or you simply **print or otherwise annotate the PDFs** we provide online.
- The **expectation is *not* that you leave the lecture understanding everything** that is said, much rather that you will understand it after you go through the lecture notes once again for yourself.
- With that said **it can be helpful to read through the lecture notes** for the next session **before going to the session** itself.
- In the lecture **I often pause and ask for questions**. This is your chance to slow me down in case you are lost: **any question is appreciated**.

Taking derivatives

Before we dive into the main content of the course, the remainder of this chapter provides some examples for numerical problems and their challenges when solving them numerically. These examples serve to pose motivating questions, which we seek to answer throughout the course.

We first consider the problem of **taking derivatives**. Derivatives characterise the change of a quantity of interest. E.g. the acceleration as a derivative of the velocity indeed characterises how the velocity changes over time.

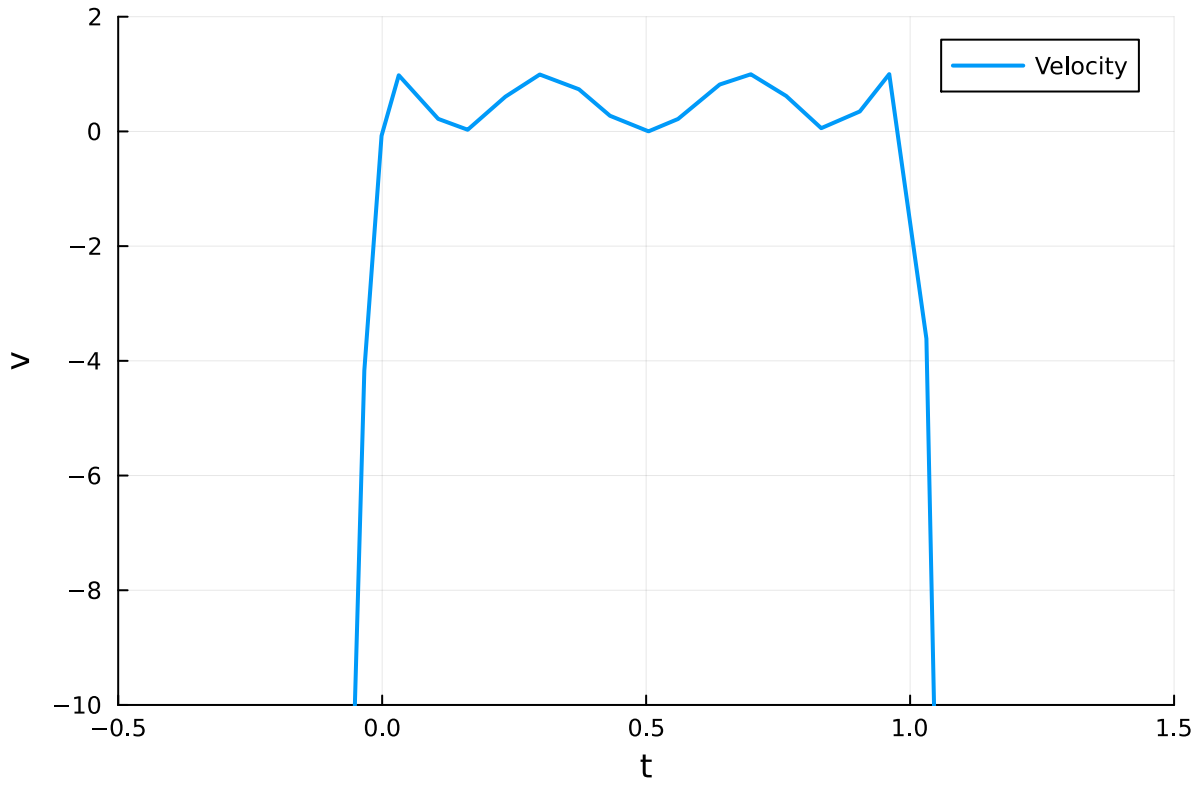
More broadly any experiment in physics or engineering operates by imposing a change to a material or chemical system (e.g. changing temperature, pressure, stress etc.) and observing how it reacts.

As a result **derivatives are therefore at the heart of physics and engineering** and appear everywhere in these subjects. But computing derivatives by hand is not always easy. To see this consider the **innocent-looking velocity function**.

