Lecture 6: Polynomial interpolation

Suppose we have only a discrete set of data points that describe an underlying mathematical function. We have seen that we can "fit" a function to the data, but what if we want an approximation to this unknown function that recovers these known values exactly? This introduces the topic of *interpolation*. In particular, we focus on using polynomial functions to capture the known values while also providing an approximation between the data points. One motivation for using polynomials is because they are easy to manipulate and numerically cheap to evaluate.

6.1 Global polynomial interpolation

To begin suppose we know the values of some function $f(x)$ on a set of $n+1$ *different* points. That is, we have a set of points $x_0, \ldots, x_n$ as we know

$$f(x_i) = f_i \quad \text{where} \quad i = 0, \ldots, n.$$

The goal of a *global polynomial interpolation* procedure is to determine a polynomial function of degree n, denoted as $p_n(x)$, such that

$$p_n(x_i) = f_i \quad \text{where} \quad i = 0, \ldots, n,$$

i.e., the functions match at the known data points. We can then use $p_n(x)$ to approximate the function $f(x)$ for any value of x within the interval formed by $x_0, \ldots, x_n$.

Remark 6.1. If we use the polynomial $p(x)$ to approximate the function $f(x)$ outside this interval it is known as *extrapolation*. It is important to remember that extrapolation is inherently unreliable. ⋈

So we want to find this polynomial function given some function data. Effectively, we ensure that the known data points will be *roots* of the polynomial interpolant. This provides $n+1$ unknown parameters for $n+1$ constraints to determine an n^{th} order interpolating polynomial function.

Definition 6.1 (Uniqueness of polynomial interpolant). Given $n+1$ points $x_0, \ldots, x_n$ with corresponding values $f_0, \ldots, f_n$ there is at most one polynomial function $p_n(x)$ of degree less than or equal to n such that

$$p_n(x_i) = f_i \quad \text{where} \quad i = 0, \ldots, n.$$

◀

This statement follows directly from the Fundamental Theorem of Algebra. Suppose we have two polynomials, $p_n(x)$ and $r_n(x)$, that interpolate the same set of data points such that $p_n(x_i) = r_n(x_i) = f_i$ for $i = 0, \ldots, n$. Then their difference is also a polynomial $q_n(x) = p_n(x) - r_n(x)$ that satisfies $q_n(x_i) = 0$ for $i = 0, \ldots, n$. However, since the number of roots of a nonzero polynomial is equal to its degree, it follows that $q_n(x) \equiv 0$. Thus, $p_n(x) = r_n(x)$.

Though the interpolating polynomial is unique, we know that there are many equivalent representations of a polynomial function (like factoring its roots). We have seen previously that in numerical computations it is very important ***how*** we represent and manipulate information in order to maintain accuracy. This is also true in forming a polynomial interpolant.

As a first investigation, we represent the interpolating polynomial in a "standard" form in terms of *monomials*

$$p_n(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \cdots + \alpha_n x^n,$$

which is a linear combination of increasing powers of the variable x. Now we apply the constraints that the interpolating polynomial must satisfy the given function data and the given points $x_0, \ldots, x_n$ to see

$$\begin{aligned}
\alpha_0 + \alpha_1 x_0 + \alpha_2 x_0^2 + \cdots + \alpha_n x_0^n &= f_0 \\
\alpha_0 + \alpha_1 x_1 + \alpha_2 x_1^2 + \cdots + \alpha_n x_1^n &= f_1 \\
\alpha_0 + \alpha_1 x_2 + \alpha_2 x_2^2 + \cdots + \alpha_n x_2^n &= f_2 \\
&\vdots \\
\alpha_0 + \alpha_1 x_n + \alpha_2 x_n^2 + \cdots + \alpha_n x_n^n &= f_n
\end{aligned}$$

or equivalently in matrix form

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix} \quad \text{or} \quad \mathbf{V}\boldsymbol{\alpha} = \mathbf{f}.$$

EXAMPLE 6.1. Say we are given the data points for a function

x	-2	-1	0	1	2
f	-9	-15	-5	-3	39

The corresponding linear system for the polynomial interpolant written in terms of monomials is then

$$\begin{pmatrix} 1 & -2 & 4 & -8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{pmatrix} = \begin{pmatrix} -9 \\ -15 \\ -5 \\ -3 \\ 39 \end{pmatrix}.$$

