Example 103. Suppose we approximate $\sin(x)$ on $[-1,1]$ by a polynomial interpolation.

(a) Give an upper bound for the maximal error if we use 5 Chebyshev nodes.

(b) How many Chebyshev nodes do we need to use in order to guarantee that the maximal error is at most 10^{-16}?

Solution. In the case $f(x)=\sin(x)$ we know that $|f^{(n)}(x)|\leqslant 1$ for all $x\in[-1,1]$ and all n.
Therefore, by Theorem 98, using n Chebyshev nodes for the interpolation $P_{n-1}(x)$, the error is bounded as $\max_{x\in[-,1,1]}|f(x)-P_{n-1}(x)|\leqslant\frac{1}{2^{n-1}n!}\max_{\xi\in[-,1,1]}|f^{(n)}(\xi)|\leqslant\frac{1}{2^{n-1}n!}$.

(a) With $n=5$ nodes, this upper bound becomes $\frac{1}{2^4\cdot 5!}=\frac{1}{16\cdot 120}=\frac{1}{1920}\approx 0.00052$.

(b) We need to choose n so that $\frac{1}{2^{n-1}n!}\leqslant 10^{-16}$ or, equivalently, $2^{n-1}n!\geqslant 10^{16}$. We compute $2^{n-1}n!$ for $n=1,2,\ldots$ and find that this first happens when $n=15$ (see the next Python example). Thus, for $n=15$ Chebyshev nodes the maximal error is guaranteed to be no more than 10^{-16}.

Comment. Recall that double precision floats have a precision of slightly less than 16 decimal digits. Thus we could, in theory, implement an accurate $\sin(x)$ function by using a degree 14 polynomial (or even lower degree if we take symmetry into account).

Advanced comment. Note that some care is required to translate this result into practice. For instance, if we proceed as in Example 93 then the resulting maximal error using 15 Chebyshev nodes turns out to be about $4.7\cdot 10^{-11}$, which is considerably larger than 10^{-16} (even if we take into account that we are limited by using double precision floats). The cause for this is that the function `interpolate.lagrange` does not compute the interpolating polynomial accurately to full precision. Indeed, in the documentation for that function, we find the following statement: *Warning: This implementation is numerically unstable. Do not expect to be able to use more than about 20 points even if they are chosen optimally.*

Example 104. Python A function for computing $n!$ is available in the Python `math` library (as well as in `numpy` and `scipy`). However, here is a possible quick implementation from scratch:

```
>>> def factorial(n):
        f = 1
        for k in range(1,n+1):
            f = f*k
        return f

>>> factorial(3)

   6
```

For the computation in the previous example, we now increase n until $2^{n-1}n!\geqslant 10^{16}$.

```
>>> n = 1

>>> while 2**(n-1) * factorial(n) < 10**16:
        n = n+1

>>> n

   15

>>> 2**(n-1) * factorial(n)

   21424936845312000

>>> 2.**(n-1) * factorial(n)

   2.1424936845312e+16
```

(Cubic) splines: piecewise polynomial interpolation

If, given data points $(x_0, y_0), \ldots, (x_n, y_n)$ (also called **knots**) with $x_0 < x_1 < \ldots < x_n$, we connect them via straight lines, then we obtain what is called a **linear spline**. This linear spline interpolates the given points but it is not a polynomial; instead it is a piecewise polynomial (namely, in this case, it is a line on each segment $[x_i, x_{i+1}]$).

A problematic feature of linear splines is their lack of smoothness. Between each linear piece, we usually have a sharp corner at which the spline is not differentiable.

C^n smooth. We say that a function is C^n smooth if its nth derivative exists and is continuous.

For instance, linear splines are C^0 (continuous) but not C^1 (unless the spline is a single line).

Of particular practical importance are **cubic splines** which are piecewise polynomials where each piece is a polynomial of degree 3 (or less).

On each segment $[x_i, x_{i+1}]$, we therefore have 4 degrees of freedom (coming from the 4 coefficients of a cubic polynomial). We need 2 of these to interpolate at the two endpoints. This leaves us with $4 - 2 = 2$ degrees of freedom which we can use to achieve smoothness. This allows us to demand that the first and the second derivative agree with the neighboring pieces. Thus the resulting spline will be C^2 smooth.

A **cubic spline** $S(x)$ through $(x_0, y_0), \ldots, (x_n, y_n)$ with $x_0 < x_1 < \ldots < x_n$ is piecewise defined by n cubic polynomials $S_1(x), \ldots, S_n(x)$ such that $S(x) = S_i(x)$ for $x \in [x_{i-1}, x_i]$. Moreover:

- $S(x)$ interpolates the given points.

This means that $S_i(x_{i-1}) = y_{i-1}$ and $S_i(x_i) = y_i$ for $i \in \{1, \ldots, n\}$. ($2n$ equations)

- $S(x)$ is C^2 smooth (i.e. it has a continuous second derivative).