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

```
1 v(t) = 64t * (1 - t) * (1 - 2t)^2 * (1 - 8t + 8t^2)^2
```

We can plot it to get an idea:




```
let
    p = plot(v;
             ylims=(-10, 2),      # Limit on y axis
             xlims=(-0.5, 1.5),   # Limit on x axis
             linewidth=2,         # Width of the blue line
             label="Velocity",    # Label of the graph
             xlabel="t",
             ylabel="v")
end
```

Clearly between 0 and 1 the function changes quite rapidly. To investigate the **acceleration** there **we take the derivative**.

This can be done by hand, but instead we will employ a Julia package (namely `Symbolics`) to take the derivative for us. The result is:

$$64 \left(1-8\,t+8\,t^2\right)^2 \left(1-2\,t\right)^2 \left(1-t\right) - 64 \left(1-8\,t+8\,t^2\right)^2 \left(1-2\,t\right)^2 t - 256 \left(1-8\,t+8$$

```
let
    @variables t           # Define a variable
    dt = Differential(t)   # Differentiating wrt. t
    dv_dt = dt( v(t) )     # Compute dv / dt
    expand_derivatives(dv_dt)
end
```

Clearly far from a handy expression and not the kind of derivative one wants to compute by hand.

So let us **compute this derivative numerically** instead. An idea to do so goes back to the definition of the derivative $v'(t)$ as the limit $h \to 0$ of the slope of secants over an intervall $[t, t+h]$, i.e.

$$v'(t) = \lim_{h \to 0} \frac{v(t+h) - v(t)}{h}.$$

A natural idea is to not fully take the limit, i.e. to take a small $h > 0$ and approximate

$$v'(t) \approx \frac{v(t+h) - v(t)}{h}. \tag{1}$$

The expectation is that as we take smaller and **smaller values for h**, this **converges to the exact derivatives**.

So let's try this at the point `t0 = 0.2` for various values for `h`

```
h_values =
▸ [1.0e-15, 3.16228e-15, 1.0e-14, 3.16228e-14, 1.0e-13, 3.16228e-13, 1.0e-12, 3.16228e-12, 1
```

```
1 h_values = 10 .^ (-15:0.5:1)
```

We compute the derivatives at each value for h ...

```
derivatives =
▸ [8.99281, 9.0755, 9.05942, 9.06497, 9.06719, 9.06567, 9.06603, 9.06608, 9.06609, 9.06609,
```

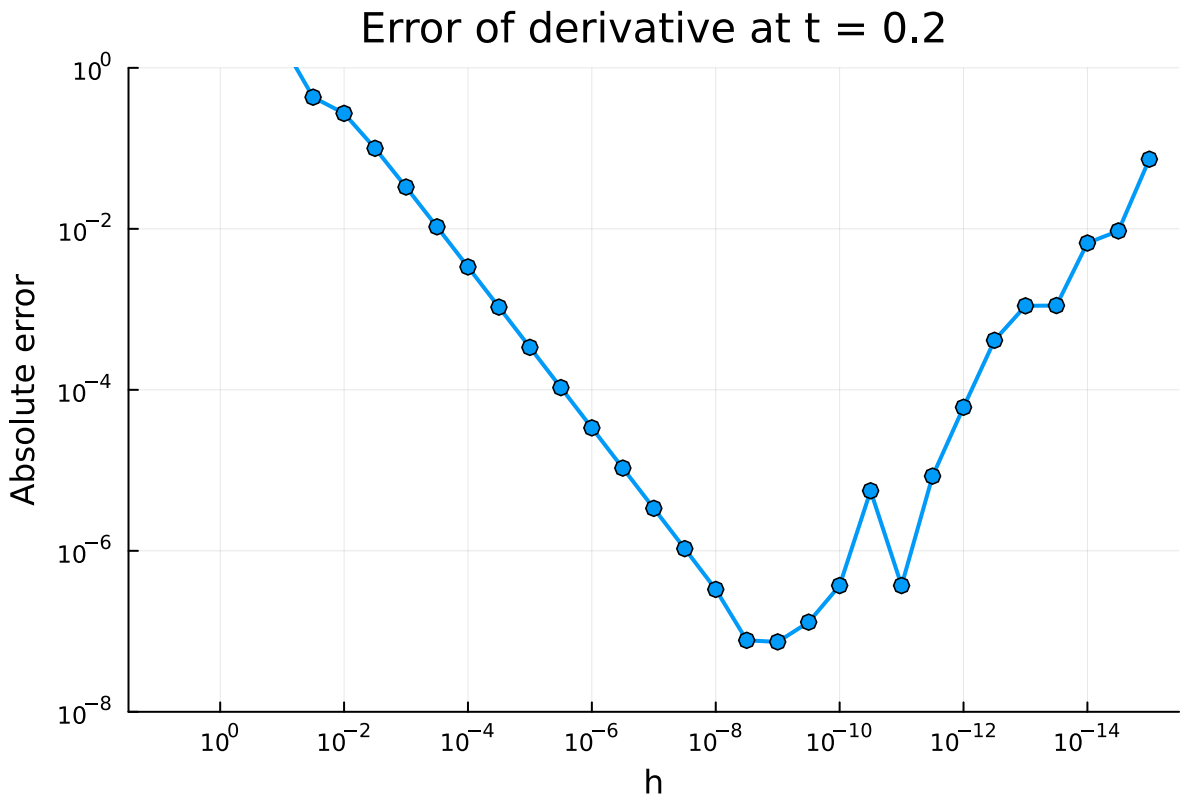
```
1 derivatives = [ ( v(t0 + h) - v(t0) )/h for h in h_values ]
```