This a dense, nonsingular matrix that must be solved. We can use an **LU** factorization with partial pivoting to solve this linear system and determine the coefficients of the interpolating polynomial to be

$$(\alpha_0\,,\,\alpha_1\,,\,\alpha_2\,,\,\alpha_3\,,\,\alpha_4)^T = (-5\,,\,4\,,\,-7\,,\,2\,,\,3)^T.$$

and yields the fourth degree interpolating polynomial $p_4(x) = -5 + 4x - 7x^2 + 2x^3 + 3x^4$ for the data. ♦

We know that, computationally, solving such a dense linear system requires $\mathcal{O}(n^3)$ operations to create the matrix factorization. More importantly, is this solution reliable?

Remark 6.2. The matrix $\mathbf{V}$ is known as a *Vandermonde matrix*. In general, this type of matrix is dense and suffers from ill-conditioning. ⋈

In turns out no! So, not only is forming and solving this linear system computationally expensive, we can't even count on it to give a good interpolating polynomial.

As mentioned above there are multiple ways to represent a polynomial. More concretely, we can decide on a particular set of *basis functions* in which to write the interpolating polynomial. The set of monomials (also called the monic basis) is just one particularly simple set of basis functions.

Remark 6.3. Recall from linear algebra that a set of n basis vectors can represent all members of the vector space $\mathbb{R}^n$. Analogously, the space of n^{th} degree polynomial functions $\mathbb{P}^n$ can be represented in terms of a set of polynomial basis functions. ⋈

Thus, if we rewrite the interpolating polynomial in terms of a different set of basis functions we can improve the reliability of the resulting interpolating polynomial.

We will explore two popular alternative forms of the interpolating polynomial. The first is due to Newton and the idea is to write the interpolant in a factored form. Note, because we already know the location of the roots to be $x_0, \ldots, x_n$ this is very straightforward. The Newton basis polynomials are given by

$$N_0(x) = 1, \quad N_1(x) = (x - x_0), \quad N_2(x) = (x - x_0)(x - x_1), \quad \cdots, \quad N_n(x) = \prod_{j=0}^{n-1}(x - x_j).$$

Remark 6.4. Note that the Newton basis and the monic basis both satisfy a *hierarchy* where the first basis function is a constant, the next is linear, then a quadratic, then a cubic, and so on. ⋈

The interpolating polynomial is then written as a combination of the Newton basis polynomials

$$p_n(x) = \sum_{i=0}^{n} \beta_i N_i(x),$$

where the values of β_j are the coefficients that we must determine. Now when we apply the interpolation constraints of the given function data we obtain a linear system of the form:

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & x_1 - x_0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 1 & x_{n-1} - x_0 & \cdots & \prod_{j=0}^{n-2}(x_{n-1} - x_j) & 0 \\ 1 & x_n - x_0 & \cdots & \prod_{j=0}^{n-2}(x_n - x_j) & \prod_{j=0}^{n-1}(x_n - x_j) \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{n-1} \\ \beta_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix}.$$

By changing the basis and how we represented the polynomial now the linear system is lower triangular! Provided all the of the interpolation points $x_0, \ldots, x_n$ are distinct, this matrix is invertible.

For the general case, we use forward substitution to determine the coefficients:

$$\beta_0 = f_0, \quad \beta_1 = \frac{f_1 - f_0}{x_1 - x_0}, \quad \beta_2 = \frac{f_2 - f_0 - \frac{x_2 - x_0}{x_1 - x_0}(f_1 - f_0)}{(x_2 - x_0)(x_2 - x_1)}, \quad \text{etc.}$$

The analytical expressions for the coefficients become increasingly complicated and unwieldy. Notice that the coefficients all involve a ratio of difference(s) between the known function data and the distance(s) between the interpolation points. We can exploit this fact and recursively take differences between function data and interpolation points to build the necessary coefficients, e.g.,

$$f[x_0] = f(x_0) = f_0, \quad f[x_0, x_1] = \frac{f_1 - f_0}{x_1 - x_0}, \quad f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}.$$

Definition 6.2 (Newton divided difference). The k^{th} *divided difference* of f is given by

$$f[x_0, \ldots, x_k] = \frac{f[x_1, \ldots, x_k] - f[x_0, \ldots, x_{k-1}]}{x_k - x_0},$$

with respect to the points $x_0, \ldots, x_k$ ◀

The divided differences gives a straightforward algorithm to determine the coefficients of the interpolating polynomial for the Newton basis. Conveniently, we can arrange the data needed for the interpolating polynomial into a *Newton divided difference table*. This is a convenient organizational technique for the data needed to build this particular form of the interpolating polynomial.

