Click here to view the PDF version.

```
begin
    using Plots
    using Polynomials
    using PlutoUI
    using PlutoTeachingTools
    using LaTeXStrings
    using LinearAlgebra
    using HypertextLiteral: @htl, @htl_str
end
```

Table of Contents

Interpolation

In this chapter we will return to one of the problems we already briefly discussed in the introductory lecture, namely:

Definition: Interpolation problem

Suppose we are given data (x_i, y_i) with $i = 1, 2, 3, \dots n$, where the x_i are all distinct. Find a function p such that $p(x_i) = y_i$.

The function p is usually called the **interpolant** or **interpolating function**.

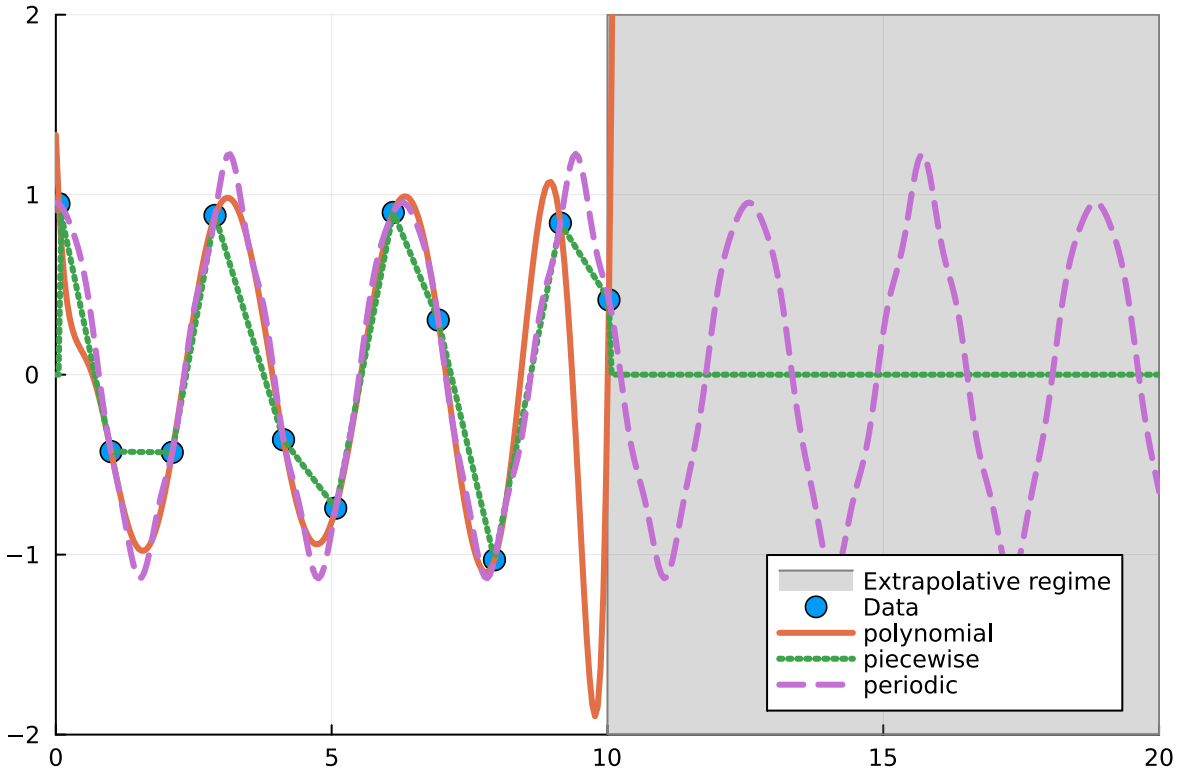
One can also consider the task of finding such an interpolating function p as one simple example for a **data-driven method**:

- Let us assume the data (x_i, y_i) originates from a complex statistical process (e.g. a lab experiment or an involved simulation)
- Having observed n observed data points $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ we thus want to obtain a cheaper **model** to predict a measurement (x_{n+1}, y_{n+1}) for a so far unseed x_{n+1}.
- By the means of interpolation we effectively **train** such a model, namely the interpolant p. Based on the interpolated p we can **make a prediction** of the data point (x_{n+1}, y_{n+1}) as $(x_{n+1}, p(x_{n+1}))$.

▶ **Data-driven methods**

Of course we have not really discussed precisely *how* to obtain such an p. Indeed the interpolation condition $p(x_i) = y_i$ is rather weak and in practice and leaves considerable freedom:

- Show polynomial interpolant: ☑
- Show piecewise linear interpolant: ☑
- Show periodic interpolant: ☑



Which of the above interpolants is the *best* is strongly dependent on the scientific context, for example:

- If we know our data to have some periodicity, than the periodic interpolant (purple) may be better.
- If we need differentiability, the piecewise linear interpolant (green) may not be a good choice.

Therefore, in practical applications the quality of the fit usually depends on how well our method to construct p agrees with the behaviour of the data itself.

The **typical ansatz** for an interpolation problem is to take p from a family of suitable functions (e.g. polynomials, trigonometric functions etc), which form a vector space. If $\varphi_1, \varphi_2, \dots \varphi_n$ is a basis for for this vector space, then we can write p as the linear combination

$$p(x) = \sum_{j=1}^{n} c_j \varphi_j(x) \tag{1}$$

of n basis functions. Employing further the condition that p needs to pass through the data points, i.e. $p(x_i) = y_i$ for all i, then we get n equations

$$p(x_i) = \sum_{j=1}^{n} c_j \varphi_j(x_i) = y_i \qquad i = 1, \dots n$$

which is a system of n equations with n unknowns. In matrix form we can write

$$\mathbf{A}\mathbf{c} = \mathbf{y} \tag{2}$$

where

$$\mathbf{A} = \begin{pmatrix} \varphi_1(x_1) & \dots & \varphi_n(x_1) \\ \varphi_1(x_2) & \dots & \varphi_n(x_2) \\ \vdots & & \\ \varphi_1(x_n) & \dots & \varphi_n(x_n) \end{pmatrix}, \qquad \mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}, \qquad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Note: The number of basis functions and the number of data points does not need to agree. We will consider such more general regression problems later.

Polynomial interpolation

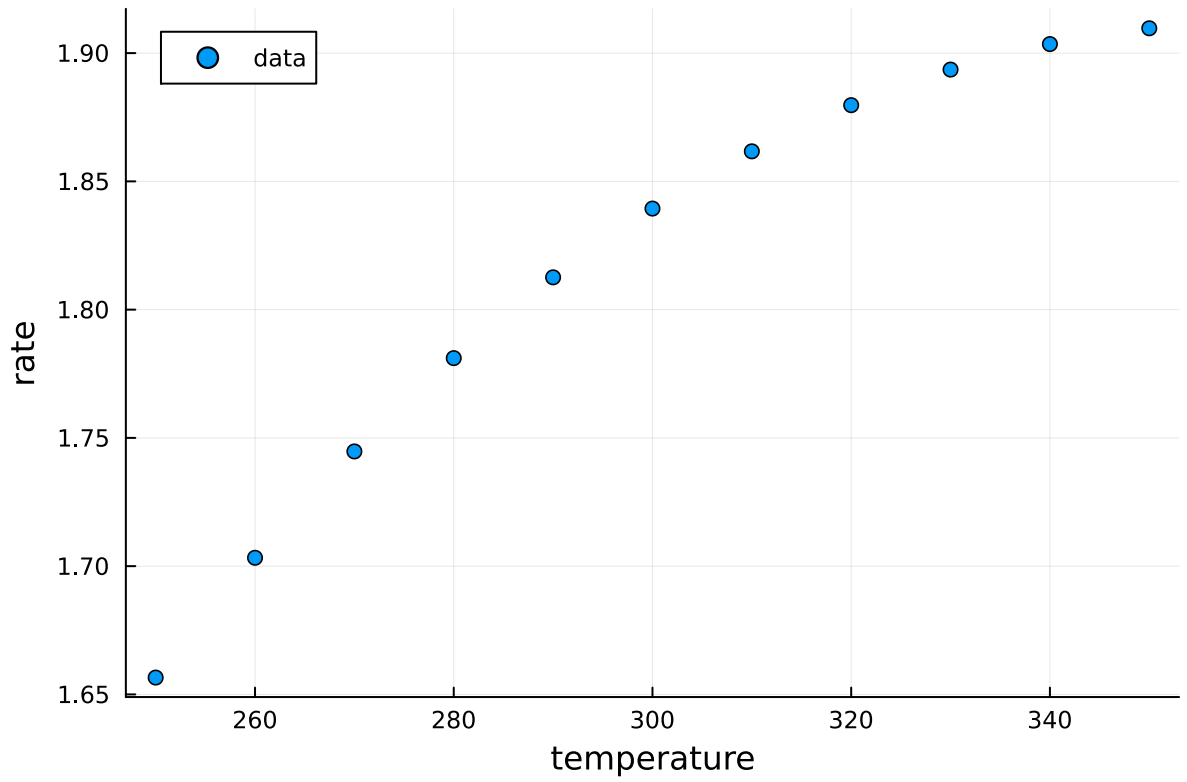
We start with one of the simplest forms of interpolation: Namely fitting polynomials through the given data.

Let us come back to our earlier problem of finding an interpolating function p for the given temperature-dependent reaction rates.

Note, that compared to the pathological example of the introduction, we start with a slightly simpler case:

```
data = """
# Temperature(K)  Rate(1/s)
  250.0           1.65657
  260.0           1.70327
  270.0           1.74472
  280.0           1.78110
  290.0           1.81259
  300.0           1.83940
  310.0           1.86171
  320.0           1.87971
  330.0           1.89358
  340.0           1.90352
  350.0           1.90968
""";
```

```
begin
    lines = split(data, "\n")
    temperature = [parse(Float64, split(line)[1]) for line in lines[2:end-1]]
    rate        = [parse(Float64, split(line)[2]) for line in lines[2:end-1]]
end;
```




```
scatter(temperature, rate; label="data", xlabel="temperature", ylabel="rate")
```

Monomial basis

Let us first consider the case of **polynomial interpolation**: given $n+1$ data points (x_1, y_1) up to (x_{n+1}, y_{n+1}) we want to find a n-th degree polynomial

$$p_n(x) = \sum_{j=0}^{n} c_j x^j, \tag{3}$$

which is an interpolating function, i.e. satisfies $p_n(x_i) = y_i$ for all $i = 1, \ldots, n+1$. Fundamental results from algebra ensure that such a polynomial of degree n, which goes through all $n+1$ data points can always be found. Moreover this polynomial (thus its coefficients c_j) is uniquely defined by the data.

Indexing conventions: Starting at 0 or 1

Note that in the definition of the polynomial (Equation (3)) the sum starts now from $j = 0$ whereas in equation (1) it started from $j = 1$.

When discussing numerical methods (such as here interpolations) it is sometimes more convenient to start indexing from 0 and sometimes to start from 1. Please be aware of this and read sums in this notebook carefully. Occasionally we use color to highlight the start of a sum explicitly.

To find this polynomial a natural idea is thus to employ the monomials $1, x, x^2, \dots x^n$ directly as the basis for our interpolation:

Definition: Monomial basis

The basis of $n + 1$ monomials are the functions $\varphi_1(x) = 1, \varphi_2(x) = x, \dots, \varphi_{n+1}(x) = x^n$.

Employing these in equations (1) and (2) to perform our interpolation leads to the linear system in the $n + 1$ unknowns $c_0, c_1, \dots, c_n$

$$\underbrace{\begin{pmatrix} 1 & x_1 & \dots & x_1^n \\ 1 & x_2 & \dots & x_2^n \\ \vdots & & & \\ 1 & x_{n+1} & \dots & x_{n+1}^n \end{pmatrix}}_{=\mathbf{V}} \underbrace{\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix}}_{\mathbf{c}} = \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{pmatrix}}_{\mathbf{y}}, \tag{4}$$

where the $(n + 1) \times (n + 1)$ matrix $\mathbf{V}$ is called the **Vandermonde matrix**. Assuming the data points (x_i, y_i) to be distinct, we know that only one interpolating polynomial exists. The linear system (4) has therefore exactly one solution and the matrix V is thus always invertible, $\det(\mathbf{V}) \neq 0$.

Let's see how this method performs on our data. We demonstrate the case for `n_data_monomial = 3`, thus a polynomial degree of 2

First we build the Vandermonde matrix:

```
3×3 Matrix{Float64}:
 1.0  250.0  62500.0
 1.0  260.0  67600.0
 1.0  270.0  72900.0
```

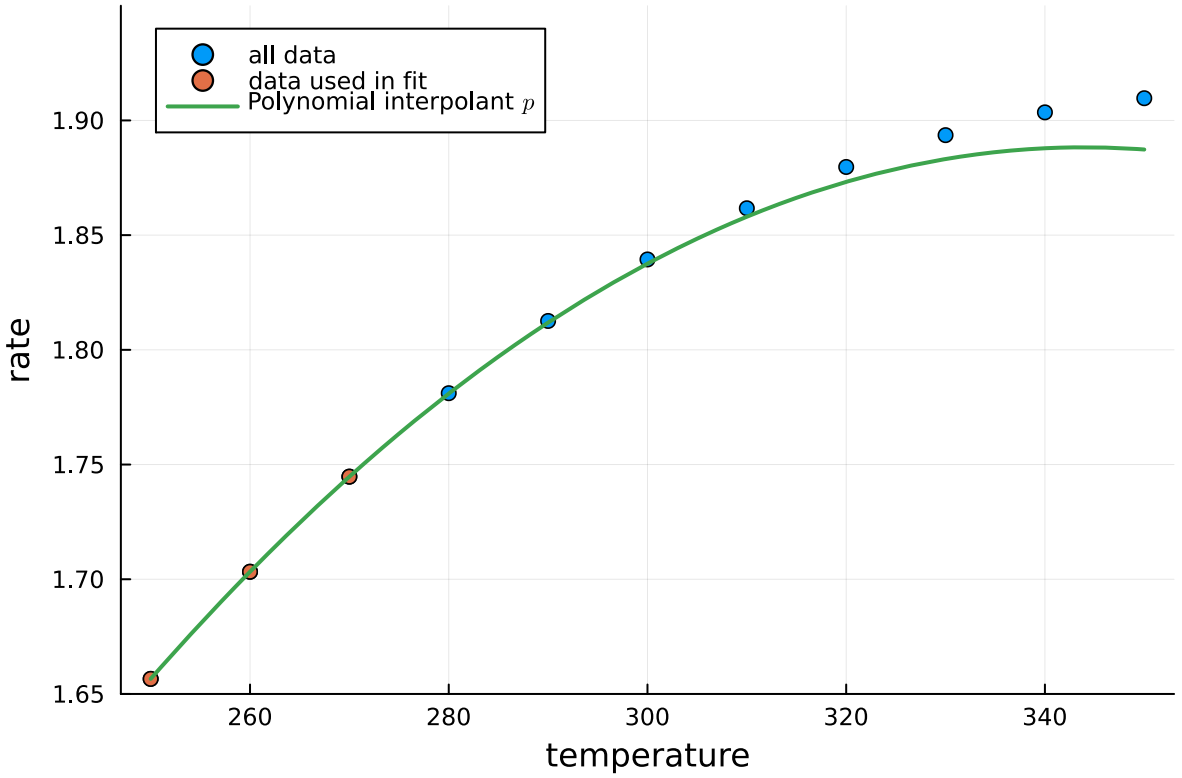
```
begin
    # We wish to find an interpolating polynomial for the mapping
    # temperature to reaction rate.

    x = temperature[1:n_data_monomial]
    y = rate[1:n_data_monomial]

    V = zeros(n_data_monomial, n_data_monomial)
    for j in 1:n_data_monomial
        for i in 1:n_data_monomial
            V[i, j] = x[i] ^ (j-1)
        end
    end
    V
end
```

Then we solve the linear system to find the polynomial coefficients:

```
c = [-1.21718, 0.0180575, -2.625e-5]
```

```
c = V \ y  # Solve V * c = y
```

... and plot the result:



Use the Slider to regulate how many of the data points are included in the interpolation.

- `n_data_monomial =` 3

We see as we use more and more data points, eventually the full data range is well-represented by interpolating polynomial.

A word of warning: Vandermonde matrices are an example of what is known as a *badly conditioned matrix*, that is a matrix where small numerical errors (such as rounding errors due to the finite-precision number format used by the computer) can amplify and lead to very inaccurate solutions. Therefore this method starts to be come very unreliable for n larger than a few tens.

We will explore a better method in the next section.

Lagrange basis

While the monomial basis seemed natural, the fact that the n-th degree interpolating polynomial is *uniquely* determined by the $n+1$ data points allows us to use *any polynomial basis* to find it.

Both a more practical as well as a numerically more stable way to find **the** interpolating polynomial is the approach using Lagrange polynomials.

Definition: Lagrange basis

The Lagrange polynomials associated to the **nodes** $x_1, x_2, \dots x_{n+1}$ are the $n+1$ polynomials

$$\begin{aligned} L_i(x) &= \prod_{\substack{j=1 \\ j \neq i}}^{n+1} \frac{x - x_j}{x_i - x_j} \\ &= \frac{(x-x_1)(x-x_2)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_{n+1})}{(x_i-x_1)(x_i-x_2)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_{n+1})} \end{aligned} \tag{5}$$

for $i = 1, \dots, n+1$.