... and compute the error against a reference value:

```
▸ [0.0732799, 0.00941812, 0.00666652, 0.00111438, 0.00110504, 0.000412217, 6.06921e-5, 8.47
```

```
1 begin
2     using ForwardDiff # Package for computing derivatives of Julia functions
3     reference = ForwardDiff.derivative(v, t0)
4
5     errors = [abs(d - reference) for d in derivatives]
6 end
```

Finally we plot the errors on a log-log plot:




```
let
    plot(h_values, errors;
        xaxis=:log, yaxis=:log,
        xflip=true,  # Flip order of x axis
        ylims=(1e-8, 1),
        mark=:o,     # Mark all points by circle
        xlabel="h",
        ylabel="Absolute error",
        linewidth=2,
        label="",
        xticks=10.0 .^ (-16:2:0),
        title="Error of derivative at t = $t0",
    )
end
```

We observe that while indeed initially the error decreases as h decreases at some point it starts to increase again with **results worse and worse**.

With this slider you can check this is indeed the case for pretty much all values of t where we take the derivative numerically:

t0 = 0.2

Related questions we will discuss:

- There seems to be some **optimal value** for h. Is it independent of the function to be differentiated ? Can we **estimate the optimal value** somehow ?
- With this algorithm there seems to be some minimal error we can achieve. Are there **more accurate derivative formulas** ?
- The convergence plot as h decreases seems to have the same slope (convergence rate) for all points t. Does this slope depend on f ? How **can we reach faster convergence** ?

Moreover numerical approximations of derivatives are key ingredients when solving **differential equations**. For example, suppose we know that the voltage of an electrical system changes in time according to the derivative

$$\frac{du(t)}{dt} = \sin\left[(u+t)^2\right]$$

and we know that at time $t = 0$ the voltage is $u(0) = -1$. Our goal is to know the behaviour of $u(t)$ for *all future values* of t.

As we will discuss towards the end of this course (in Initial value problems), approximating the derivative $\frac{du(t)}{dt}$ by a formula like (1) will lead to a family of numerical schemes, which achieves exactly that.

Representing numbers on a computer

Let us try to shed some light why results in the previous example on numerically computing derivatives get worse and worse as we take smaller stepsizes h. For this purpose let us reconsider the expression for approximating the derivative $v'(t)$, which we obtained in (1), namely

$$v'(t) \approx \frac{v(t+h) - v(t)}{h}.$$

As we saw when h gets smaller this approximation gets worse in practice.

Let us disect one particular example in detail. We consider $v(t) = t$, where one computes

$$\frac{v(t+h) - v(t)}{h} = \frac{(t+h) - t}{h},$$

which is clearly equal to 1 independent of the choice of h. However, computing this expression numerically for $h = 10^{-16}$ we observe

```
0.0
```

```
begin
    t = 1
    h = 1e-16

    ( (t + h) - t ) / h
end
```

which is clearly wrong.

We are faced with this error because the **representation of numbers on a computer has only finite precision**. In fact in standard double-precision accuracy representing the number $1 + 10^{-16}$ **is not possible**. The computer therefore has to **round it to the next representable number**, which is itself 1. Therefore

```
true
```

```
1 + 1e-16 == 1
```

which explains the zero result we obtain.

Structure of floating-point numbers

As we found out above in practice the **set of floating-point numbers** is much smaller than the set of real numbers $\mathbb{R}$ and **not all numbers can be represented as a floating-point numbers**. This leads to the natural question: **which numbers can be represented** on a computer.