For instance, if we have four data points the structure of the Newton divided difference table is shown at the right. One can actually take any connected path through the table to define the coefficients of the interpolating polynomial. Each path will change the particular value in front of the terms in the Newton basis. But the actual interpolant does not change because the interpolating polynomial is unique! In the figure at the right, we indicate the *typical* path one takes through the divided difference table to construct the Newton interpolating polynomial.

$$
\begin{array}{llll}
x_0 & f_0 & & \\
 & & f[x_0,x_1] & \\
x_1 & f_1 & & f[x_0,x_1,x_2] \\
 & & f[x_1,x_2] & & f[x_0,x_1,x_2,x_3] \\
x_2 & f_2 & & f[x_1,x_2,x_3] \\
 & & f[x_2,x_3] & \\
x_3 & f_3 & &
\end{array}
$$

DEFINITION 6.3 (NEWTON INTERPOLATING POLYNOMIAL). The typical form of the interpolating polynomial created with Newton's strategy is

$$
\begin{aligned}
p_n(x) &= f_0 + f[x_0,x_1](x-x_0) + f[x_0,x_1,x_2](x-x_0)(x-x_1) + \cdots + f[x_0,\ldots,x_n](x-x_0)(x-x_1)\cdots(x-x_{n-1}) \\
&= f_0 + f[x_0,x_1]N_1(x) + f[x_0,x_1,x_2]N_2(x) + \cdots + f[x_0,\ldots,x_n]N_n(x),
\end{aligned}
$$

where $N_i(x)$ is the Newton basis described above with a given data set $(x_i, f(x_i))$ and $i = 0,\ldots,n$. ◀

EXAMPLE 6.2. Use the data set from EXAMPLE 6.1 to create a Newton interpolating polynomial. First, we compute the values for the Newton divided difference table:

x	$f(x)$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot,\cdot]$
-2	-9				
		-6			
-1	-15		8		
		10		-4	
0	-5		-4		3
		2		8	
1	-3		20		
		42			
2	39				

The values we use for the interpolant are highlighted and we find the Newton polynomial to be

$$p_4(x) = -9 - 6(x+2) + 8(x+1)(x+2) - 4x(x+1)(x+2) + 3x(x-1)(x+1)(x+2).$$

As expected, if we expand the Newton interpolating polynomial into monomial form it is identical to the previous computation in the monic basis with the Vandermonde matrix. ♦

The second common tactic to rewrite the interpolating polynomial we cover is due to Lagrange. Here, we use an alternative set of polynomial functions as our basis:

DEFINITION 6.4 (LAGRANGE POLYNOMIALS). Given a set of distinct interpolation points $x_0,\ldots,x_n$ the *Lagrange polynomials* are n^{th} degree polynomials defined as

$$\ell_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}, \quad \text{where} \quad i = 0,\ldots,n.$$

By design the Lagrange polynomials satisfy the identity

$$\ell_i(x_k) = \begin{cases} 1, & \text{if } i = k \\ 0, & \text{if } i \neq k \end{cases}$$

also known as the cardinality property. ◀

Remark 6.5. Notice that the Lagrange basis polynomials ***are not*** hierarchical! Each one of them is a polynomial of degree n. ⋈

What is gained when we move to this more sophisticated set of basis functions? It turns out it is trivial to write the interpolating polynomial in Lagrange form

$$p_n(x) = \sum_{i=0}^{n} \gamma_i \ell_i(x).$$

because of the cardinality property of the functions $\ell_i(x)$. When we apply the interpolation constraints to the interpolating polynomial written in Lagrange form we obtain the system