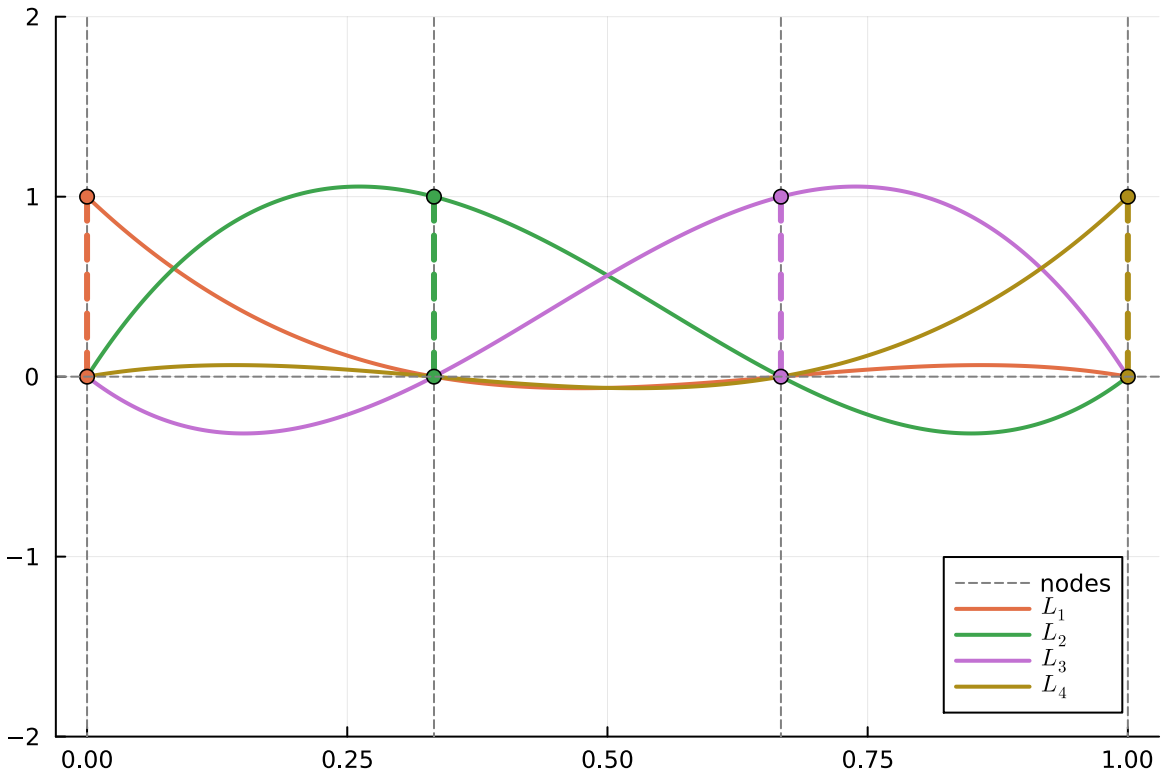
Each of these polynomials is of degree n, moreover they satisfy the **cardinality condition**:

$$L_j(x_i) = \delta_{ji} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{6}$$

i.e. they are 1 only on the nodal point with the same index, but 0 on all other nodal points.

Visualisation of Lagrange polynomials on an equally spaced grid between 0 and 1:

- Number of nodal points: 4



In particular the nodal property (6) makes it extremely convenient to use a Lagrange polynomial basis to find the interpolating polynomial p_n:

Proposition 1

The n-th degree polynomial p_n interpolating the data (x_i, y_i) for $i = 1, \ldots, n+1$ is given by

$$p_n(x) = \sum_{i=1}^{n+1} y_i L_i(x) \tag{7}$$

Proof: This can be easily verified:

- Since every linear combination of an n-th degree polynomial is itself an n-th degree polynomial, p_n (as a linear combination of the n-th degree Lagrange polynomials) is a n-th degree polynomial.
- Moreover

$$p_n(x_i) = y_1 \underbrace{L_1(x_i)}_{=0} + y_2 \underbrace{L_2(x_i)}_{=0} + \cdots + y_i \underbrace{L_i(x_i)}_{=1} + \cdots + y_{n+1} \underbrace{L_{n+1}(x_i)}_{=0} = y_i$$

which confirms that p_n is the interpolating polynomial.

Example: Using Lagrange polynomials

Using Lagrange basis functions, find the interpolating polynomial through the points $(x_1 = -1, y_1 = -6)$, $(x_2 = 1, y_2 = 0)$, $(x_3 = 2, y_3 = 6)$:

- Using equation (7) we obtain

$$p_2(x) = y_1 L_1(x) + y_2 L_2(x) + y_3 L_3(x) = -6L_1(x) + 6L_3(x)$$

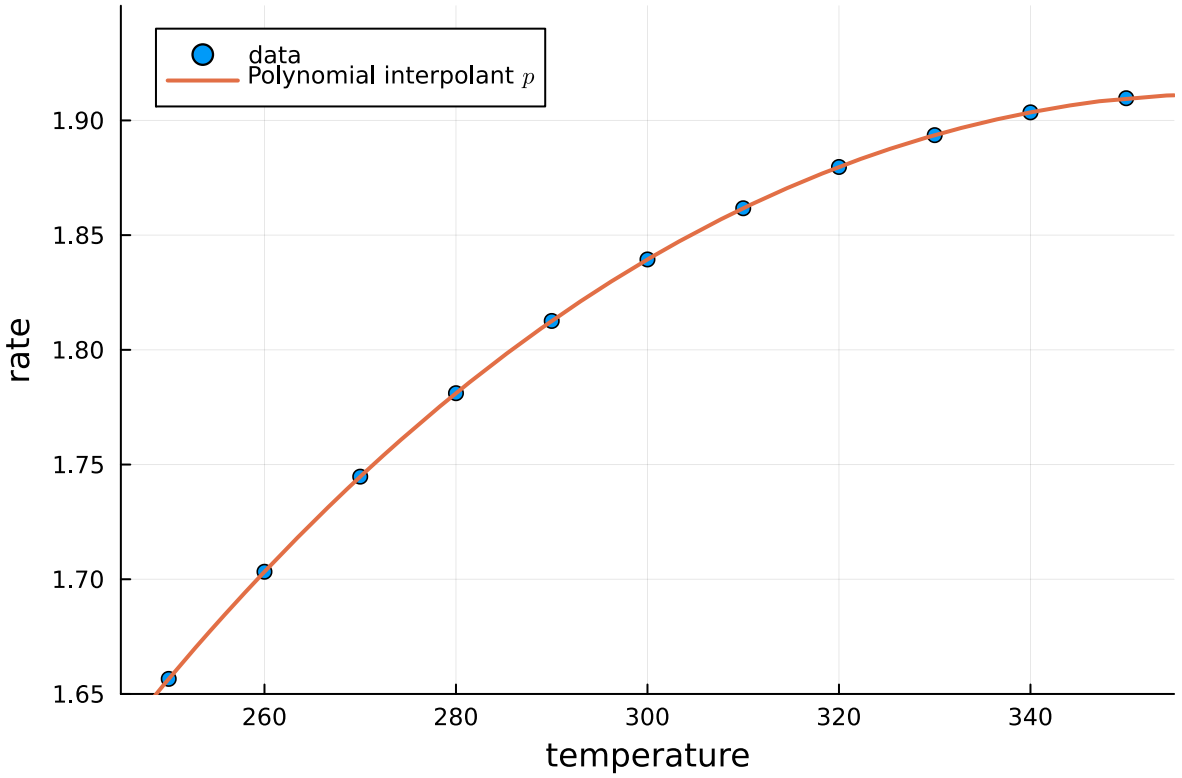
where

$$L_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} = \frac{(x-1)(x-2)}{(-1-1)(-1-2)} = \frac{(x-1)(x-2)}{6}$$
$$L_3(x) = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} = \frac{(x+1)(x-1)}{(2+1)(2-1)} = \frac{(x+1)(x-1)}{3}$$

such that

$$p_2(x) = -6\frac{(x-1)(x-2)}{6} + 6\frac{(x+1)(x-1)}{3} = -4 + 3x + x^2.$$

For reference: In Julia a convenient way to interpolate a polynomial to a given set of points is provided by the `Polynomials` package, e.g.




```
let
    poly = Polynomials.fit(temperature, rate)
    p = scatter(temperature, rate; ylims=(1.65, 1.95), xlims=(245, 355),
                label="data", xlabel="temperature", ylabel="rate")
    plot!(p, poly; label=L"Polynomial interpolant $p$", lw=2, xlims=(245, 355))
end
```

Error analysis

With the Lagrange polynomials we have a simple approach to find an interpolating polynomial. From a numerical analysis perspective this raises the question **how good a polynomial approximation is**. Or to put it into the language of **data-driven methods**: Having obtained an interpolation model, how well does it generalise to unseen x_n ?

To study this mathematically we consider the following setup: Assume we have a true function f and we are allowed to observe n samples $(x_n, f(x_n))$ from it. We want to understand the following questions:

- **Would the polynomial approximation converge** to f as we observe more and more samples of f?
- **How fast** is this convergence ?
- **What is our error** if we only observe n samples $(x_n, f(x_n))$ and are not given any more information ?

In a mathematical language, we want to check whether

$$p_n \to f \quad \text{as} \quad n \to \infty.$$

Provided this convergence is indeed the case, we are usually also interested in the **convergence rate** — similar to our discussion about fixed-point algorithms. Clearly, the faster p_n converges f the better, since we need **less samples** to obtain an **accurate reconstruction** of the original f.

A standard metric to measure how good the approximation of p_n to f is, is to check the largest deviation between the differences of function vaules on the domain of our data. Assuming we want to approximate f on the interval $[a, b]$ we thus compute

$$\max_{x \in [a,b]} |f(x) - p_n(x)| = \|f - p_n\|_\infty,$$

which is the so-called **infinity norm** of the difference $f - p_n$. More generally the **infinity norm** $\|\phi\|_\infty$ for a function $\phi : D \to \mathbb{R}$ is the expression

$$\|\phi\|_\infty = \max_{x \in D} |\phi(x)|,$$

i.e. the maximal absolute value the function takes over its input domain D.

Note that the error $\|f - p_n\|_\infty$ effectively **measures** how well our **polynomial interpolation model p_n generalises to unseen datapoints** $(x_{n+1}, f(x_{n+1}))$ with $x \in D$: If this error $\|f - p_n\|_\infty$ is small, p_n is a very good model for f. If this error is large, it is a rather inaccurate model.

We **return to the interpolation problem**. For illustration in this section we contrast two cases, namely the construction of a polynomial interpolation of the functions

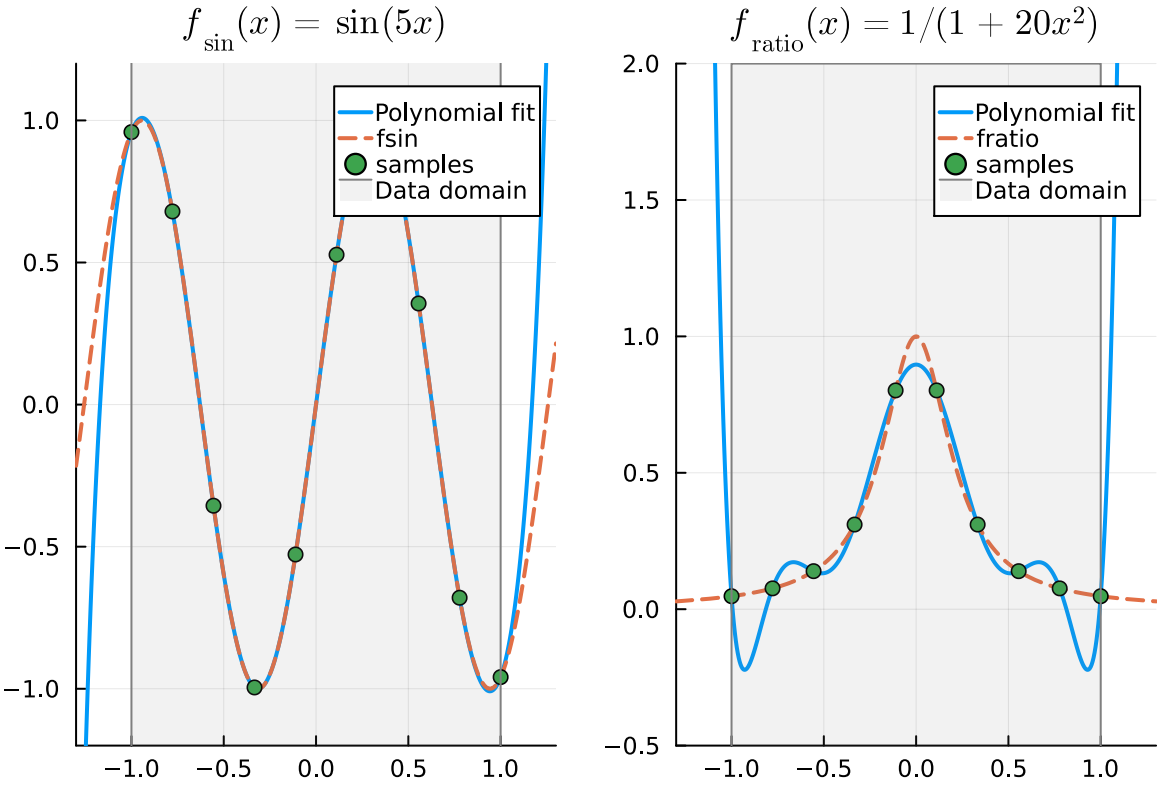
```
fratio (generic function with 1 method)
```

```
begin
    fsin(x)   = sin(5x)
    fratio(x) = 1 / (1 + 20x^2)
end
```

each defined on [-1, 1].

Change the number of samples using the slider and observe the convergence:

- `n_samples_comparison =` 10



We see that the polynomial interpolation of `fsin` (left) converges very well to the original function. Moreover this convergence is very quickly, since already for `n_samples_comparsion = 10` hardly any difference is visible. On the other hand `fratio`, i.e the rather innocent looking function

$$f_{\text{ratio}}(x) = \frac{1}{1 + 20x^2}$$

converges very poorly. In particular from about `n_samples_comparsion = 8` spurious oscillations start to appear towards the end of the domain $[-1, 1]$, which become more and more pronounced as we *increase* the number of samples and the polynomial degree.

To understand this behaviour the following error estimate is useful:

Theorem 2

For a $(n+1)$-times differentiable function $f : [a, b] \to \mathbb{R}$ and $a = x_1 < x_2 < \cdots < x_{n+1} = b$ *equally distributed* nodes in $[a, b]$ the n-th degree polynomial interpolant p_n of the data $(x_i, f(x_i))$ with $i = 1, 2, \ldots, n+1$ satisfies the estimate

$$\|f - p_n\|_\infty \leq \frac{1}{4(n+1)} \left(\frac{b-a}{n}\right)^{n+1} \left\|f^{(n+1)}\right\|_\infty \tag{8}$$

where the **infinity norm** $\|\phi\|_\infty$ for a function $\phi : D \to \mathbb{R}$ mapping from a domain D is the expression

$$\|\phi\|_\infty = \max_{x \in D} |\phi(x)|,$$

i.e. the maximal absolute value the function takes.

The **key conclusion of the previous theorem** is that if the right-hand side (RHS) of (8) goes to zero, than the the error $\|f - p_n\|_\infty$ neccessirly vanishes as n increases.

So let's check this for our functions.

- For $f_{\sin}(x) = \sin(5x)$ we can easily verify $|f_{\sin}^{(n+1)}(x)| = 5|f_{\sin}^{(n)}(x)|$ as well as $\max_{[-1,1]} |f_{\sin}(x)| = \max_{x \in [-1,1]} |\sin(5x)| = 1$, such that

  $$\|f_{\sin}^{(n+1)}\|_\infty = \max_{x \in [-1,1]} |f_{\sin}^{(n+1)}(x)| = 5^{n+1}$$

  and (8) becomes (using $b = 1$ and $a = -1$):

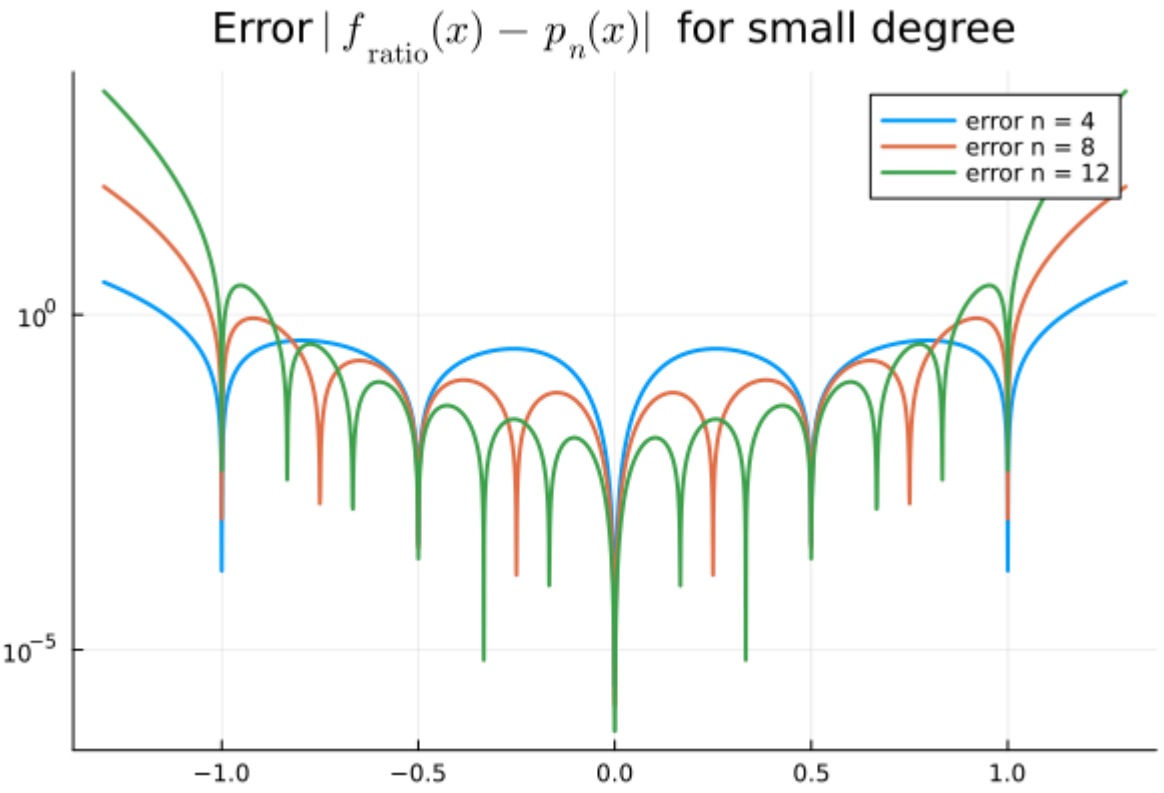
  $$\|f_{\sin} - p_n\|_\infty \leq \frac{10^{n+1}}{4(n+1)n^{n+1}} \longrightarrow 0 \qquad \text{as } n \to \infty$$