Answering this in all details is beyond the scope of this course (see for example https://teaching.matmat.org/error-control/05_Floating_point_arithmetic.html for more information). However, getting some intuition is helpful to obtain a feeling how large floating-point errors can be.

The **set of floating-point numbers** has elements of the form

$$y = \pm m\, \beta^{e-t}.$$

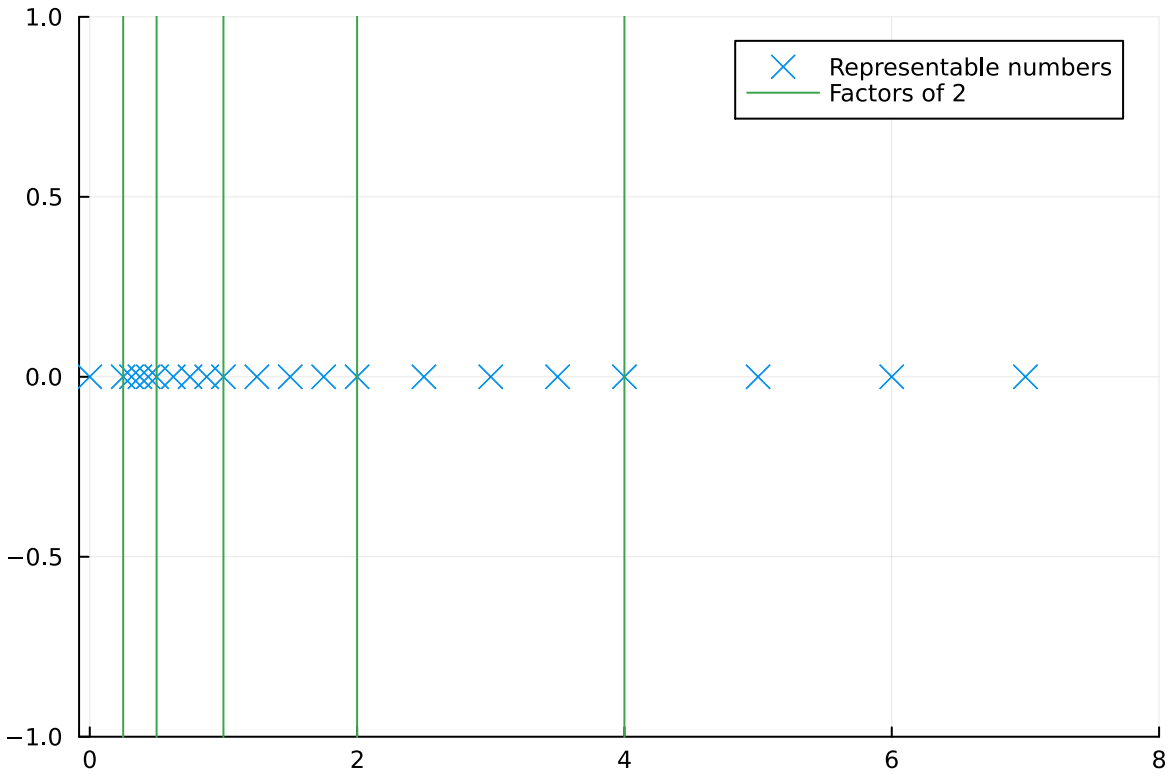
It is thus characterised by four integer parameters:

- the **base** (or radix) β. Almost always binary, i.e. $\beta = 2$,
- the **number of significand digits** t and
- the **exponent range** $e_{\text{min}} \leq e \leq e_{\text{max}}$.

In the definition of y the integer m is the **significand** (or mantissa), which has t number of digits, i.e. $m = (a_1 a_2 \cdots a_t)$, which satisfy $0 < a_1 \leq \beta - 1$ and $0 \leq a_i \leq \beta - 1$ for $i = 2, \ldots, t$.

The most common case are **double-precision floating-point numbers** with $\beta = 2$, $t = 24$, $e_{\min} = -1023$ and $e_{\max} = +1022$.

This is unfortunately too many numbers to illustrate easily. To still get a visual representation of the "denseness" of floating-point numbers, we therefore consider a toy case with $\beta = 2, t = 3$, $e_{\min} = -1$ and $e_{\max} = 3$. Considering only the positive numbers, the following numbers can be represented:



We notice that the **representable numbers** get **less and less dense** as we move to to larger values –- a conclusion which remains true for double-precision floating-point numbers.