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \begin{pmatrix} \gamma_0 \\ \gamma_1 \\ \vdots \\ \gamma_{n-1} \\ \gamma_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-1} \\ f_n \end{pmatrix}.$$

Definition 6.5 (Lagrange interpolating polynomial). Given a set of distinct interpolation points $x_0, \ldots, x_n$ with corresponding data $f_0, \ldots, f_n$ then

$$p_n(x) = \sum_{i=0}^{n} f_i \ell_i(x),$$

is the *Lagrange interpolating polynomial*. ◀

Example 6.3. Use the data set from Example 6.1 to create a Lagrange interpolating polynomial. We already have the function data, so once we construct the appropriate Lagrange polynomials we are done:

$$\begin{aligned}
\ell_0 &= \frac{(x+1)(x-0)(x-1)(x-2)}{(-2+1)(-2)(-2-1)(-2-2)} &&= \frac{(x+1)(x-0)(x-1)(x-2)}{24} \\
\ell_1 &= \frac{(x+2)(x-0)(x-1)(x-2)}{(-1+2)(-1)(-1-1)(-1-2)} &&= \frac{(x+2)(x-0)(x-1)(x-2)}{-6} \\
\ell_2 &= \frac{(x+2)(x+1)(x-1)(x-2)}{(0+2)(0+1)(0-1)(0-2)} &&= \frac{(x+2)(x+1)(x-1)(x-2)}{4} \\
\ell_3 &= \frac{(x+2)(x+1)(x-0)(x-2)}{(1+2)(1+1)(1)(1-2)} &&= \frac{(x+2)(x+1)(x-0)(x-2)}{-6} \\
\ell_4 &= \frac{(x+2)(x+1)(x-0)(x-1)}{(2+2)(2+1)(2)(2-1)} &&= \frac{(x+2)(x+1)(x-0)(x-1)}{24}
\end{aligned}$$

Therefore, the Lagrange interpolating polynomial $p_4(x) = \sum_{i=0}^{4} f_i \ell_i(x)$ is

$$\begin{aligned}
p_4(x) = &-9\left[\frac{(x+1)(x-0)(x-1)(x-2)}{24}\right] - 15\left[\frac{(x+2)(x-0)(x-1)(x-2)}{-6}\right] - 5\left[\frac{(x+2)(x+1)(x-1)(x-2)}{4}\right] \\
&- 3\left[\frac{(x+2)(x+1)(x-0)(x-2)}{-6}\right] + 39\left[\frac{(x+2)(x+1)(x-0)(x-1)}{24}\right]
\end{aligned}$$

Again, if we expand the Lagrange representation into monic form we recover the same polynomial. ⧫

Now we have several strategies to create the polynomial interpolant. Next, we address an efficient way to evaluate the resulting polynomial function. Consider the monic form of a polynomial $p_n(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \cdots + \alpha_n x^n$. Naively, for a given value of x, one could evaluate each of the polynomial powers, scale with the appropriate constant α, and add them together to obtain the result. This amounts to $\mathcal{O}(n^2)$ complexity as well as being prone to cancellation errors because the power of x might be very large. To reduce the computational cost and sensitivity of the polynomial evaluation, we use a *factoring trick*:

Definition 6.6 (Horner's rule). To reduce the computation cost, we evaluate the monic polynomial in the form

$$p_n(x) = \alpha_0 + x\left(\alpha_1 + x\left(\alpha_2 + x\left(\alpha_3 + \cdots + x\left(\alpha_{n-1} + x\alpha_n\right)\cdots\right)\right)\right)$$

which requires only n multiplications and n additions, i.e. $\mathcal{O}(n)$. It also reduces cancellation errors. ◀

Algorithm 6.1: *Horner's Rule*: Efficiently evaluate a polynomial at a given value

Procedure Horner's Rule
Input: $\boldsymbol{\alpha}$, x, n `// vector of coefficients, value, and polynomial degree`
$p \leftarrow \alpha_n$
for $k = n-1$ **to** 0 **step** -1 **do**
  $p \leftarrow p * x + \alpha_k$
Output: p `// evaluation of` $p_n(x)$
End Procedure Horner's Rule