- In contrast for $f_{\text{ratio}}(x)$ one can show

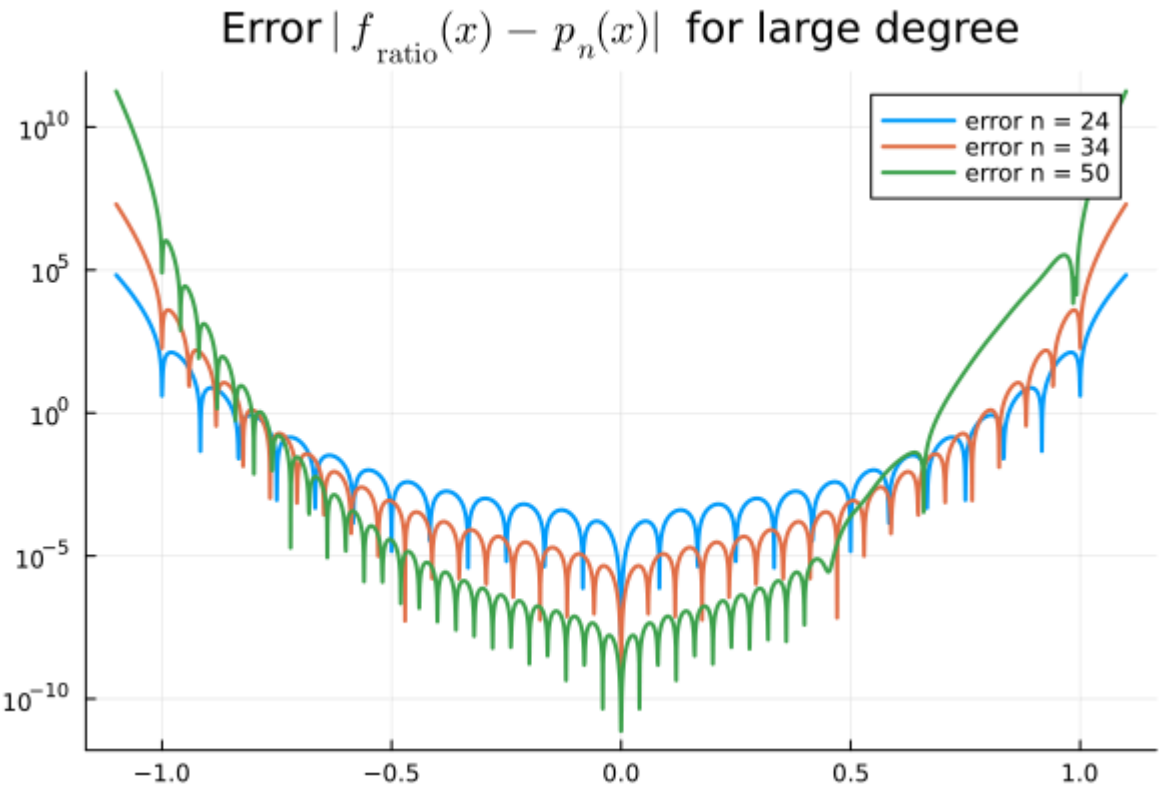
  $$\|f_{\text{ratio}}^{(n+1)}\|_\infty \sim 20^n n! \qquad \text{as } n \to \infty$$

  and as a result convergence is not guaranteed.

Looking more closely at the error of the problematic case, that is plotting $|f_{\text{ratio}}(x) - p_n(x)|$ as a function of x, we observe an interesting pattern:



For small polynomial degrees, the error decreases homogeneously as we increase n. Additionally we observe some almost vertical "drops", where the error goes down to machine precision (about 10^{-16}). This is because at the nodal points — by construction — the polynomial is exact and only the floating-point, error remains.



Switching to higher degrees we observe the error to again *increase* at near the boundaries of the interval $[-1, 1]$. However in the central part of the interpolation domain $[-0.5, 0.5]$ the error remains constantly small. This visual result is also confirmed by a more detailed analysis, which reveals, that the origin is our choice of a *regular* spacing between the sampling points, an effect known as Runge's phaenomenon.

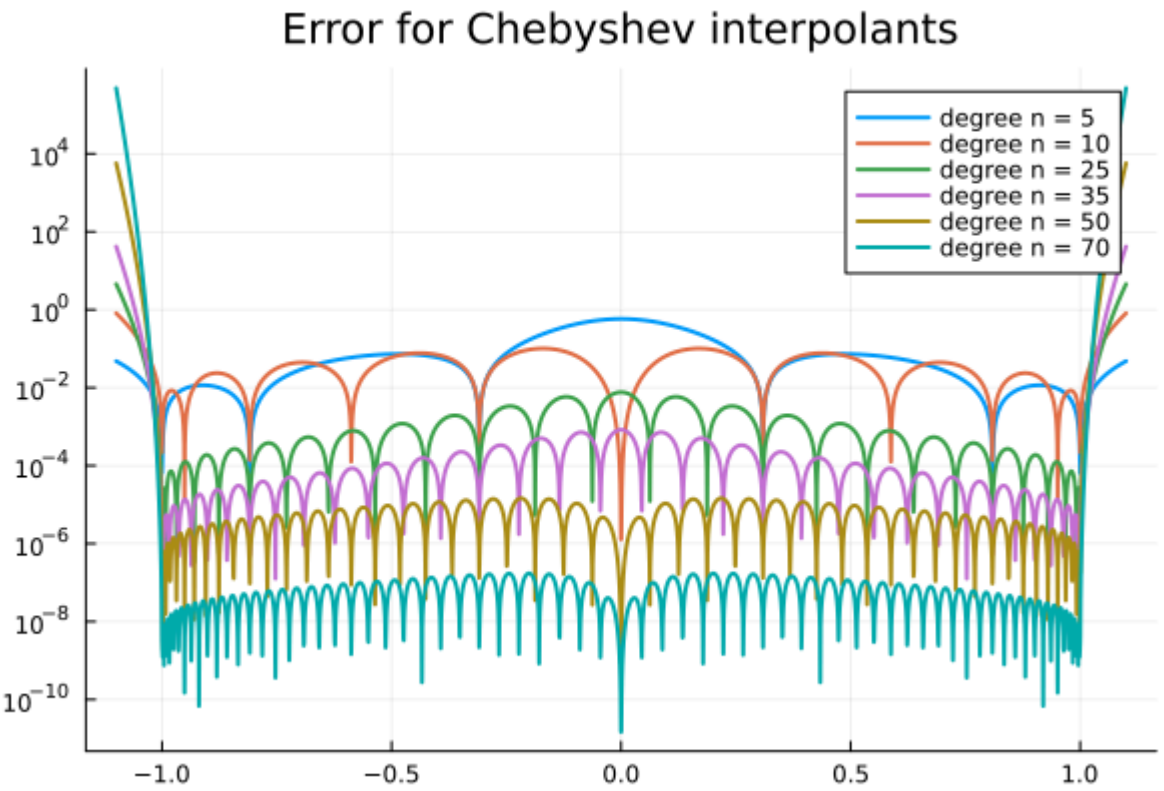
Observation: Runge's phaenomenon

Polynomial interpolation employing **equally spaced nodal points** to construct the interpolant p_n **may lead to a non-convergence** as $n \to \infty$. Moreover while for small n the error in the infinity norm $\| \cdot \|_\infty$ still decreases, for large n this behaviour can change, such that **for large n the error can keep increasing** as n increases.

The solution to this dilemma is to employ **irregularly spaced** nodal points, in particular the nodal points need to become *more densely spaced* towards the boundary of the domain. One especially important node family are the **Chebyshev extreme points** defined by

$$x_k = -\cos\left(\frac{k\pi}{n}\right) \qquad \text{for } k = 0, 1, \dots n.$$

Using these to interpolate a degree n polynomial gives a uniform convergence behaviour as $n \to \infty$:



Notably Chebyshev nodes enjoy the following convergence result:

Theorem 3

Let $f : [-1, 1] \to \mathbb{R}$ be a function, which is analytic in an open interval containing $[-1, 1]$, that is the Talyor series of f converges to $f(x)$ for any x from this open interval. Then we can find constants $\alpha > 0$ and $0 < C < 1$ such that

$$\|f - p_n\|_\infty = \max_{x \in [-1,1]} |f(x) - p_n(x)| \leq \alpha C^n$$

where p_n is the unique polynomial of degree n defined by interpolation on $n + 1$ Chebyshev points.

This is an example of **exponential convergence**: The error of the approximation scheme reduces by a *constant factor* whenever the polynomial degree n is increased by a constant increment.

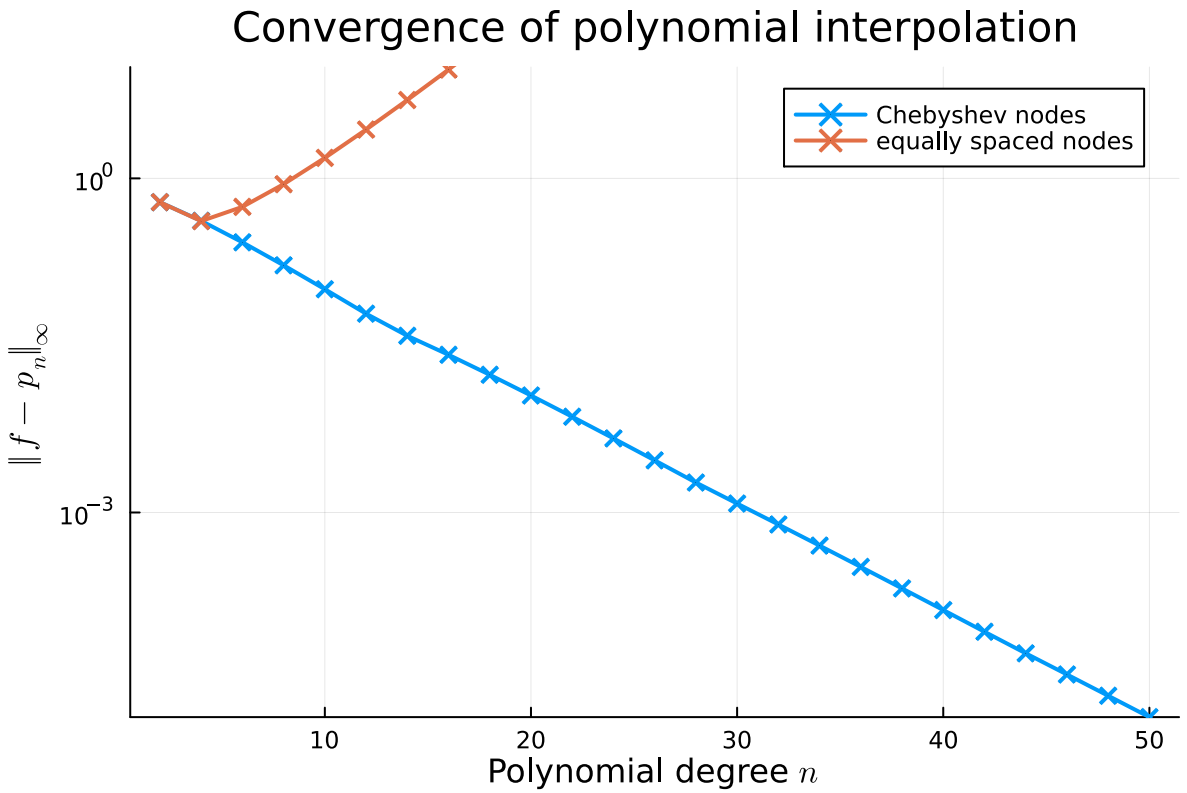
The **graphical characterisation** is similar to the iterative schemes we discussed in the previous chapter: We employ a **semilog plot** (using a linear scale for n and a logarithmic scale for the error),

where exponential convergence is characterised by a straight line:

⚠ TODO ⚠

'previous chapter' remark likely outdated after pushing interpolation back

```
1 TODO("'previous chapter' remark likely outdated after pushing interpolation back")
```



When designing approximation schemes, **obtaining exponential convergence** is one of the **desired properties**.

Observations

- For **exponential convergence**, the error reduces by a constant factor
- A straight line is obtained when looking at the error norm on a `log`-scale versus an appropriate accuracy parameter (such as the polynomial degree n, the spacing of the interpolating notes etc.)

Potential confusion: Convergence terminology

When discussing convergences rates of iterative numerical algorithms and the accuracy of numerical approximation schemes (interpolation, differentiation, integration, discretisation) unfortunately a different terminology is employed. In the following let $\alpha > 0$ and $0 < C < 1$ denote appropriate constants.

- **Iterative schemes:** Linear convergence
  - If the error scales as αC^n where n is the iteration number, we say the scheme has **linear convergence**. (Compare to the last chapter.)
- **Approximation schemes:** Exponential convergence
  - If the error scales as αC^n where n is some accuracy parameter (with larger n giving more accurate results), then we say the scheme has **exponential convergence**.

⚠ TODO ⚠

'Last chapter' reference is likely outdated after pushing interpolation back

```
TODO("'Last chapter' reference is likely outdated after pushing interpolation back")
```

Stability of polynomial interpolation

In the previous discussion we identified the Chebyshev nodal points to provide exponential convergence by avoiding Runge's phaenomenon. Another common question to ask in numerical analysis is referred to as **numerical stability**. The goal of stability analysis is to quantify by how much small perturbations in the input data translate to the obtained result of a numerical algorithm.

We consider the case of interpolating a polynomial p_n to $n+1$ data points (x_i, y_i) for $i = 1, 2, \dots, n+1$, where $a = x_1 < x_2 < \cdots < x_{n+1} = b$ and the y_i are drawn from a function f, i.e. $y_i = f(x_i)$. Our goal is effectively to recover f as close as possible. However, in many practical settings we don't have access to the true data (x_i, y_i), since we are only able to obtain data through a noisy measurement. This means that the data we are *actually* able to collect is much rather $\tilde{y}_i = f(x_i) + \varepsilon_i$ where ε_i represents the measurement error. We suppose further that $|\varepsilon_i| \leq \varepsilon$ for all $i = 1, 2, \dots, n+1$ where $\varepsilon > 0$ is a small number, i.e. that overall the errors are small. Instead of producing an interpolating polynomial p_n using the exact data (x_i, y_i), our procedure only has access to the noisy data $(x_i, \tilde{y}_i)$, thus producing the polynomial $\tilde{p}_n$.

In **stabilty analysis** we now ask the question: How different are p_n and $\tilde{p}_n$ given a measurement noise of order ε.

Let us investigate this using the Lagrange basis, where

$$p_n(x) = \sum_{i=1}^{n+1} y_i L_i(x) \qquad \tilde{p}_n(x) = \sum_{i=1}^{n+1} (y_i + \varepsilon_i) L_i(x)$$

therefore for all $x \in [a, b]$:

$$|\tilde{p}_n(x) - p_n(x)| = \left| \sum_{i=1}^{n+1} \varepsilon_i L_i(x) \right| \leq \sum_{i=1}^{n+1} |\varepsilon_i L_i(x)| \leq \varepsilon \sum_{i=1}^{n+1} |L_i(x)|$$

Introducing

Definition: Lebesgue's constant

Given n nodes $a = x_1 < x_2 < \cdots < x_{n+1} = b$ from the interval $[a, b]$ we denote Lebesgue's constant as the quantity

$$\Lambda_n = \max_{x \in [a,b]} \sum_{i=1}^{n+1} |L_i(x)|.$$

we can rewrite this as

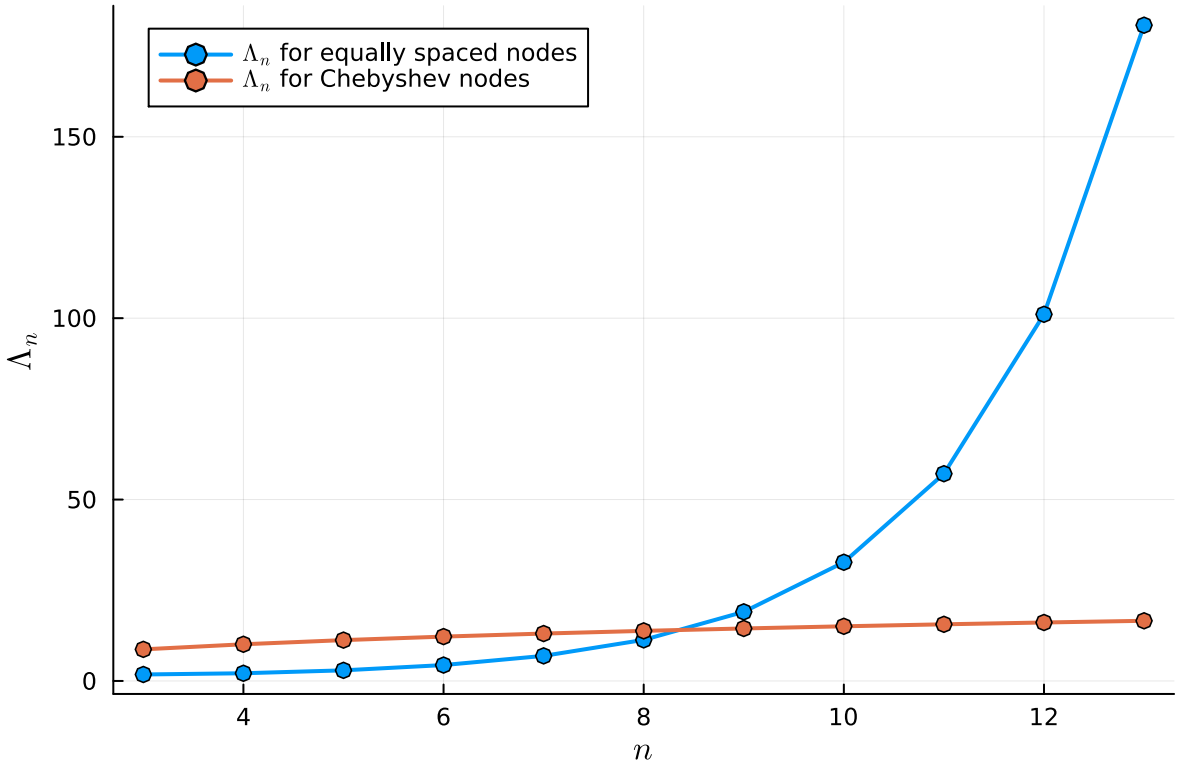
$$|\tilde{p}_n(x) - p_n(x)| \leq \Lambda_n \varepsilon. \tag{9}$$

Notably, if Λ_n is small, then small measurement errors ε can only lead to small perturbations in the interpolating polynomial. In that case our polynomial interpolation procedure would be called **stable** or **well-conditioned**. By contrast, if Λ_n is very high, then already a small measurement error ε allows for notably deviations in the resulting interpolant — we are faced with a **badly conditioned** problem.