This construction has one key advantage, namely that the **relative error due to rounding** remains finite. Or in other words if $fl(x)$ denotes the result of rounding the real number x into the set of representable floating-point numbers, then

Definition: Machine epsilon

The **machine epsilon** $\varepsilon_M = \beta^{1-t}$ is the smallest real number greater than zero, such that $fl(1 + \varepsilon_M) > 1$. The **roundoff error** $\frac{\varepsilon_M}{2}$ is an upper bound to the relative error in representing a real number $x \in \mathbb{R} \setminus \{0\}$, i.e.

$$\frac{|x - fl(x)|}{|x|} \leq \frac{\varepsilon_M}{2}$$

Standard elementary floating-point operations (e.g. addition, subtraction, multiplication, division) have similarly at most a relative error of $\frac{\varepsilon_M}{2}$. E.g. for the case of adding two floating-point numbers x and y we have

$$\frac{|(x+y) - fl(x+y)|}{|x+y|} \leq \frac{\varepsilon_M}{2}.$$

For double-precision numbers we have for the machine epsilon

```
2.220446049250313e-16
```
```
1 eps(Float64)
```

which finally explains

```
true
```
```
1 1 + 1e-16 == 1
```

namely because 10^{-16} is *smaller* than the machine epsilon, such that $fl(1 + 10^{-16}) = 1$.

Takeaway: Properties of double-precision numbers

For standard double-precision floating-point numbers (`Float64`) we have $\varepsilon_M \approx 2 \cdot 10^{-16}$, such that floating-point operations are accurate to a **relative error** of $\frac{\varepsilon_M}{2} \approx 10^{-16}$.

Tip: This is the only information you need to remember about the structure of floating-point numbers for this class.

Errors in algorithms can be much larger than machine epsilon

ust because the relative error of an individual floating-point operation is bounded by $\frac{\varepsilon_M}{2}$, this **does not imply that the error of an algorithm is bounded** by this value. This is exactly what we saw in the numerical differentiation example above, where the observed error in the computed derivative amounts to values well beyond $\frac{\varepsilon_M}{2}$, such that all accuracy in the final answer is lost.

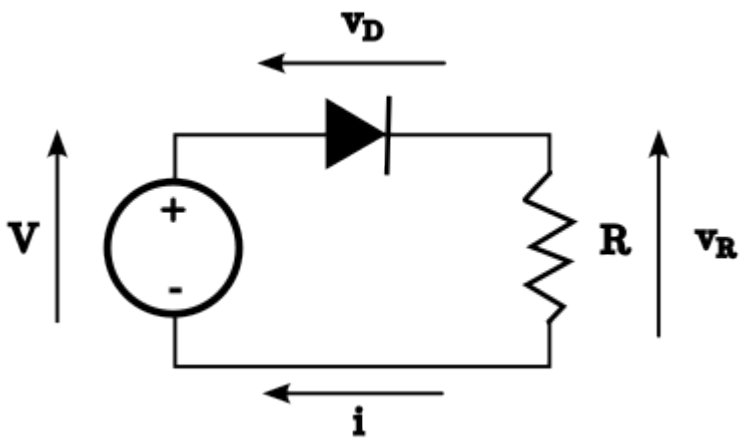
Related questions we will discuss:

- What measures can we take to keep floating-point error small in algorithms, such as the numerical computation of derivatives ?

Fixed-point problems

The previous discussion was mostly centred around the *accuracy* of a computation. In this example we will get some first idea about the challenges when designing a good numerical algorithm.

We consider the simple Diode model circuit



which consists of

- Some voltage source, generating a voltage V
- A resistor R with the linear relationship

  $$v_R = Ri$$

  between the current i and its voltage v_R.
- A diode for which we will take the standard Shockley diode model, i.e. the equation

  $$i = i_0(e^{v_D/v_0} - 1)$$

  between the diode's current i and voltage v_D. v_0 and i_0 are parameters, which characterise the diode.

- According to the circuit laws, the voltages across the circuit elements need to sum, i.e. $V = v_D + v_R$.

We now want to solve this problem numerically, that is **find the Diode voltage v_D** and the **current i** from the parameters R, V, i_0 and v_0.

If you have never encountered circuit theory, just take it for granted that we now have a problem consisting of the three equations

$$V = v_D + v_R \tag{2a}$$
$$v_R = Ri \tag{2b}$$
$$i = i_0(e^{v_D/v_0} - 1) \tag{2c}$$

where v_D is the unknown we want to determine.

Our goal is to model this circuit, i.e. understand the current i and its voltage v_D across the circuit elements.