This means that $S_i'(x_i) = S_{i+1}'(x_i)$ and $S_i''(x_i) = S_{i+1}''(x_i)$ for $i \in \{1, \ldots, n-1\}$. ($2n - 2$ equations)

Note that there are $4n$ degrees of freedom for such a spline $S(x)$, while we only have $2n + (2n - 2) = 4n - 2$ equations. To define a unique spline, we therefore need to impose 2 more constraints. These are usually chosen as **boundary conditions**.

The following are common choices for the **boundary conditions of cubic splines**:

- **natural**: $S_1''(x_0) = S_n''(x_n) = 0$

The resulting splines are simply called **natural cubic splines**.

- **not-a-knot**: $S_1'''(x_1) = S_2'''(x_1)$ and $S_n'''(x_{n-1}) = S_{n-1}'''(x_{n-1})$
- **periodic**: $S_1'(x_0) = S_n'(x_n)$ and $S_1''(x_0) = S_n''(x_n)$ (only makes sense if $y_0 = y_n$)

There are other common choices such **clamped cubic splines** for which the first derivatives at the endpoints are being set ("clamped") to user-specified values.

Comment. The name "natural" comes from the fact that the resulting spline is what one gets if one pins thin (idealized) elastic strips in the position of the given knots.

Comment. The "not-a-knot" condition has the consequence that $S_1 = S_2$ and $S_n = S_{n-1}$ (because cubic polynomials must be equal if they agree up to the third derivative at a point). Hence x_1 and x_{n-1} are no longer "knots" between separate polynomials.

As illustrated in the next example, we can compute splines (with 2 boundary conditions) by spelling out the $4n$ defining equations. We can then solve the resulting linear equations using linear algebra. There are, of course, various clever observations that make this approach more efficient. However, since we have other sights to see on our journey, we will not get into these aspects.

Review. By Taylor's theorem (Theorem 52), applied to a function $f(x)$ around $x=x_0$, we have

$$f(x) = \underbrace{f(x_0) + f'(x_0)(x-x_0) + \cdots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n}_{n\text{th Taylor polynomial}} + \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}}_{\text{error}}.$$

If $f(x)$ is a polynomial of degree n, then $f^{(n+1)}(x)=0$ so that $f(x)$ is equal to its nth Taylor polynomial. In particular, if $f(x)$ is a cubic polynomial then, for any x_0,

$$f(x) = a(x-x_0)^3 + b(x-x_0)^2 + c(x-x_0) + d,$$

with $a=\frac{1}{6}f^{(3)}(x_0)$, $b=\frac{1}{2}f^{(2)}(x_0)$, $c=f'(x_0)$ and $d=f(x_0)$.

Example 105. Determine the natural cubic spline through $(-1,2)$, $(1,0)$, $(2,5)$.

Solution. Let us write the spline as $S(x) = \begin{cases} S_1(x), & \text{if } x\in[-1,1], \\ S_2(x), & \text{if } x\in[1,2]. \end{cases}$

To simplify our life, we expand both S_i around $x=1$ (the middle knot).

$$S_i(x) = a_i(x-1)^3 + b_i(x-1)^2 + c_i(x-1) + d_i.$$

- As noted in the review above, $d_i = S_i(1)$, $c_i = S_i'(1)$ and $b_i = \frac{1}{2}S_i''(1)$. Because $S(x)$ is C^2 smooth, we have $b_1=b_2$, $c_1=c_2$ and $d_1=d_2$. We simply write b, c and d for these values in the sequel.
- $d=0$ because $S_1(1)=S_2(1)=0$.
- $S(x)$ further interpolates the other two points, $(-1,2)$ and $(2,5)$, resulting in the following two equations:

$$\begin{aligned} S_1(-1) &= -8a_1+4b-2c = 2 \\ S_2(2) &= a_2+b+c = 5 \end{aligned}$$

- The natural boundary conditions provide two more equations: (Note that $S_i''(x) = 6a_i(x-1)+2b_i$.)

$$\begin{aligned} S_1''(-1) &= -12a_1+2b = 0 \\ S_2''(2) &= 6a_2+2b = 0 \end{aligned}$$

We use these last two equations to replace $a_1=\frac{1}{6}b$ and $a_2=-\frac{1}{3}b$ in the other two equations in terms of b:

$$\begin{aligned} -8\cdot\frac{1}{6}b+4b-2c &= \frac{8}{3}b-2c = 2 \\ -\frac{1}{3}b+b+c &= \frac{2}{3}b+c = 5 \end{aligned}$$

Solving these two equations in two unknowns, we find $b=3$ and $c=3$.

Consequently, $a_1=\frac{1}{6}b=\frac{1}{2}$ and $a_2=-\frac{1}{3}b=-1$.

Hence, the desired natural cubic spline is

$$S(x) = 3(x-1)+3(x-1)^2+(x-1)^3\begin{cases} \frac{1}{2}, & \text{if } x\in[-1,1], \\ -1, & \text{if } x\in[1,2]. \end{cases}$$