Considering the two polynomial interpolation schemes we discussed, one can show

- Equally distributed nodes: $\Lambda_n \sim \frac{2^{n+1}}{e\, n \log n}$ as $n \to \infty$
- Chebyshev nodes: $\frac{2}{\pi}\log(n+1) + a < \Lambda_n < \frac{2}{\pi}\log(n+1) + 1, \qquad a = 0.9625\ldots$

As a graphical visualisation:



Therefore for **Chebyshev nodes** the **condition number** grows only **logarithmically** with n, while for **equally spaced nodes** it grows **exponentially** !

Thus **Chebyshev nodes** do **not only** lead to **faster-converging polynomial interpolations**, but also to notably **more stable answers**. As a result they are one of the standard ingredients in many numerical algorithms.

General principle: Condition number

For numerical problems the factor relating the error in the output quantity — here $|\tilde{p}_n(x) - p_n(x)|$ — to the error in the input quantity — here $\varepsilon = \max_i |\tilde{x}_i - x_i|$ — the **condition number** of the problem. For polynomial interpolation the condition number is exactly Lebesgue's constant Λ_n.

Since for Chebyshev nodes Λ_n stays relatitvely small, we would call Chebyshev interpolation **well-conditioned**. In contrast interpolation using equally spaced nodes is **ill-conditioned** as the condition number Λ_n can get very large, thus **even small input errors can amplify** and **drastically reduce the accuracy** of the obtained polynomial.

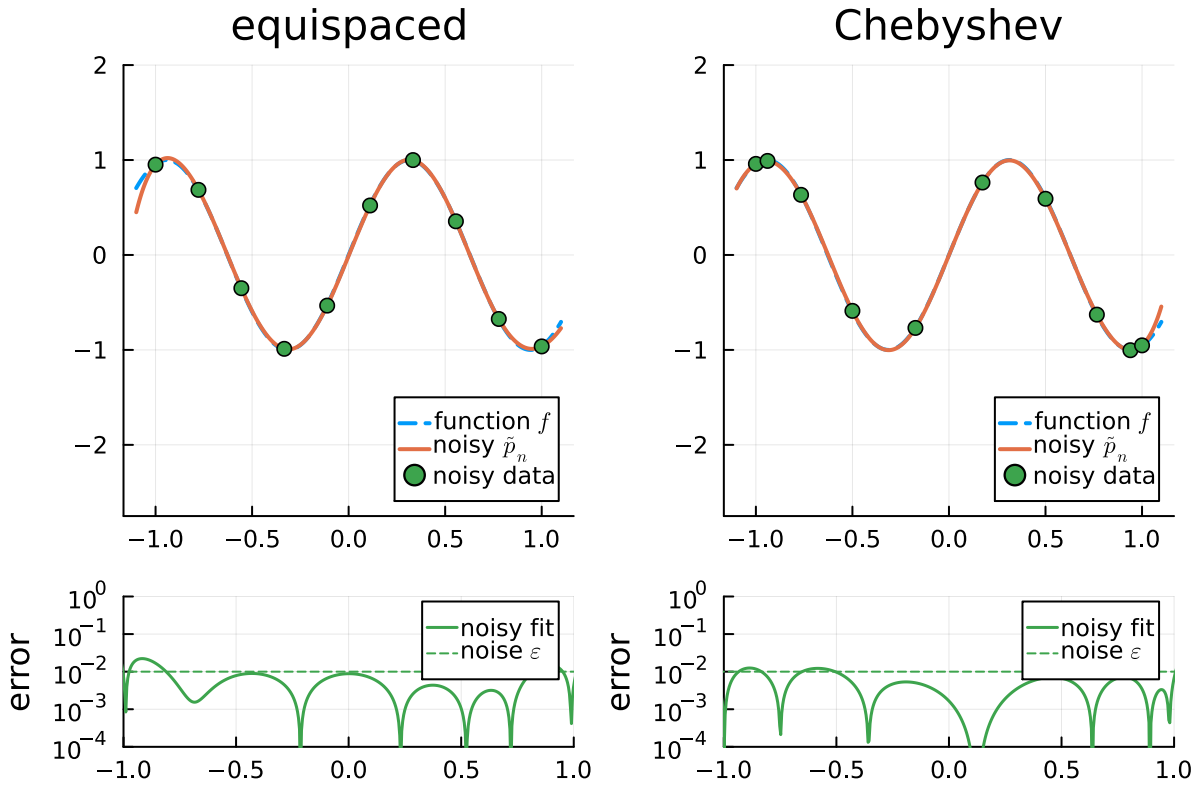
We will meet other condition numbers later in the lecture, e.g. in Iterative methods for linear systems.

Let us **illustrate this result grapically**. We consider again our function $f_{\sin}(x) = \sin(5x)$ in the interval $[-1, 1]$, which we evaluate at the distinct nodes $\{x_i\}_{i=1}^{\texttt{n\_nodes\_poly}}$ — either equally spaced (left plot) or using Chebyshev nodes (right plot). Additional for both cases we consider an exact evaluation, i.e. points $(x_i, y_i) = (x_i, f(x_i))$ as well as noisy evaluations $(x_i, \tilde{y}_i)$ with

$$\tilde{y}_i = f(x_i) + \varepsilon_i$$

where ε_i is a random number of magnitude $|\varepsilon_i| \leq \texttt{ε\_poly}$.

- Number of nodes `n_nodes_poly` = 10
- Noise amplitude `ε_poly` = 0.010000000000000002



As we increase polynomial order and noise, we see larger discrepancies for the interpolation based on equispaced points then for the Chebyshev points.

Piecewise linear interpolation

Note: We will only discuss the high-level ideas of this part in the lecture. You can expect that there will not be any detailed exam questions on Jacobi and Gauss-Seidel without providing you with the

formulas and algorithms.

In the previous section we looked at n-th degree polynomial interpolation, which we found to be poorly conditioned, e.g. when equispaced nodes and high polynomial degree are employed. An alternative construction is to employ piecewise polynomials, i.e. to interpolate a separate polynomial betwen each data point. The most simple approach in this regard is *linear interpolation*, i.e. just connecting the dots of each data point by a straight line.

Definition: Piecewise linear interpolation

Given nodes $x_1 < x_2 < \cdots < x_{n+1}$ and associated data (x_i, y_i) for $i = 1, \ldots, n+1$ the **piecewise linear interpolant** $p_{1,h}$ is given by

$$p_{1,h}(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) \qquad \text{for } x \in [x_i, x_{i+1}]$$

Instead of using this definition to implement piecewise polynomial interpolation, a more practical approach is to follow the idea of equation (1) and construct an appropriate set of basis functions for piecewise polynomial interpolation. These are the

Definition: Hat function

Given nodes $x_1 < x_2 < \cdots < x_{n+1}$ their associated hat functions are the functions

$$H_i(x) = \begin{cases} \frac{x - x_{i-1}}{x_i - x_{i-1}} & \text{if } i > 1 \text{ and } x \in [x_{i-1}, x_i] \\ \frac{x_{i+1} - x}{x_{i+1} - x_i} & \text{if } i \leq n \text{ and } x \in [x_i, x_{i+1}] \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

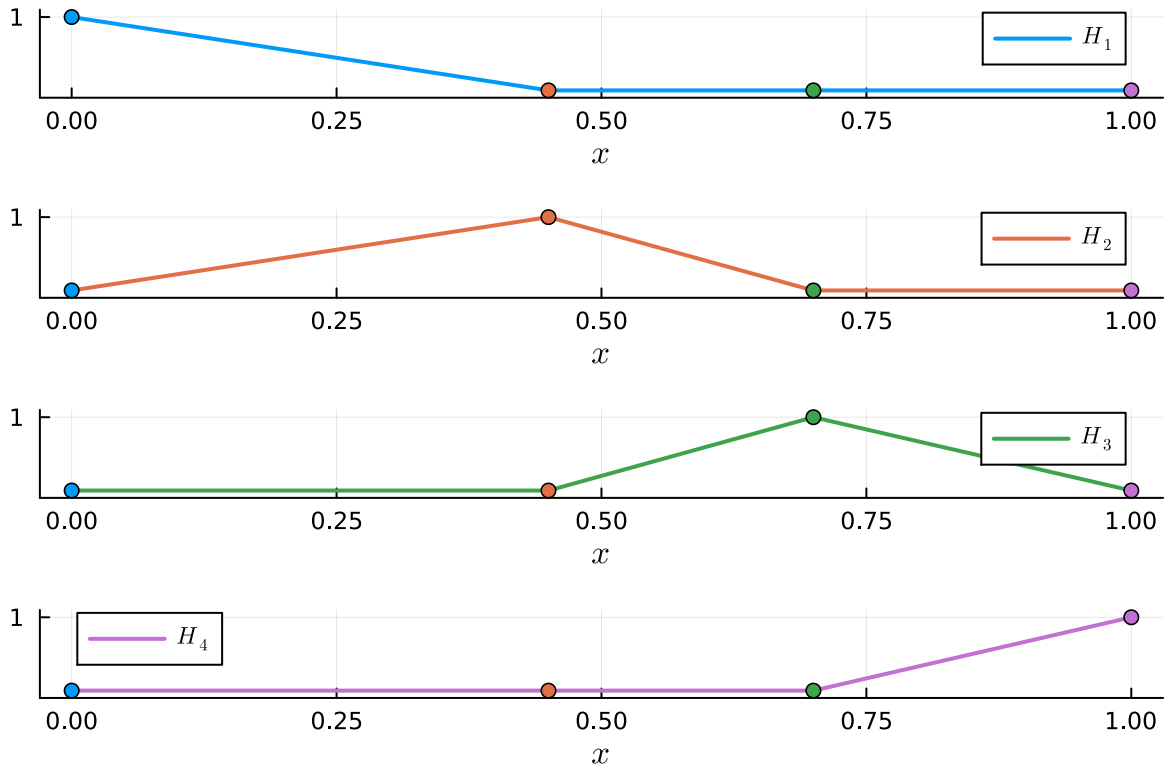
for $i = 1, \ldots, n+1$.

In code these may be obtained as:

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

```
function hatfun(nodes, i)
    # Function to generate the i-th hat function corresponding to nodal
    # points given in the vector nodes. It is assumed that nodes is sorted
    n = length(nodes) - 1

    # Define an inner function, which is returned
    return function (x)
        if i > 1 && nodes[i-1] ≤ x ≤ nodes[i]
            return (x - nodes[i-1]) / (nodes[i] - nodes[i-1])
        elseif i ≤ n && nodes[i] ≤ x ≤ nodes[i+1]
            return (nodes[i+1] - x) / (nodes[i+1] - nodes[i])
        else
            return 0
        end
    end
end
```

We plot the four hat functions for the example nodal points $x_1 = 0$, $x_2 = 0.45$, $x_3 = 0.7$, $x_4 = 1.0$:




```
let
    nodes = [0.0, 0.45, 0.7, 1.0]

    p = plot(; layout=(4, 1), xlabel=L"x", ylims=[-0.1, 1.1], ytick=[1.0])
    for i in 1:length(nodes)
        Hᵢ = hatfun(nodes, i)
        plot!(p, Hᵢ, 0, 1; subplot=i, label=LaTeXString("\$H_$i\$"), lw=2, c=i)
        for (ii, node) in enumerate(nodes)
            scatter!(p, [node], [Hᵢ(node)]; subplot=i, c=ii, label="")
        end
    end
    p
end
```

Since each function H_i is globally continuous and moreover linear inside every interval $[x_i, x_{i+1}]$, any linear combination $\sum_{i=1}^{n+1} c_i H_i(x)$ of hat functions will have the same property. Conversely, since the piecewise linear functions form a vector space, every piecewise linear functions is expressible in the hat function basis, i.e. in particular for our piecewise linear interpolant we have

$$p_{1,h}(x) = \sum_{i=1}^{n+1} c_i H_i(x)$$

for some coefficients c_i, $i = 1, \ldots n+1$.

Similar to the Lagrange polynomials the hat functions satisfy a **cardinality condition**

$$H_i(x_j) = \delta_{ij} \qquad \text{for } i, j = 1, \ldots n+1. \tag{11}$$

With this in mind interpolating a piecewise linear polynomial becomes analogous to (7) the simple expression

$$p_{1,h}(x) = \sum_{i=1}^{n+1} y_i H_i(x) \tag{12}$$

(i.e. the coefficients in above expressions $c_i = y_i$).

As a result piecewise linear interpolation becomes easy to implement:

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

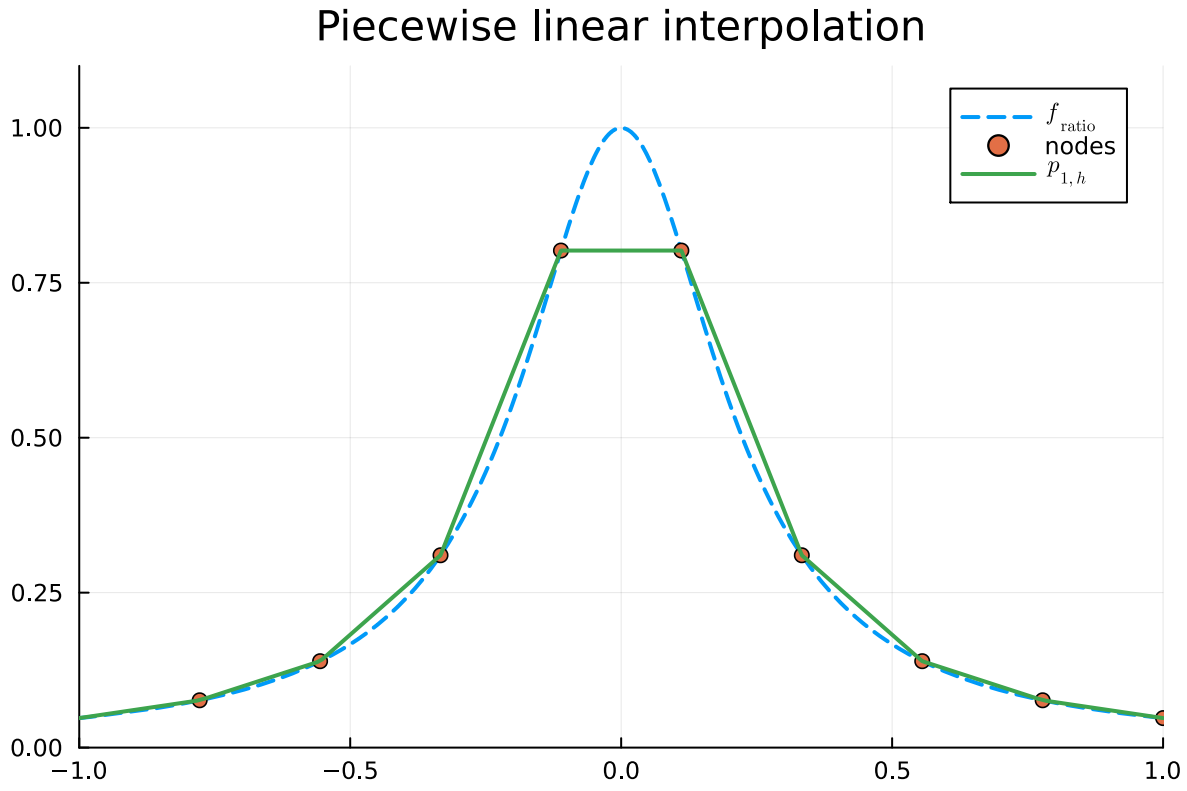
```
function pwlinear(x, y)
    # Construct a piecewise linear interpolation for the data values (xᵢ, yᵢ)
    # given in the vectors x = [x₁, …, xₙ₊₁] and y = [y₁, …, yₙ₊₁]
    H = [hatfun(x, i) for i in 1:length(x)]
    function interpolant(xx)
        sum(y[i] * H[i](xx) for i in 1:length(x))  # (12)
    end
end
```

We try this interpolation algorithm on our previous challenging example

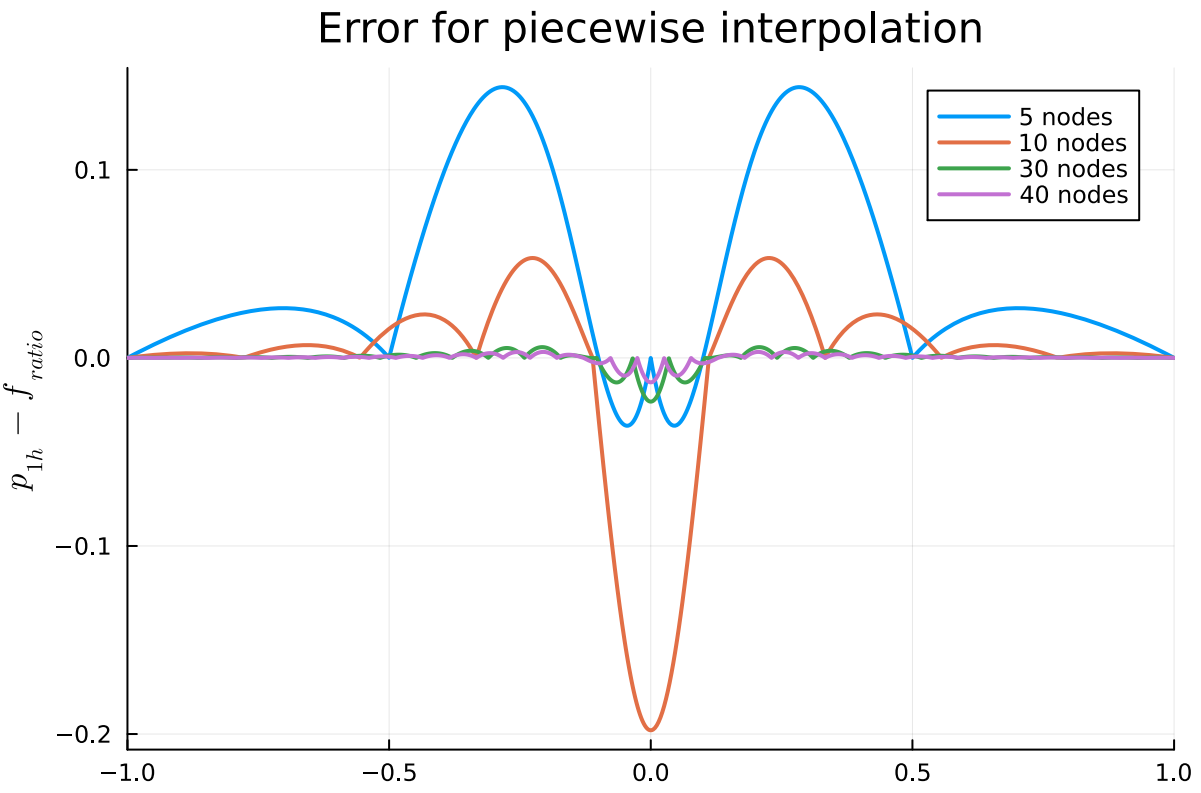
$$f_{\text{ratio}}(x) = \frac{1}{1 + 20x^2}$$

using `n_pwlinear` *equispaced* nodes in $[-1, 1]$:

- `n_pwlinear =` 10




```
let
    nodes  = range(-1, 1; length=n_pwlinear)
    p₁ₕ = pwlinear(nodes, fratio.(nodes))

    p = plot(fratio; xlims=(-1, 1), ylims=(0.0, 1.1), lw=2,
    label=L"f_\textrm{ratio}", ls=:dash,
             title="Piecewise linear interpolation")
    scatter!(p, nodes, fratio.(nodes); label="nodes")
    plot!(p, p₁ₕ, lw=2; label=L"p_{1,h}")
end
```




```
let
    p = plot(; xlims=(-1, 1), title="Error for piecewise interpolation",
    ylabel=L"p_{1h} - f_{ratio}")
    for n_nodes in (5, 10, 30, 40)
        nodes = range(-1, 1; length=n_nodes)
        p₁ₕ = pwlinear(nodes, fratio.(nodes))

        error(x) = p₁ₕ(x) - fratio(x)
        plot!(p, error, lw=2; label="$(n_nodes) nodes")
    end
    p
end
```

Unlike the case of polynomial interpolation, we observe a nice convergence as we increase `n_pwlinear`. Let's analyse this in more detail in the next section.