Method 1

Using equations (2a) and (2b) we can derive the relationship

$$i = \frac{v_R}{R} = \frac{V - v_D}{R} \tag{3}$$

Rearranging (2c) in turn leads to

$$v_D = v_0 \log\left(\frac{i}{i_0} + 1\right).$$

Inserting (3) into this latter equation then gives

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

or by introducing the function

$$g_{\log}(x) = v_0 \log\left(\frac{V - x}{Ri_0} + 1\right). \tag{4}$$

the problem

$$g_{\log}(v_D) = v_D.$$

Problems of this form, where one seeks a point v_D at which the function $g_{\log}$ returns just the point itself are called **fixed-point problems**.

Note that, since f is not just a simple polynomial, but a **non-linear function** there is no explicit analytic formula for finding its solution. We therefore need to consider iterative approaches.

Interestingly enough in this case simple repeated iteration, i.e.

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

just works to find the fixed point. To see that we select the parameters

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

and define the fixed-point map

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

```
function glog(vD)
    v0 * log((V - vD) / (R * i0) + 1)
end
```

and apply this function 10 times:

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

After 10 steps about 12 significant figures have stabilised, so we seem to obtain convergence to a value v_D, i.e. mathematically

$$\lim_{k\to\infty} v_D^{(k)} = v_D.$$

At this point we have $v_D = g_{\log}(v_D)$ (otherwise the iteration would not stabilise). Since

$$v_D = g_{\log}(v_D) = v_0 \log\left(\frac{V - v_D}{R i_0} + 1\right)$$

one can show that this voltag v_D actually satisfies equations (2a) to (2c) simultaneously, i.e. satisfies the diode equation.

Just by repeatedly applying $g_{\log}$ we obtained **a numerical algorithm to solve the diode problem**.

Method 2

Now the arguments to introduce Method 1 were in fact a little convoluted. Let's try to directly balance the voltage across the circuit, i.e. directly employ $V = v_R + v_D$. Then using equations (2a) to (2c) this leads to:

$$V = v_R + v_D = Ri + v_D = Ri_0\left(e^{v_D/v_0} - 1\right) + v_D \tag{5}$$

or

$$V - Ri_0\left(e^{v_D/v_0} - 1\right) = v_D$$

Introducing a function

$$g_{\text{exp}}(x) = V - R\,i_0\left(e^{x/v_0} - 1\right). \tag{6}$$

we again observe a fixed-point problem $g_{\text{exp}}(v_D) = v_D$. We again iterate as before:

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

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

```
let
    vD = 0.1
    for i in 1:10
        vD = gexp(vD) # Call fixed-point map
        @printf "iter %2i:  vD = %.15f\n" i vD  # Print formatted data
    end
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 2 seems equally plausible as method 1** on first sight, **method 2 does not converge**. As a result this method **is not helpful to us to obtain the fixed point / the diode voltage**.

As we will see in the next lectures there are a number of numerical techniques to solve such nonlinear fixed-point problems $g(x) = x$. Each of these have their advantages and disadvantages and hardly any of them just work.

Let us also note already here that **fixed-point problems are closely related to root-finding problems** (*frz. trouver des zéros*). Namely if x is a fixed-point of g than it is a root of $f(x) = g(x) - x$.

Often for a scientific problem there are **many choices how to formulate it as a fixed-point or root-finding problem**. Therefore there are usually **many choices how to solve the same problem on a computer**. As we saw above not all of these options work equally well.

This immediately raises some questions:

Related questions we will discuss:

- How can I understand whether an obtained numerical answer is credible ?
- What techniques based on visualisation or plotting help me to understand the accuracy of a numerical algorithm ?
- How can I understand under which conditions algorithms converge ?
- If they do not converge, how can I overcome numerical issues ?
- What are numerically stable techniques for solving standard scientific problems ?
- How can I understand and evaluate the speed of convergence of an algorithm and improve it even further ?

Even if you now might think, that your heart beats for **experimental research** and you will never need numerics, keep in mind that **all data analysis** uses procedures **based on such techniques**. **Even commercial packages** for experimental post-processing rely heavily on the procedures we will cover and **sometimes get it wrong**. See for example this report on Microsoft Excel). This **can and has compromised scientific fundings in the past**. Therefore it's **best to be prepared**, so you can judge what is wrong: The experiment or the numerics.

Optional: Interpolating data

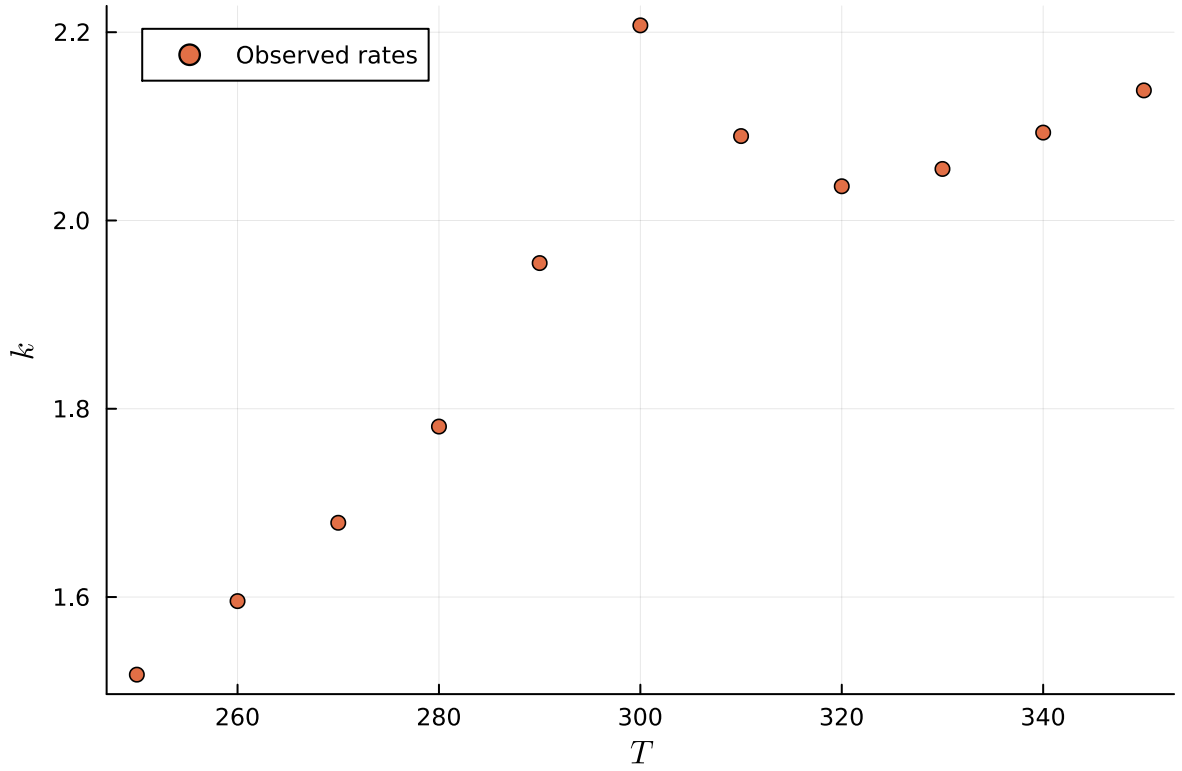
Imagine now we did some measurement of the rate of a chemical reation at different temperatures, let's say