Error analysis

For the error analysis of the piecewise linear interpolation we restrict ourself to the case of *equidistant* nodes, which we considered in the most recent exercise. That is we assume a partitioning of the interval $[a, b]$ with $a = x_1 < x_2 < \cdots < x_{n+x} = b$ and equal nodal distance $h = x_{i+1} - x_i$.

In this setting the error analysis is a consequence of Theorem 2, equation (8). Indeed, for every interval $[x_i, x_{i+1}]$ we are constructing a linear interpolation between the points (x_i, y_i) and

(x_{i+1}, y_{i+1}) . In Theorem 2 we can thus set $n = 1$ and $b - a = x_{i+1} - x_i = h$ and obtain

$$\max_{x \in [x_i, x_{i+1}]} |f(x) - p_{1,h}(x)| \leq \frac{h^2}{8} \max_{x \in [x_i, x_{i+1}]} |f''(x)|$$

therefore

$$\begin{aligned} \|f - p_{1,h}\|_\infty &= \max_{x \in [a,b]} |f(x) - p_{1,h}(x)| \\ &= \max_{i=1,\ldots,n} \max_{x \in [x_i, x_{i+1}]} |f(x) - p_{1,h}(x)| \\ &\leq \max_{i=1,\ldots,n} \frac{h^2}{8} \max_{x \in [x_i, x_{i+1}]} |f''(x)| \\ &= \frac{h^2}{8} \|f''\|_\infty \end{aligned} .$$

We summarise in a Theorem:

Theorem 4

Let $f : [a, b] \to \mathbb{R}$ be a C^2 function and $a = x_1 < x_2 < \cdots < x_{n+1} = b$ with equal nodal distance $h = (b - a)/n$. The piecewise linear polynomial interpolating the data $(x_i, f(x_i))$ satisfies the error estimate

$$\|f - p_{1,h}\|_\infty \leq \alpha h^2 \|f''\|_\infty \tag{13}$$

with $\alpha = 1/8$.

Note, that this theorem is only true if the second derivative of f is continuous. Usually the second derivative f'' and thus its maximum norm $\|f\|_\infty$ are not known. However, we obtain that the interpolation error goes as $O(h^2)$ as $n \to \infty$. This is an example of **quadratic convergence**. More generally we define

Definition: Algebraic convergence

If an approximation has an error with asymptotic behaviour $O(h^m)$ as $h \to 0$ with m integer and h being a discretisation parameter (e.g. grid spacing, nodal distance, etc.), we say the approximation has **algebraic convergence**. If m is the largest such integer (i.e. the error is *not* $O(h^{m+1})$) then m is the order of accuracy.

Moreover we often refer to the case $m = 1$ as **linear convergence**, $m = 2$ as **quadratic convergence** and so on.

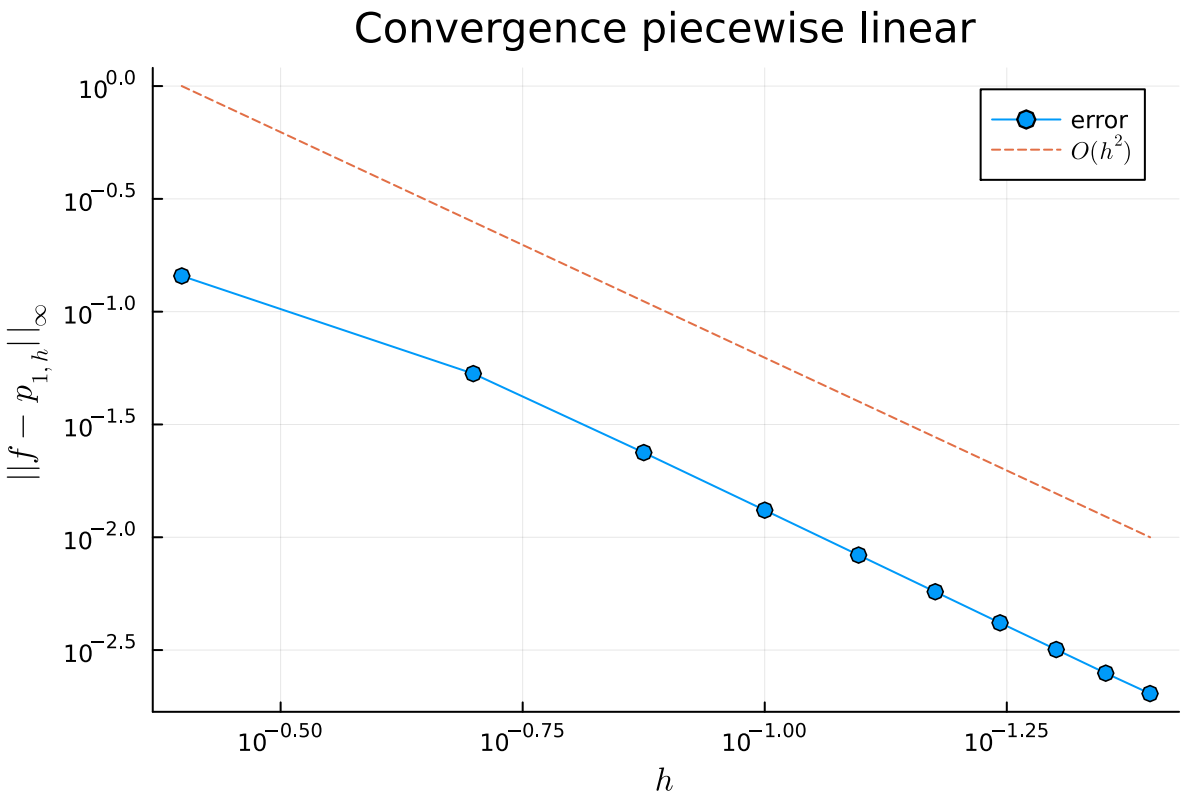
Potential confusion: Convergence terminology

Note again the difference in convergence terminology between **iterative methods** and **approximation schemes**. What is called algebraic convergence for approximation schemes would be sub-linear convergence for iterative schemes.

Let us finally illustrate the quadratic convergence of piecewise polynomial interpolation graphically. Since the error $\|f - p_{1,h}\|_\infty \sim \alpha h^2$ as $h \to 0$, taking logarithms on both sides we obtain

$$\log\left(\|f - p_{1,h}\|_\infty\right) \sim \log(\alpha) + 2\log(h).$$

Therefore the logarithm of the error is a *linear function* in the logarithm of the discretisation parameter. Moreover the **slope** gives us the convergence order, here $m = 2$. Note, that to obtain this behaviour any logarithm would suffice. We choose a $\log_{10}$-$\log_{10}$ plot as this is most easily realised in `Plots.jl` and indeed exhibits a slope of 2, meaning quadratic convergence (note that the x-axis is reversed in the plot.)




```julia
let
    fine = range(-1, 1, length=1000)  # Very fine grid

    n = 5:5:50   # Number of nodes
    maxerror = Float64[]  # Empty array for Float64 numbers
    for (i, n_nodes) in enumerate(n)
        nodes = range(-1, 1; length=n_nodes)
        p₁ₕ = pwlinear(nodes, fratio.(nodes))

        push!(maxerror, maximum(p₁ₕ.(fine) - fratio.(fine)))
    end

    h = 2 ./ n   # Discretisation parameter  == [2/element for element in n]
    p = plot(h, maxerror; label="error", title="Convergence piecewise linear",
             xlabel=L"h", xflip=true, xscale=:log10, ylabel=L"|| f-p_{1,h} ||_\infty",
             yscale=:log10, mark=:o, legend=:topright)

    # Generate guiding slope
    order2 = (h ./ h[1]) .^ 2
    plot!(p, h, order2, label=L"O(h^2)", ls=:dash)

    yticks!(p, 10.0 .^ (-2.5:0.5:0))
    xticks!(p, 10.0 .^ (-0.5:-0.25:-2))

    p
end
```

Convergence with respect to n or h

When discussing the convergence of a numerical method or an approximation scheme one typically finds two formulations of limits in the literature.

- **Convergence as $n \to \infty$:** In this case n is typically the **number of steps**, the **number of samples** or the **size of the approximation space**. For our **case of building an interpolating function** p to a ground-truth function f the number n is the number of data points and we study how $p \to f$ as $n \to \infty$.
- **Convergence as $h \to 0$:** In this case h is typically the **size of a step**, the **distance between samples** or the **size of a small displacement**. For our **case of building an interpolating function** p the value h is the spacing between nodes, so $h = \frac{b-a}{n}$ and we study how $p \to f$ as $h \to 0$.

The two formulations are often used interchangably when discussing convergence and the general idea is that n is (up to constants) the inverse of h and vice versa, so $n = O(1/h)$ or $h = O(1/n)$.

To complete the definition of convergence classes, let us return to the case of Chebyshev polynomial interpolation. We already mentioned below Theorem 3, that this method exhibits exponential convergence. While for this setting the nodal points are not equally spread, the definition of the nodal points ($x_k = -\cos\left(\frac{k\pi}{n}\right)$ for $k = 0, 1, \dots n$) allows to identify a discretisation parameter $h \sim 1/n$, which scales the distance between the nodes. We obtain the definition:

Definition: Exponential convergence

If an approximation has an error scaling as $O(c^{-\alpha h})$ with $c > 0$ and $\alpha > 0$ two constants and h a discretisation parameter, we say the approximation has **exponential convergence**.

To **determine the convergence behaviour graphically** we thus need to look at

- a log-log plot ($\log(y)$ versus $\log(x)$) if we suspect *algebraic convergence*. In this case the convergence will be a straight line with slope m.
- a log-linear plot ($\log(y)$ versus x) if we suspect *exponential convergence*. Again a straight line with slope $-\alpha$ will result.

Stability analysis

We are again interested in the effect of measurement noise on the quality of the polynomial interpolation. The goal is to compare the interpolation of a piecewise linear polynomial $p_{1,h}$ employing the noise-free data (x_i, y_i) with $y_i = f(x_i)$ with the polynomial $\tilde{p}_{1,h}$ obtained from the noisy data $(x_i, \tilde{y}_i)$ with $\tilde{y}_i = f(x_i) + \varepsilon_i$ and $|\varepsilon_i| \leq \varepsilon \; \forall i = 1, \ldots, n+1$. Due to (12) we can directly write

$$p_{1,h}(x) = \sum_{i=1}^{n+1} y_i H_i(x)$$

and

$$\tilde{p}_{1,h}(x) = \sum_{i=1}^{n+1} \tilde{y}_i H_i(x) = \sum_{i=1}^{n+1} (y_i + \varepsilon_i) H_i(x).$$

By linearity we note

$$|p_{1,h}(x) - \tilde{p}_{1,h}(x)| = \left| \sum_{i=1}^{n+1} \varepsilon_i H_i(x) \right| \leq \varepsilon \sum_{i=1}^{n+1} |H_i(x)| = \varepsilon \sum_{i=1}^{n+1} H_i(x) \overset{(*)}{=} \varepsilon \tag{14}$$

where in the last step $(*)$ we used the partitition of unity property, which you are asked to prove as an exercise.

Exercise

Use (12) to prove the **partition of unity**, namely $\sum_{i=1}^{n+1} H_i(x) = 1$.

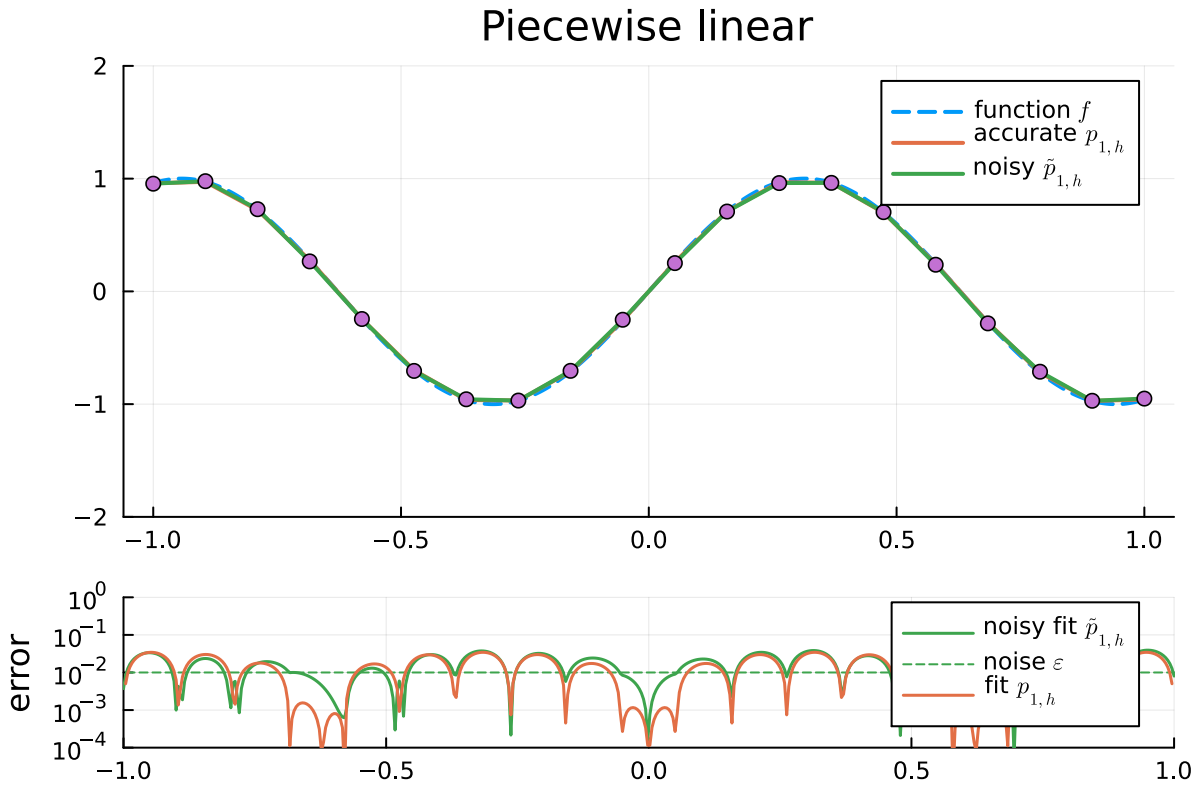
Based on (14) we can conclude:

- The **condition number** of piecewise linear interpolation is 1 **independent of the data**.
- As a result **small errors in the input values** x_i (e.g. from measurements) **only introduce small perturbations** in the polynomial: Piecewise linear interpolation is numerically stable.

We conclude by illustrating this result graphically. Again we consider the function $f_{\sin}(x) = \sin(5x)$ on the interval $[-1, 1]$, once evaluated without noise and once taking $\tilde{y}_i = f(x_i) + \eta_i$ where $|\eta_i| \leq 10^{\texttt{n\_log\_pl}}$. For construct the piecewise linear interpolation we take `n_nodes_pl` equally spaced nodes.

- Number of nodes `n_nodes_pl` = 20

- Noise amplitude `ε_pl` = 0.0100000000000000002

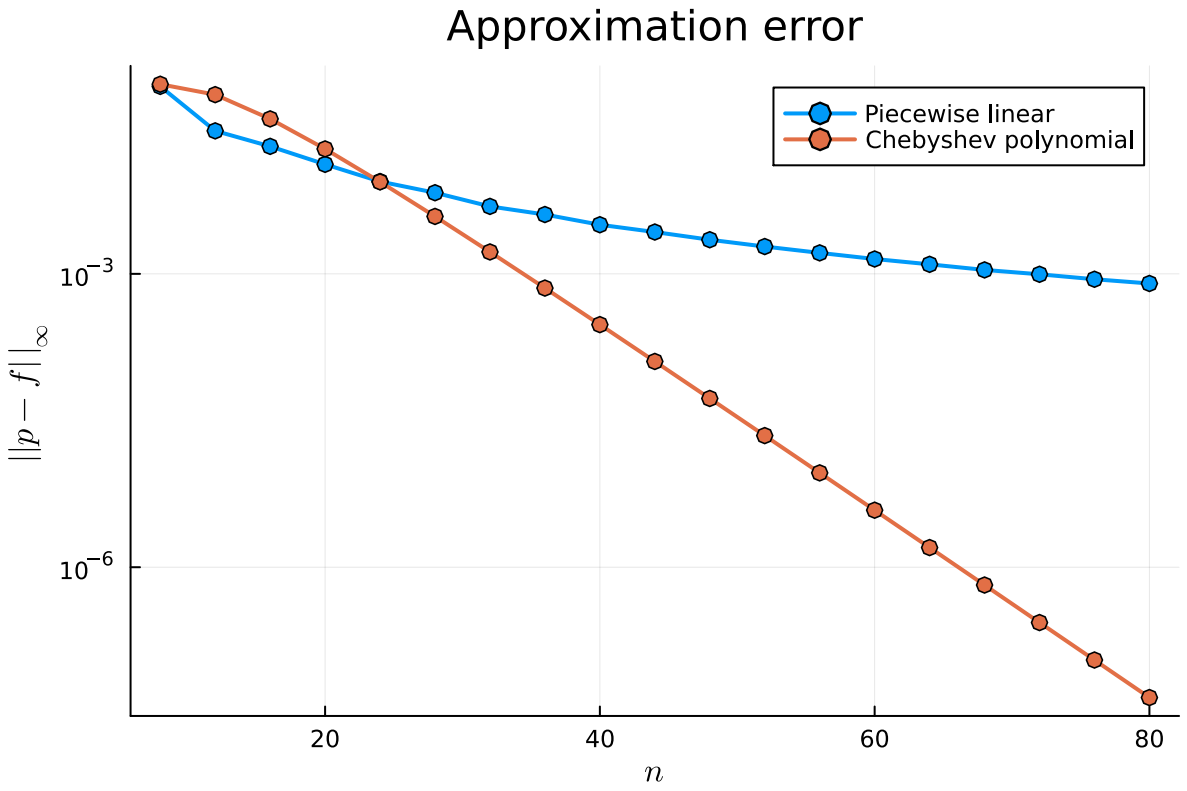


Optional: Spline interpolation

Over polynomial interpolation our previously discussed piecewise linear interpolation has the advantage that it is *always* convergent as the number of nodal points increases, even for an equidistant scheme. Moreover it enjoys a condition number of 1, thus is numerically very stable. However, it suffers from a few flaws, which make it unsuitable for many applications:

- The interpolant $p_{1,h}$ is globally only continuous since at all nodal points it is not neccessarily differentiable.
- Piecewise linear interpolation only converges quadratically. To understand why this is a problem, consider the following convergence curves on our example $f_{\sin} = \sin(5x)$. We simply need a lot more nodal points (and thus data / measurements) to get the same accuracy !

Show splines: ☐



An alternative approach is *cubic spline interpolation*:

Definition: Spline interpolation

Let $(x_i, y_i), i = 1, \dots, n+1$ denote the available data with ordering $a = x_1 < x_2 < \cdots < x_{n+1} = b$ of the nodal points. An **interpolating cubic spline** is a function $s_{3,h}$ such that

1. Between the nodal points, i.e. within the intervals $[x_i, x_{i+1}]$ the spline $s_{3,h}$ is a polynomial of degree 3.
2. $s_{3,h} \in C^2([a,b])$, that is $s_{3,h}$ is globally twice differentiable.
3. $s_{3,h}(x_i) = y_i$, that is $s_{3,h}$ interpolates the data.

Due to condition 1. the restriction $s_{3,h}|_{[x_i,x_{i+1}]}$ to an interval can be written as

$$s_{3,h}|_{[x_i,x_{i+1}]}(x) = a_i + b_i x + c_i x^2 + d_i x^3 \qquad \text{for } i = 1, \dots n,$$

which makes in total $4n$ unknowns to be determined.

In order to satisfy the continuity condition 2. we need $s_{3,h}$ and its first and second derivative to be continuous at the nodal points, that is

$$
\begin{aligned}
s_{3,h}(x_i^-) &= s_{3,h}(x_i^+), \qquad i = 2, \dots, n \\
s'_{3,h}(x_i^-) &= s'_{3,h}(x_i^+), \qquad i = 2, \dots, n \\
s''_{3,h}(x_i^-) &= s''_{3,h}(x_i^+), \qquad i = 2, \dots, n
\end{aligned}
$$

where we used the notation $f(x^+) = \lim_{\xi \searrow x} f(\xi)$ and $f(x^-) = \lim_{\xi \nearrow x} f(\xi)$. This gives $3(n-1)$ equations.

Additionally the interpolating condition 3. gives us $n+1$ equations, one for each of the nodes.

Summarising our findings we are thus left with

$$
4n - 3(n-1) - (n+1) = 2
$$

degrees of freedom, which we need to set to determine a unique spline. For this there are two mjor alternatives:

- **Natural spline:** $s_1'''(x_1) = s_1'''(x_{n+1}) = 0$
- **Not-a-knot spline:** $s_1'''(x_1) = s_2'''(x_2)$ and $s_1'''(x_n) = s_2'''(x_{n+1})$

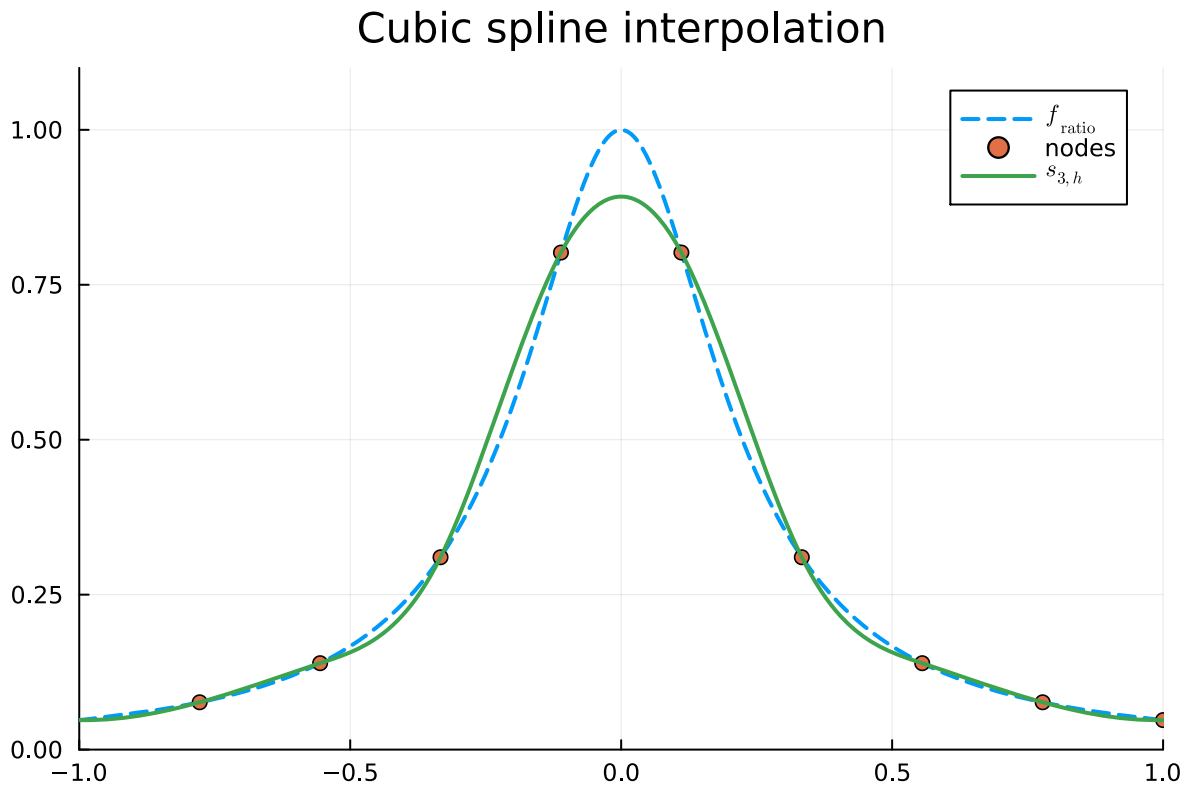
Natural splines have properties that make them theoretically more interesting, but not-a-knot splines give better pointwise accuracy.

Let's check how splines behave for our function

$$
f_{\text{ratio}}(x) = \frac{1}{1 + 20x^2}
$$

using `n_spline` equispaced nodes in $[-1, 1]$.

- `n_spline =`  10




```
let
    nodes = range(-1, 1; length=n_spline)
    s₃ₕ   = spinter(nodes, fratio.(nodes))

    p = plot(fratio; xlims=(-1, 1), ylims=(0.0, 1.1), lw=2,
             label=L"f_\textrm{ratio}", ls=:dash,
             title="Cubic spline interpolation")
    scatter!(p, nodes, fratio.(nodes); label="nodes")
    plot!(p, s₃ₕ, lw=2; label=L"s_{3,h}")
end
```

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

Already around 18 nodal points there is hardly any difference between the function and the interpolant visible.

For examples as well as more details on the computation of splines see chapter 5.3 of Driscoll, Brown: *Fundamentals of Numerical Computation*.

Optional: Error analysis

For cubic splines the following convergence result is known:

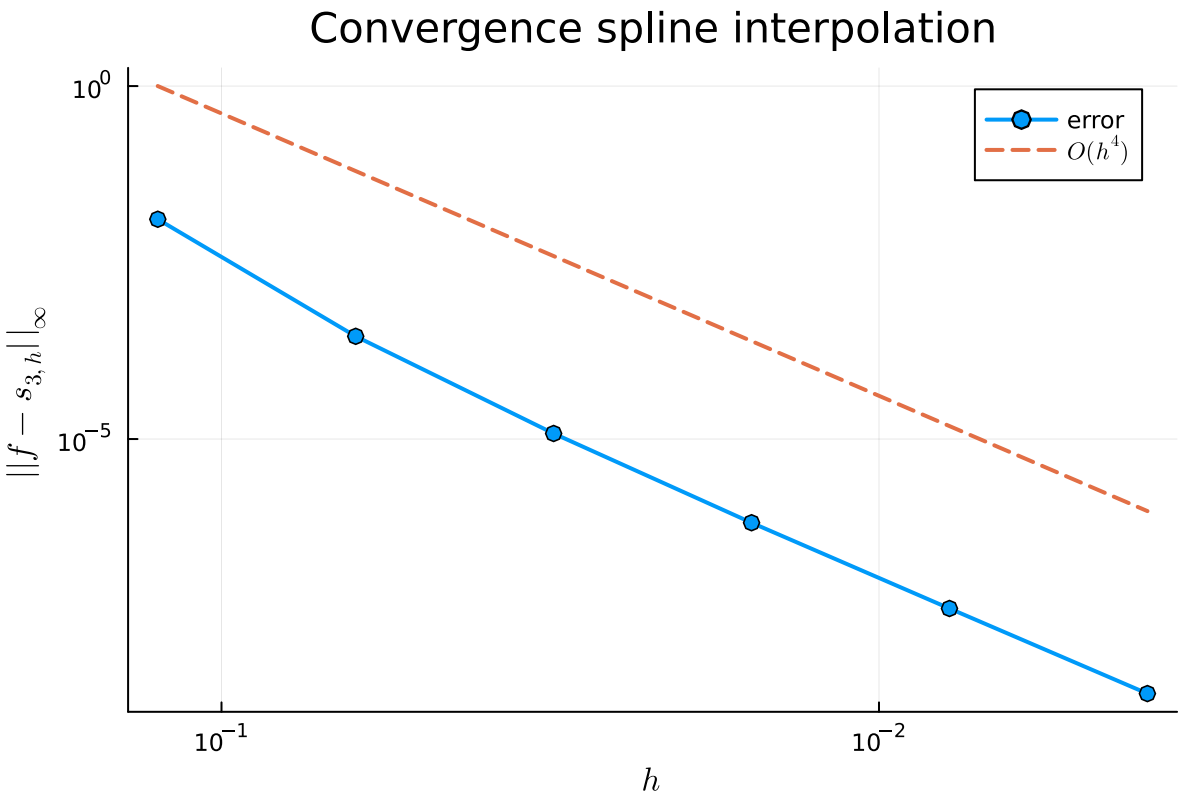
Theorem 5

Let $f \in C^4([a,b])$ be 4 times differentiable and let $a < x_1 < x_2 < \cdots < x_{n+1} = b$ be equispaced nodal points in $[a,b]$. The cubic spline $s_{3,h}$ interpolating the data $(x_i, f(x_i))$ satisfies the error bounds

$$\begin{aligned}
\|f - s_{3,h}\|_\infty &\leq C_0\, h^4\, \|f^{(4)}\|_\infty \\
\|f' - s'_{3,h}\|_\infty &\leq C_1\, h^3\, \|f^{(4)}\|_\infty \\
\|f'' - s''_{3,h}\|_\infty &\leq C_2\, h^2\, \|f^{(4)}\|_\infty
\end{aligned}$$

where $h = \frac{b-a}{n}$ is the length of each interval and C_0, C_1, C_2 are constants not depending on h.

We would thus expect a 4th order convergence. Let us investigate this visually using the function $f_{\text{ratio}}(x) = \frac{1}{1+20x^2}$. Since we expect algebraic convergence, we employ a log-log scale:




```
let
    fine = range(-1, 1, length=1000)  # Very fine grid

    n = 2 .^ (4:9)
    maxerror = Float64[]  # Empty array for Float64 numbers
    for (i, n_nodes) in enumerate(n)
        nodes = range(-1, 1; length=n_nodes)
        s₃ₕ = spinter(nodes, fratio.(nodes))
        push!(maxerror, maximum(s₃ₕ.(fine) - fratio.(fine)))
    end

    h = 2 ./ n
    p = plot(h, maxerror; label="error", title="Convergence spline interpolation",
             xlabel=L"h", xflip=true, xscale=:log10, ylabel=L"|| f-s_{3,h}
    ||_\infty",
             yscale=:log10, mark=:o, legend=:topright, lw=2)

    # Generate guiding slope
    order4 = (h ./ h[1]) .^ 4
    plot!(p, h, order4, label=L"O(h^4)", ls=:dash, lw=2)

    p
end
```

Regression and curve fitting

Consider again the case where we have acquired n data points $(x_i, \tilde{y}_i)$, $i = 1, 2, \ldots, n$ where

$$\tilde{y}_i = f(x_i) + \varepsilon_i$$

and the **measurement noice ε_i is substantial**. If we were to use an *interpolation technique* — as discussed in the previous sections — our goal would be to obtain a (potentially piecewise) polynomial p such that $p(x_i) = \tilde{y}_i$. However, since $\tilde{y}_i$ is only a *noisy* observation it makes **little sense to force the fitted polynomial to go through the data exactly**.

In this section we will now consider a generalisation to interpolation, where we give up on the interpolation condition $q(x_i) = \tilde{y}_i$. Instead we will now seek the representative $q \in \mathbb{P}$ taken from a model space $\mathbb{P}$, which best represents the data.

For example, in **least-squares linear regression** one seeks the straight line $q(x) = ax + b$, which yields the lowest squared error

$$e_{\text{LS}} = \sum_{i=1}^{n} |p_1^{\text{LS}}(x_i) - y_i|^2. \tag{15}$$

If we denote by $\mathbb{P}_m$ the space of all polynomials of degree at most m, then linear regression seeks the $q \in \mathbb{P}_1$, which gives smallest e_{LS}, or more mathematically

$$p_1^{\text{LS}} = \mathbf{argmin}_{q \in \mathbb{P}_1} |q(x_i) - y_i|^2.$$

A generalisation to higher-order polynomials can be formally defined as

Definition: Least-squares m-th degree polynomial regression

Given n data points $(x_i, \tilde{y}_i)$, $i = 1, 2, \ldots, n$ the m-th degree polynomial $p_m^{\text{LS}} \in \mathbb{P}_m$ satisfying

$$p_m^{\text{LS}} = \mathbf{argmin}_{q \in \mathbb{P}_m} |q(x_i) - y_i|^2. \tag{16}$$

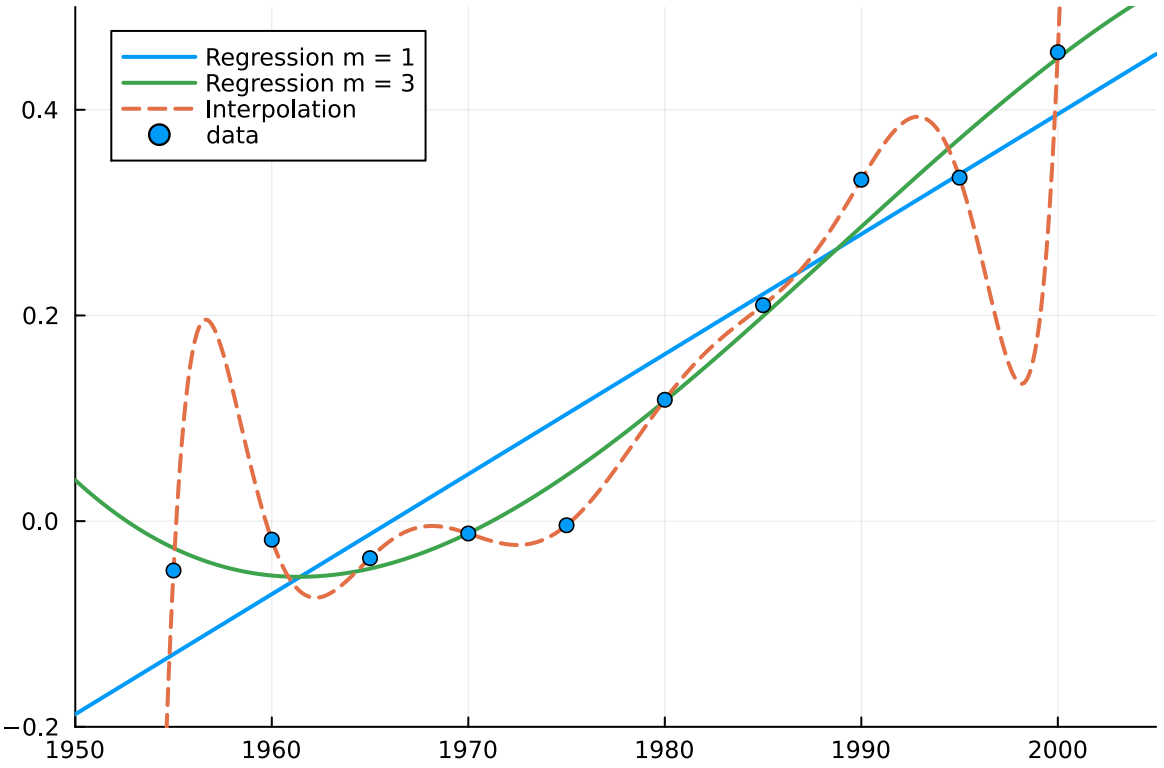
is called m-th degree polynomial least squares approximation of the data. In other words p_m^{LS} is such that

$$\sum_{i=1}^{n} \left(y_i - p_m^{\text{LS}}(x_i)\right)^2 \leq \sum_{i=1}^{n} \left(y_i - q(x_i)\right)^2 \qquad \forall q \in \mathbb{P}_m.$$

Least-squares regression is usually much **better suited to capture trends** in data than polynomial interpolation. To see this consider the following setting, where we fit polynomials of various degree. Our example data is the 5-year averages of the worldwide temperature anomaly as compared to the 1951–1980 average:

```
begin
    year = 1955:5:2000
    temp = [ -0.0480, -0.0180, -0.0360, -0.0120, -0.0040,
              0.1180, 0.2100, 0.3320, 0.3340, 0.4560 ]
end;
```

In the plot we show interpolation (i.e. $m = n - 1 = 9$ in this case) as well as polynomial regression with $m = 1$, $m = 3$ and $m = 5$.