```
data = """
# Temperature(K)  Rate(1/s)
  250.0           1.51756
  260.0           1.59566
  270.0           1.67888
  280.0           1.78110
  290.0           1.95476
  300.0           2.20728
  310.0           2.08967
  320.0           2.03635
  330.0           2.05474
  340.0           2.09338
  350.0           2.13819
""";
```

Now we are curious about the rates at another temperature, say $255K$. The basic laws of thermodynamics and chemical kinetics tell us that the rate should be a *continuous function* of the temperature. For example Arrhenius' law

$$k(T) = A \exp\left(-\frac{E_A}{k_B T}\right) \tag{4}$$

establishes such a relationship, where A, E_A and k_B are some constants. Note, that in this case the behaviour is more complicated as is immediately obvious from a plot of the observed data:



Still, it seems reasonable that we should be able to **determine some continuous function f from the observed data**, which **establishes a good model** for the relationship $k = f(T)$. Albeit $255K$ has not been measured, we can thus evaluate $f(255K)$ and get an estimate for the rate at this temperature. Since the new temperature $255K$ islocated *within* the observed data range (i.e. here $250K$ to $350K$) we call this procedure **interpolation**.

In this lecture we will discuss some common interpolation techniques. In particular we will develop the **mathematical formalism** for these techniques and –- most importantly –- we will use this formalism to understand conditions when these methods work and **when these methods fail**.

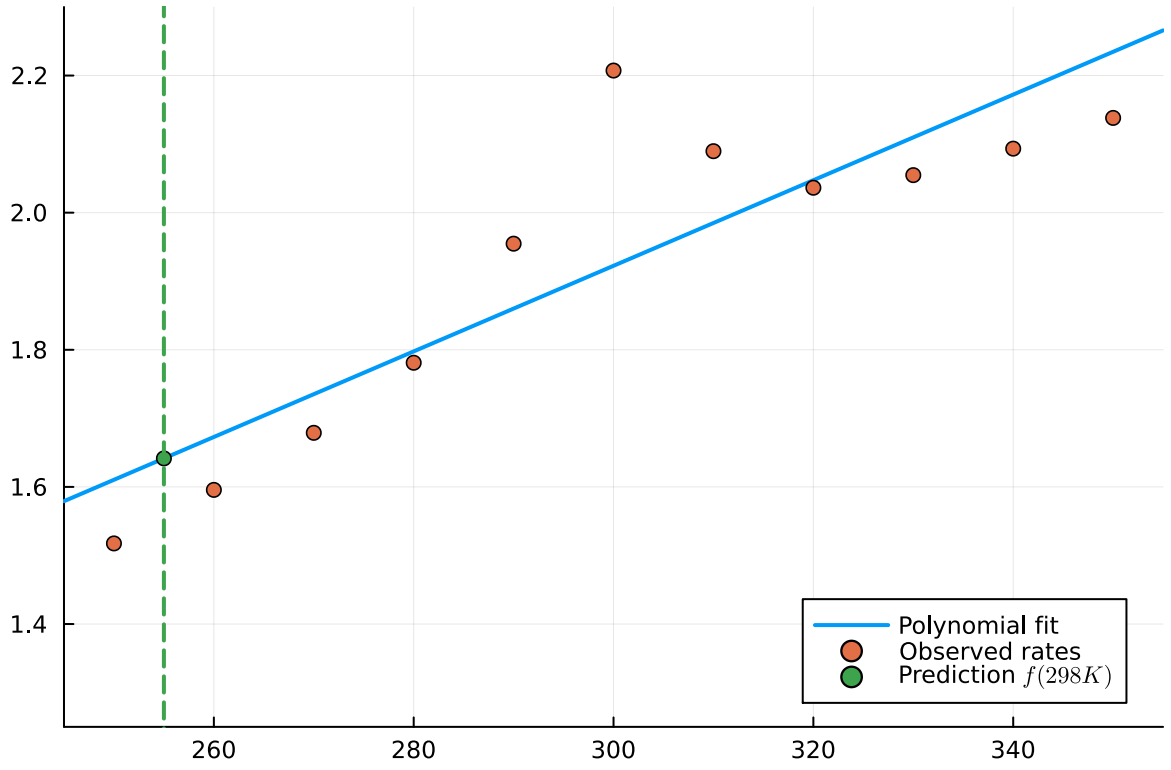
In fact our example here is already a little tricky. Let us illustrate this point for **polynomial interpolation**, one particularly frequent technique. Without going into details for now, one basic idea is to fit a polynomial of degree N

$$k(T) = \sum_{n=1}^{N} \alpha_n T^n$$

to the data. By some procedure discussed later we thus determine the unknown polynomial coefficients α_n, such that the polynomial best matches our observations and use that to determine $f(255K)$.

Let's see the result of such a fit. The slider let's you play with the polynomial order:

- N = 1



Clearly, the quality of such a fit depends strongly on the polynomial order. Perhaps suprising is, however, that **higher degree is not necessarily better**. We will discuss why this is.

Related questions we will discuss:

- How can I fit a parametric model (e.g. the above polynomial) to data points ?

- How accurate can I expect such a polynomial fit to be ?
- Can I choose an optimal polynomial order to obtain the most accurate answer ?
- If I have control over the strategy to acquire data (e.g. how to design my lab experiment), can I have an influence on the accuracy of such interpolations ?

```
(::Main.var"workspace#5".var"#Sidebar#Sidebar##0") (generic function with 1 method)
```

```
let
    RobustLocalResource("https://teaching.matmat.org/numerical-
    analysis/sidebar.md", "sidebar.md")
    Sidebar(toc, ypos) = @htl("""<aside class="plutoui-toc aside indent"
        style='top:$(ypos)px; max-height: calc(100vh - $(ypos)px - 55px);'
        >$toc</aside>""")
    # Sidebar(Markdown.parse(read("sidebar.md", String)), 300)
end
```