As expected for such noisy data: The lower-degree polynomials seem to do a much better job.

Note that in polynomial regression the choice of the error metric (15) essentially determines which of the members of the model space $\mathbb{P}_m$ is considered to be the "best fit" — in the sense of being the minimiser of the minimisation problem (16). In this lecture we will only consider **least squares regression**, for which the error metric is the **sum of squares error**

$$e_{\text{LS}} = \sum_{i=1}^{n} |q(x_i) - y_i|^2.$$

Other choices are for example to employ

- $\mathbf{max}_i \, |q(x_i) - y_i|$, the maximal elementwise deviation
- $\sum_{i=1}^{n+1} |q(x_i) - y_i|$, absolute deviations

However, we will not consider these further.

Least squares problems

We will now consider how to solve polynomial regression problems (16). For this we introduce some linear algebra.

Recall from equation (1) at the start of discussion of polynomial interpolation, that each polynomial $q \in \mathbb{P}_m$ can be written as a linear combination

$$q(x) = \sum_{j=0}^{m} c_j x^j$$

in terms of the monomial basis $\mathbf{1} = x^0, x^1, \ldots, x^m$. Inserting this, the least-squares error expression can be rewritten as

$$e_{\text{LS}} = \sum_{i=1}^{n} \left| y_i - \sum_{j=0}^{m} c_j x_i^j \right|^2$$

and further by introducing the matrix / vector notation

$$\mathbf{V} = \begin{pmatrix} x_1^0 & x_1^1 & \cdots & x_1^m \\ x_2^0 & x_2^1 & \cdots & x_2^m \\ & \vdots & & \\ x_n^0 & x_n^1 & \cdots & x_n^m \end{pmatrix} \qquad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \qquad \mathbf{c} = \begin{pmatrix} c_0 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

as

$$e_{\text{LS}} = \|\mathbf{y} - \mathbf{V}\mathbf{c}\|_2^2 \tag{16}$$

where $\|v\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$ is the Euklidean norm. We again recognise $\mathbf{V}$ to be a Vandermonde matrix similar to the polynomial interpolation case, just in this case a rectangular matrix as

$n > m+1$, that is

$$\mathbf{V} = \begin{pmatrix} 1 & x_1 & \dots & x_1^m \\ 1 & x_2 & \dots & x_2^m \\ \vdots & & & \\ 1 & x_n & \dots & x_n^m \end{pmatrix} \in \mathbb{R}^{n\times(m+1)}$$

In polynomial regression our job is now to minimise expression (16), which means that we want to find the coefficient vector $\mathbf{c}$, which minimises $\|\mathbf{y} - \mathbf{V}\mathbf{c}\|_2^2$. A procedure to obtain this coefficient vector from of $\mathbf{y}$ and $\mathbf{V}$ is obtained by noting that polynomial regression is just a special case of a general least-squares problem, defined as:

Definition: Least-squares problem

Given $\mathbf{A} \in \mathbb{R}^{k\times l}$ and $\mathbf{b} \in \mathbb{R}^k$ with $k > l$ find

$$\mathbf{argmin}_{\mathbf{x}\in\mathbb{R}^l} \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2^2. \tag{17}$$

The vector $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x} \in \mathbb{R}^k$ is referred to as the **residual** of the least-squares problem.

Least squares problems arise in many applications (statistics, machine learning, ...) as well as polynomial regression. For the latter case the Vandermonde matrix $\mathbf{V}$ plays the role of $\mathbf{A}$, the vector of y_i-values $\mathbf{y}$ the role of $\mathbf{b}$ and the vector of coefficients $\mathbf{c}$ the role of the unknown $\mathbf{x}$.

Solving (17) can in fact be achieved by a concise explicit expression:

Theorem 6

Let $\mathbf{A} \in \mathbb{R}^{k\times\ell}$ and $\mathbf{b} \in \mathbb{R}^k$ with $k > \ell$. If $\mathbf{x} \in \mathbb{R}^\ell$ satisfies $\mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{0}$ then $\mathbf{x}$ solves the least-squares problem, i.e. $\mathbf{x}$ is the minimiser of $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|$.

Proof: We first consider the elementary identity for the sum of two vectors $\mathbf{u}$ and $\mathbf{v}$:

$$\begin{aligned} \|\mathbf{u}+\mathbf{v}\|^2 &= (\mathbf{u}+\mathbf{v})^T(\mathbf{u}+\mathbf{v}) \\ &= \mathbf{u}^T\mathbf{u} + \mathbf{u}^T\mathbf{v} + \mathbf{v}^T\mathbf{u} + \mathbf{v}^T\mathbf{v} \\ &= \|\mathbf{u}\|^2 + 2\mathbf{u}^T\mathbf{v} + \|\mathbf{v}\|^2 \end{aligned} \tag{18}$$

Now let $\mathbf{y} \in \mathbb{R}^l$ be an arbitrary vector and set $\mathbf{u} = \mathbf{A}\mathbf{x} - \mathbf{b}$ and $\mathbf{v} = \mathbf{A}\mathbf{y}$ in the above development. This results in

$$\begin{aligned}\|\mathbf{A}(\mathbf{x}+\mathbf{y}) - \mathbf{b}\|^2 &= \|(\mathbf{A}\mathbf{x} - \mathbf{b}) + \mathbf{A}\mathbf{y}\|^2 \\ &\overset{(18)}{=} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 + 2(\mathbf{A}\mathbf{y})^T(\mathbf{A}\mathbf{x} - \mathbf{b}) + \|\mathbf{A}\mathbf{y}\|^2 \\ &\geq \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 + 2\mathbf{y}^T\mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{b})\end{aligned}$$

Therefore, if $\mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) = 0$, then

$$\|\mathbf{A}(\mathbf{x}+\mathbf{y}) - \mathbf{b}\|^2 \geq \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 \qquad \text{for all } \mathbf{y} \in \mathbb{R}^l,$$

which implies that $\mathbf{x}$ is the minimiser of $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|$.

Due to the overall importance of least-squares problems the solution equation $\mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{0}$ is usually referred to as the normal equation.

Definition: Normal equations

Given $\mathbf{A} \in \mathbb{R}^{k \times l}$ and $\mathbf{b} \in \mathbb{R}^k$, which define a least-squares problem $\mathbf{argmin}_{\mathbf{x} \in \mathbb{R}^l} \|\mathbf{b} - \mathbf{A}\mathbf{x}\|$, the solution equation

$$\mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{0}$$

or equivalently

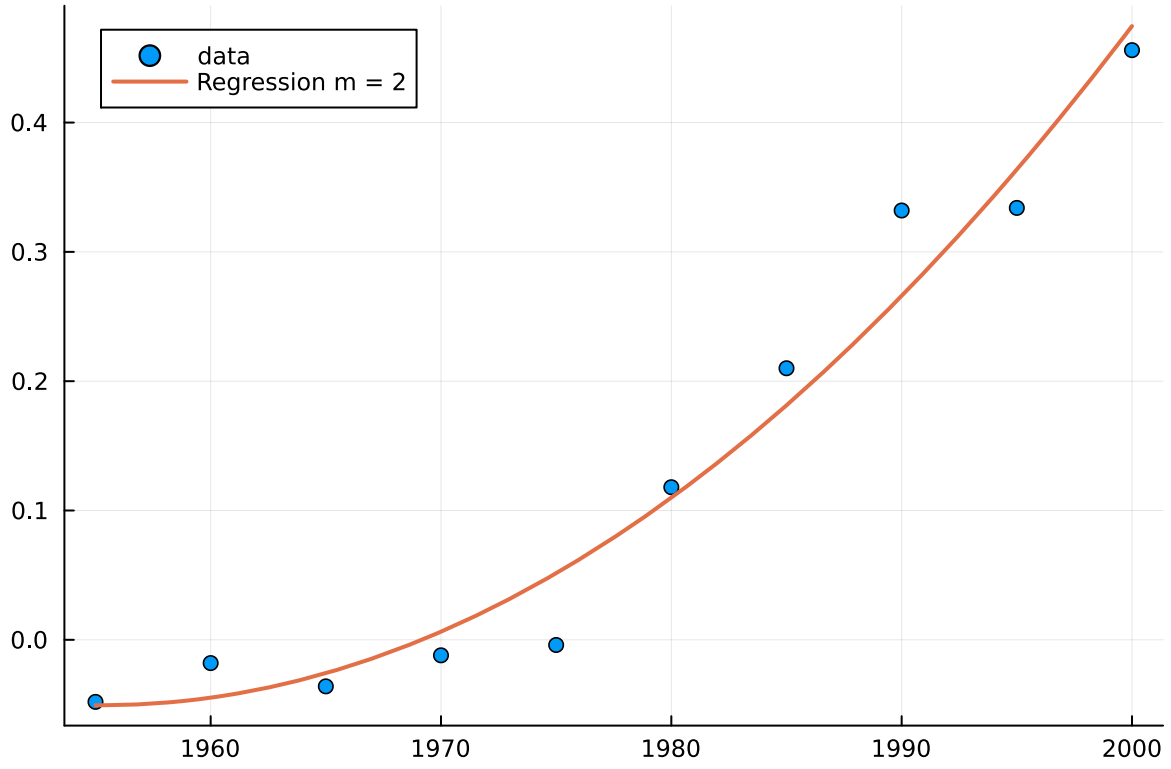
$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}. \tag{19}$$

are called the **normal equations**.

For a geometric interpretation of the normal equations, see chapter 3.2 of Driscoll, Brown: *Fundamentals of Numerical Computation*.

Based on this strategy we can now perform polynomial regression. We again employ the word temperature data between 1950 and 2000. You can change the polynomial degree using the slider:

- `m_poly =` 2



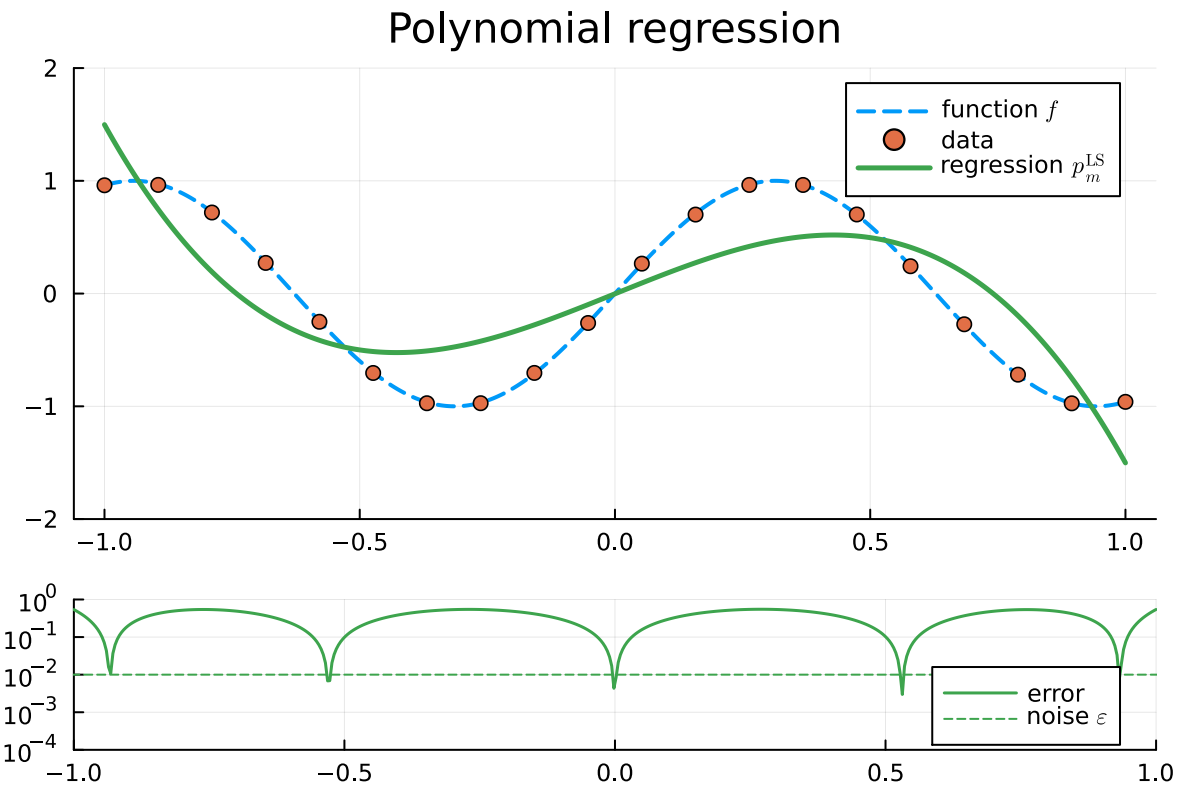

```
let
    x = year
    y = temp

    # Build Vandermonde matrix (using BigFloat to avoid numerical issues)
    V = ones(BigFloat, length(x), m_poly + 1)
    for k in 2:m_poly+1
        V[:, k] = V[:, k-1] .* x
    end

    # Solve normal equations
    c = (V'V) \ (V' * y)

    # Construct polynomial
    polynomial(x) = evalpoly(x, c)

    # Plot result
    p = scatter(year, temp; label="data")
    plot!(p, polynomial; label="Regression m = $m_poly", lw=2)

    p
end
```

Error analysis

In the general setting the analysis of least-squares problems is rather involved. Here, we will restrict ourselves to exploring the parameter space a little using an interactive visualisation.

▶ **Some interesting experiments with the visualisation below.**

- Number of samples `n` = 20
- Polynomial degree `m` = 2
- Noise amplitude `ε` = 0.010000000000000002



To close off we state some key results, which illustrate the key properties of the least-squares problem. We recall the setting: We are given n data points $(x_i, \tilde{y}_i)$, $i = 1, 2, \ldots, n$ where

$$\tilde{y}_i = f(x_i) + \varepsilon_i$$

are the noisy observations of a function f. Further we assume that $\mathbb{E}(\varepsilon_i) = 0$, i.e. the noise follows a distribution with mean zero and $\sigma = \sqrt{\mathbb{V}\mathbf{ar}(\varepsilon_i)}$ is the square root of the variance of the noise.

- In the **absence of measurement noise**, i.e. $\sigma = 0$, **least-squares can recover polynomials exactly**. Imagine f to be a polynomial of degree $m \ll n$. In this case the polynomial f is the *only* m-th degree polynomial to give a zero least-squares error, i.e.

$$0 = \sum_{i=1}^{n} |p_m^{\text{LS}}(x_i) - f(x_i)|^2 \text{ with } p_m^{\text{LS}} \in \mathbb{P}_m \quad \Leftrightarrow \quad p_m^{\text{LS}} = f.$$

  and our result is exact.
- With **non-zero measurement errors** the approximation error $\|f - p_m^{\text{LS}}\|_\infty$ is proportional to $\sqrt{\frac{m+1}{n}}\sigma$. Thus while every measurement has been perturbed by an error ε_i of order around σ, the **approximation error** is much smaller of order $\sqrt{\frac{m+1}{n}}\sigma$ and moreover **becomes smaller** as the **number of samples increases**.

  However, if the **degree of the fitting polynomial m becomes too large** — say comparable to n —, then the least-squares polynomial becomes similar to the interpolating polynomial. This we saw to become unstable as m gets large. Therefore the **quality of the least-squares approximation** does **deteriorate when $m \to n$**.

QR factorisation

In the above discussion on regression we arrived at least-squares problems (17)

$$\text{argmin}_{\mathbf{x} \in \mathbb{R}^l} \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2^2$$

with $\mathbf{A} \in \mathbb{R}^{k \times l}$ and $k > l$ and $\mathbf{b} \in \mathbb{R}^k$. Our proposed solution strategy was to solve the normal equations (19)

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}.$$

which are mathematically equivalent to (17). Since $\mathbf{A}^T\mathbf{A} \in \mathbb{R}^{l \times l}$ such that we can use LU factorisation as described in Algorithm 3 of the Direct methods for linear systems chapter.

Let us consider the numerical stability of this approach. Clearly $\mathbf{N} := \mathbf{A}^T\mathbf{A}$ is a symmetric matrix. We can therefore apply Lemma 6 of the Direct methods for linear systems chapter and conclude

$$\kappa(\mathbf{N}) = \frac{\lambda_{\max}^{\text{abs}}(\mathbf{N})}{\lambda_{\min}^{\text{abs}}(\mathbf{N})} = \frac{\lambda_{\max}^{\text{abs}}(\mathbf{A}^T\mathbf{A})}{\lambda_{\min}^{\text{abs}}(\mathbf{A}^T\mathbf{A})} = \kappa(\mathbf{A})^2.$$

In other words the condition number of solving the normal equations is the **square of the condition number of $\mathbf{A}$**.

For typical polynomial regression problems the condition number of the normal equations, $\kappa(\mathbf{A})^2$ can get huge. Take as an example the condition number of the vandermode matrix for regressing a 5-th degree polynomial with 10 equispaced datapoints between $\mathbf{0}$ and $\mathbf{10}$:

```
let
    deg   = 5  # Polynomial degree
    nodes = range(0.0, 10.0; length=10)  # 10 equispaced datapoints

    # Library function to compute Vandermonde matrix
    V = Polynomials.vander(Polynomial, nodes, deg)

    κA = cond(V)  # Condition number of the Vandermond matrix, i.e. our κ(A)

    println("κ(A) = $κA")
    println("κ(N) = $(κA^2)")
end
```

```
κ(A) = 368158.11195104703
κ(N) = 1.3554039539535968e11
```

Notably the **condition number** of even such a simple regression problem is 10^{11}, which **is enormous**.

As a result the accuracy of an approach based on LU factorisation is very poor and **extremely sensitive to any noise** in the RHS $\mathbf{b}$, respectively the input data for regression.

Definition of QR factorisation

An alternative way of solving least squares problems is to employ QR factorisation.

Definition: QR factorisation

Every real matrix $\mathbf{A} \in \mathbb{R}^{k \times l}$ with $k \geq l$ can be written as $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q} \in \mathbb{R}^{k \times l}$ is a matrix with orthonormal columns and $\mathbf{R} \in \mathbb{R}^{l \times l}$ is an upper-triangular matrix.

In Julia QR factorisation is available via the `qr` function. For example:

```
A = 6×3 Matrix{Float64}:
     0.869351  -0.491542    0.531827
     0.684243  -0.0424514   1.08095
     0.016627  -1.16842    -0.762098
     0.655444  -0.280242    1.93213
     0.9714     1.16325    -0.417193
     1.12885    1.68718     0.138746
```

```
A = randn(6, 3)  # A random matrix
```

```
begin
    Qcompact, R = qr(A)

    # For reasons of numerical efficiency `qr` does not actually return the Q
    # matrix, but a different data structure, that first needs to be converted
    # into the Q matrix we use in the above definition:
    Q = Matrix(Qcompact)
end;
```

The resulting factors $\mathbf{Q}$ and $\mathbf{R}$ are:

```
6×3 Matrix{Float64}:
 -0.441815    -0.486916  -0.159771
 -0.347741    -0.219595   0.259123
 -0.00845004  -0.559985  -0.583036
 -0.333105    -0.32418    0.646938
 -0.493678     0.269564  -0.378383
 -0.573697     0.472599  -0.0754588
```

```
Q
```

```
3×3 Matrix{Float64}:
 -1.96768  -1.20704  -1.12166
  0.0       2.10474  -0.742808
  0.0       0.0       2.03681
```

```
R
```

We notice $\mathbf{R}$ to be an **upper triangular matrix**. The **columns of $\mathbf{Q} \in \mathbb{R}^{k \times l}$ are orthonormal**, i.e.

$$\mathbf{q}_i \cdot \mathbf{q}_j = \delta_{ij} \qquad 1 \leq i, j \leq l$$

or in code

```
for i in 1:3, j in 1:3
    qᵢ = Q[:, i]
    qⱼ = Q[:, j]
    result = dot(qᵢ, qⱼ)

    println(" ", i, "  ", j, "  ", round(result; digits=2))
end
```

```
1  1  1.0
1  2  -0.0
1  3  -0.0
2  1  -0.0
2  2  1.0
2  3  -0.0
3  1  -0.0
3  2  -0.0
3  3  1.0
```

We further easily verify $\mathbf{A} = \mathbf{QR}$:

```
5.878434998123904e-16
```

```
norm(  A - Q * R  )
```

Note that a consequence of (20), i.e. orthonormal columns, are the following properties:

Theorem 7: Properties of matrices with orthogonal columns

Let $\mathbf{Q} \in \mathbb{R}^{k \times l}$ with $k \geq l$ be a matrix with orthonormal columns, i.e.

$$\mathbf{q}_i \cdot \mathbf{q}_j = \delta_{ij} \qquad 1 \leq i, j \leq l \tag{20}$$

where q_i is the i-th column of $\mathbf{Q}$. Then it holds

1. $\mathbf{Q}^T\mathbf{Q} = \mathbf{I} \in \mathbb{R}^{l \times l}$, the identity matrix
2. $\|\mathbf{Qx}\| = \|\mathbf{x}\|$ for all vectors $\mathbf{x} \in \mathbb{R}^l$.
3. $\|\mathbf{Q}\| = 1$

Proof:

1. $O_{ij} = (\mathbf{Q}^T\mathbf{Q})_{ij} = \sum_{a=1}^{k} Q_{ai}\, Q_{aj} = \mathbf{q}_i \cdot \mathbf{q}_j = \delta_{ij}$. Therefore $\mathbf{O} = \mathbf{Q}^T\mathbf{Q}$ is an identity matrix.
2. Using what we just proved, we get

$$\|\mathbf{Qx}\|^2 = (\mathbf{Qx})^T(\mathbf{Qx}) = \mathbf{x}^T\mathbf{Q}^T\mathbf{Qx} \overset{1.}{=} \mathbf{x}^T\mathbf{Ix} = \|\mathbf{x}\|^2$$

3. Again making use of 1. we get

$$\|\mathbf{Q}\|^2 = \lambda_{\max}^{\text{abs}}(\mathbf{Q}^T\mathbf{Q}) \overset{1.}{=} \lambda_{\max}^{\text{abs}}(\mathbf{I}) = 1$$

Employing QR for solving least-squares

We insert the factorisation $\mathbf{A} = \mathbf{QR}$ into the normal equations (19) and simplify using Theorem 7 point 1.:

$$\begin{aligned}\mathbf{A}^T\mathbf{Ax} &= \mathbf{A}^T\mathbf{b}\\ \mathbf{R}^T\mathbf{Q}^T\mathbf{QRx} &= \mathbf{R}^T\mathbf{Q}^T\mathbf{b}\\ \mathbf{R}^T\mathbf{Rx} &= \mathbf{R}^T\mathbf{Q}^T\mathbf{b}\end{aligned}$$

We now assume the normal equations $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$ to be well-posed, i.e. that the matrix $\mathbf{A}^T\mathbf{A}$ is non-singular. As a result

$$\det(\mathbf{R}^T)^2 = \det(\mathbf{R}^T\mathbf{R}) = \det(\mathbf{A}^T\mathbf{A}) \neq 0$$

implying that $\det(\mathbf{R}^T) \neq 0$ and thus that $\mathbf{R}^T$ is invertible. Multiplying from the left with $(\mathbf{R}^T)^{-1}$ we thus obtain:

$$\mathbf{Rx} = \mathbf{Q}^T\mathbf{b}. \tag{21}$$

Since $\mathbf{R}$ is an upper-triangular matrix, this linear system can be efficiently solved using backward substitution. This leads to the following algorithm:

Algorithm 1: Solving least-squares problems using QR factorisation

Given a least-squares problem (17)

$$\mathbf{argmin}_{\mathbf{x}\in\mathbb{R}^l}\|\mathbf{b} - \mathbf{Ax}\|_2^2$$

with $\mathbf{A} \in \mathbb{R}^{k\times l}$, $\mathbf{b} \in \mathbb{R}^k$, $k \geq l$ and well-posed normal equations. This can be solved for $\mathbf{x} \in \mathbb{R}^l$ as follows:

1. Perform QR factorisation $\mathbf{A} = \mathbf{QR}$.
2. Compute $\mathbf{z} = \mathbf{Q}^T\mathbf{b}$
3. Solve $\mathbf{Rx} = \mathbf{z}$ for $\mathbf{x}$ using *backward* substitution.

Let us understand this algorithm by executing the steps manually for solving the problem

$$\text{argmin}_{\mathbf{x}_M \in \mathbb{R}^l} \|\mathbf{b}_M - \mathbf{M}\mathbf{x}_M\|_2^2$$

with $\mathbf{M}$ and $\mathbf{b}_M$ defined as:

```
M = 6×3 Matrix{Float64}:
      0.436573  -0.733883  -0.658225
     -0.886233  -0.483165  -0.338844
     -0.962202   1.80926   -0.135935
     -0.597819   1.50037   -0.0794233
      1.16542    0.357884   0.355076
     -0.398133   0.498849  -1.25601
```

```
1 M = randn(6, 3)
```

```
b_M = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

```
1 b_M = ones(6)
```

Step 1: Perform QR factorisation of $\mathbf{M}$:

```
1 begin
2     fac = qr(M)
3     Q_M = Matrix(fac.Q)  # Get Q factor as a matrix (see above)
4 end;
```

Step 2: Compute $\mathbf{z} = \mathbf{Q}^T\mathbf{b}_M$

```
z_M = [0.639362, 0.954482, -1.4031]
```

```
1 z_M = Q_M' * b_M
```

Step 3: Backward-substitute $\mathbf{z}_M$ wrt. $\mathbf{R}$:

```
x_D = [0.276841, 0.504727, -1.01444]
```

```
1 x_D = backward_substitution(fac.R, z_M)
```

Verify result: This solves the normal equations $\mathbf{M}^T\mathbf{M}\mathbf{x}_M = \mathbf{M}^T\mathbf{b}_M$:

```
[-2.22045e-16, -1.33227e-15, 1.77636e-15]
```

```
1 M' * M * x_D - M' * b_M
```

Recall that our starting point was that when solving the least-squares problem

$$\text{argmin}_{\mathbf{x} \in \mathbb{R}^l} \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2^2$$

using the normal equations, then the condition number for such an algorithm was $\kappa(\mathbf{A})^2$, which is very bad if the condition number $\kappa(\mathbf{A})$ of the matrix becomes large.

When employing a QR factorisation the central problem to be solved is (21), which has a condition number $\kappa(\mathbf{R})$. Now noticing

$$\kappa(\mathbf{R}) = \frac{\sqrt{\lambda^{\text{abs}}_{\text{max}}(\mathbf{R}^T\mathbf{R})}}{\sqrt{\lambda^{\text{abs}}_{\text{min}}(\mathbf{R}^T\mathbf{R})}} \overset{\mathbf{Q}^T\mathbf{Q}=\mathbf{I}}{=} \frac{\sqrt{\lambda^{\text{abs}}_{\text{max}}(\mathbf{R}^T\mathbf{Q}^T\mathbf{Q}\mathbf{R})}}{\sqrt{\lambda^{\text{abs}}_{\text{min}}(\mathbf{R}^T\mathbf{Q}^T\mathbf{Q}\mathbf{R})}} = \frac{\sqrt{\lambda^{\text{abs}}_{\text{max}}(\mathbf{A}^T\mathbf{A})}}{\sqrt{\lambda^{\text{abs}}_{\text{min}}(\mathbf{A}^T\mathbf{A})}} = \kappa(\mathbf{A})$$

We realise that the **condition number of solving least squares with QR factorisation** remains $\kappa(\mathbf{A})$ [1]. **Using QR to solve least-squares problems is more numerically stable** than using LU factorisation of the normal equations.

[1]:
> This again assumes our error model of the *Numerical stability* section of Direct methods for linear systems, i.e. that the matrix $\mathbf{A}$ is given exactly and all numerical noise comes from $\mathbf{b}$. As a result computing the QR factorisation of $\mathbf{A}$ is free of error. This is an assumption, which is wrong in practice. See for example chapter 4.8.3 of Stoer, Bulirsch: *Introduction to Numerical Analysis* (2002) for a detailed discussion.

Computing QR factorisations

One standard algorithm to compute QR factorisations is using the **Gram-Schmidt process**. It's idea is to produce the orthonormal matrix $\mathbf{Q}$ by orthonormalising the columns of $\mathbf{A}$. The matrix $\mathbf{R}$ is then obtained by keeping track of the linear combinations of the column vectors of $\mathbf{Q}$ that are necessary to obtain $\mathbf{A}$.

This is best illustrated using an example. Consider the matrix

$$\mathbf{A} = \begin{pmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \mathbf{a}_3 \\ \downarrow & \downarrow & \downarrow \end{pmatrix} = \begin{pmatrix} 4 & 2 & 3 \\ 0 & 3 & -4 \\ 0 & 4 & 3 \end{pmatrix}$$

and our goal is to obtain the matrices

$$\mathbf{Q} = \begin{pmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \mathbf{q}_3 \\ \downarrow & \downarrow & \downarrow \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{pmatrix}$$

where the vectors $\mathbf{q}_1$, $\mathbf{q}_2$ and $\mathbf{q}_3$ are orthonormal.

The Gram-Schmidt algorithm proceeds as follows:

- **Iteration 1:** We normalise the first vector and save the normalisation constant:

$$\mathbf{q}_1 = \mathbf{a}_1 / R_{11} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \qquad \text{where} \quad R_{11} = \|\mathbf{a}_1\| = 4$$

- **After iteration 1**:

$$\mathbf{Q} = \begin{pmatrix} 1 & & \\ 0 & & \\ 0 & & \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} 4 & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{pmatrix}$$

- **Iteration 2:**
  - **Step 1:** Orthogonalise $\mathbf{a_2}$ against $\mathbf{q_1}$, i.e. Remove the part of $\mathbf{a_2}$ already contained in $\mathbf{q_1}$:

$$\mathbf{v} = \mathbf{a}_2 - \underbrace{(\mathbf{q}_1 \cdot \mathbf{a}_2)}_{=R_{12}=2} \mathbf{q}_1 = \begin{pmatrix} 0 \\ 3 \\ 4 \end{pmatrix}$$

  - **Step 2:** Normalise and save the result:

$$\mathbf{q}_2 = \mathbf{a}_2 / R_{22} = \begin{pmatrix} 0 \\ 0.6 \\ 0.8 \end{pmatrix} \qquad \text{where} \quad R_{22} = \|\mathbf{v}\| = 5$$

- **After iteration 2**:

$$\mathbf{Q} = \begin{pmatrix} 1 & 0 & \\ 0 & 0.6 & \\ 0 & 0.8 & \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} 4 & 2 & R_{13} \\ 0 & 5 & R_{23} \\ 0 & 0 & R_{33} \end{pmatrix}$$

- **Iteration 3:**
  - **Step 1:** Orthogonalise $\mathbf{a_3}$ against $\mathbf{q_1}$ and $\mathbf{q_2}$:

$$\mathbf{v} = \mathbf{a}_3 - \underbrace{(\mathbf{q}_1 \cdot \mathbf{a}_3)}_{=R_{13}=3} \mathbf{q}_1 - \underbrace{(\mathbf{q}_2 \cdot \mathbf{a}_3)}_{=R_{23}=0} \mathbf{q}_2 = \begin{pmatrix} 0 \\ -4 \\ 3 \end{pmatrix}$$

  - **Step 2:** Normalise and save the result:

$$\mathbf{q}_3 = \mathbf{a}_3 / R_{33} = \begin{pmatrix} 0 \\ -0.8 \\ 0.6 \end{pmatrix} \qquad \text{where} \quad R_{33} = \|\mathbf{v}\| = 5$$

- **Final answer:**

$$\mathbf{Q} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0.6 & -0.8 \\ 0 & 0.8 & 0.6 \end{pmatrix} \qquad \mathbf{R} = \begin{pmatrix} 4 & 2 & 3 \\ 0 & 5 & 0 \\ 0 & 0 & 5 \end{pmatrix}$$

Generalising this algorithm to a matrix $\mathbf{A} \in \mathbb{R}^{k \times l}$ leads to the following Julia implementation:

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

```
function gram_schmidt_qr(A)
    k, l = size(A)
    Q = zeros(k, l)
    R = zeros(l, l)

    for j in 1:l
        v = A[:, j]
        for i in 1:j-1
            R[i, j] = dot(Q[:, i], A[:, j])
            v = v - R[i, j] * Q[:, i]
        end
        R[j, j] = norm(v)
        Q[:, j] = v / R[j, j]
    end

    (; Q, R)
end
```

Let's verify this implementation works as expected:

```
AA = 3×3 Matrix{Int64}:
 4  2   3
 0  3  -4
 0  4   3
```

```
AA = [4 2  3;
      0 3  -4;
      0 4  3]
```

```
fac_AA = gram_schmidt_qr(AA);
```

```
3×3 Matrix{Float64}:
 1.0  0.0   0.0
 0.0  0.6  -0.8
 0.0  0.8   0.6
```

```
fac_AA.Q
```

```
3×3 Matrix{Float64}:
 4.0  2.0  3.0
 0.0  5.0  4.44089e-16
 0.0  0.0  5.0
```

```
1 fac_AA.R
```

```
3×3 Matrix{Float64}:
 0.0  0.0   0.0
 0.0  0.0   4.44089e-16
 0.0  0.0  -4.44089e-16
```

```
1 fac_AA.Q * fac_AA.R - AA
```

In practice this algorithm is often **numerically not sufficiently stable**, which is why Julia's `qr` by default employs a different procedure using so-called Householder reflectors. For more details see chapter 3.4 of Driscoll, Brown: *Fundamentals of Numerical Computation*.

Summary

- **Interpolation** and **least-squares regression** techniques is one common approach to **extract a model p from n observed data** points (x_i, y_i) (with $i = 1, 2, \ldots, n$). Based on this model one can make predictions about unseen x_{n+1} namely as the points $(x_{n+1}, p(x_{n+1}))$.
- Polynomial interpolation on **equally spaced data points** leads to Runge's phaenomenon as the polynomial degree m is growing.
    - Moreover this problem is **ill-conditioned**, i.e. extremely susceptible to numerical or experimental noise in the training data (x_i, y_i).
- **One solution**: Keep the polynomial degree m low, for example:
    - Use piecewise polynomial (e.g. **piecewise linear**) interpolation techniques.
    - Use **more observations n than m**, i.e. perform **least-squares regression** with $n \gg m$
    - This generally leads to methods with **algebraic convergence** when approximating a smooth function f.
    - Moreover such problems are generally **well-conditioned**. E.g. for piecewise linear interpolation the condition number is 1 independent of the employed data — the best-possible value.
- The **other solution** is to use **non-equally spaced points**:
    - The typical approach are **Chebyshev nodes**
    - These lead to **exponential convergence**

Notice that all of these problems lead to linear systems $\mathbf{A}\mathbf{x} = \mathbf{b}$ that we need to solve. How this can me done numerically we will see in Direct methods for linear systems.

Appendix

Some functions we need to define:

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

```
function backward_substitution(U, b)
    n = size(U, 1)
    x = zeros(n)
    x[n] = b[n] / U[n, n]
    for i in n-1:-1:1  # Note that this loop goes *backwards* n-1, n-2, ..., 1
        row_sum = 0.0
        for j in i+1:n
            row_sum += U[i, j] * x[j]
        end
        x[i] = (b[i] - row_sum) / U[i, i]
    end
    x
end
```

Numerical analysis

1. Introduction
2. The Julia programming language
3. Revision and preliminaries
4. Root finding and fixed-point problems
5. Direct methods for linear systems
6. Iterative methods for linear systems
7. Interpolation
8. Numerical integration
9. Numerical differentiation
10. Boundary value problems
11. Eigenvalue problems
12. Initial value problems