Remark 6.6. Newton's form of the interpolating polynomial can also be rewritten with Horner's rule as

$$p_n(x) = f_0 + (x-x_0)(f[x_0,x_1] + (x-x_1)(f[x_0,x_1,x_2] + \cdots + (x-x_{n-2})(f[x_0,\cdots,x_{n-1}] + f[x_0,\cdots,x_n](x-x_{n-1})\cdots))$$

and improve performance provided one gives the procedure the interpolation nodes $x_0,\ldots,x_n$ as well. ⋈

Remark 6.7. Here we summarize the computational costs to construct the different forms of the polynomial interpolant as well as its evaluation:

	Computing interpolation coefficients	Evaluation of $p_n(x)$
Monomial form	$\mathcal{O}(n^3)$	$\mathcal{O}(n)$
Newton form	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Lagrange form	0	$\mathcal{O}(n^2)$

Notice, that the work can be reduced by considering different basis representations but it has to be done at some point. This highlights an overarching principle in numerical mathematics. There is computational work that must be performed, but we can concentrate it in different areas depending on the formulation. But remember, *there is no free lunch*! ⋈

Remark 6.8. Unfortunately, the current form of the Lagrange interpolating polynomial **cannot** be factored according to Horner's rule. However, it is possible to recast the Lagrange basis functions into a form that can exploit Horner's rule. This is known as the *barycentric formulation*. ⋈

Finally, we discuss the the error of the global interpolating polynomial in approximating a particular function $f(x)$. Note, we **will not** discuss convergence because it is very mathematically technical. Instead, we will only examine the pointwise error between a function and an n^{th} degree polynomial interpolant.

Definition 6.7 (Pointwise error). Let $[a,b]$ be an interval on which $f(x)$ is $(n+1)$-times differentiable. Further, we assume that the interpolation nodes $x_0,\ldots,x_n$ as well as a value x are all contained within $[a,b]$. Then the *pointwise error* between the function and a global polynomial interpolant for values $f_0,\ldots,f_n$ is

$$E_n(x) = f(x) - p_n(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_n)}{(n+1)!} f^{(n+1)}(\xi) = \frac{\Psi_n(x)}{(n+1)!} f^{(n+1)}(\xi),$$

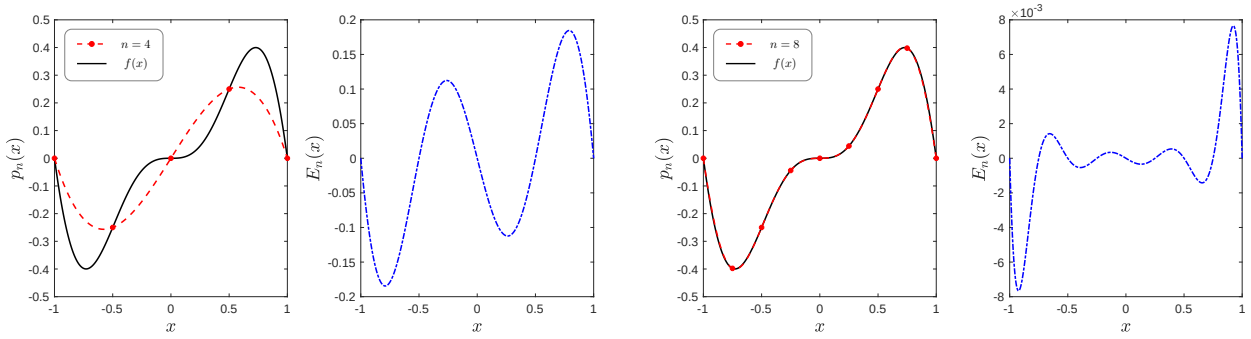
for some value of $\xi \in [a,b]$ and we introduce a compact notation $\Psi_n(x) = (x-x_0)(x-x_1)\cdots(x-x_n)$. ◀

From the construction of the interpolating polynomial we know that the error at the interpolation nodes is zero, i.e. $E_n(x_i) = 0$ for $i = 0,\ldots,n$. Note that this pointwise error tends to be very oscillatory. Also, if the derivative of the function $f^{(n+1)}(x)$ is nicely bounded on $[a,b]$ then the error can be estimated. However, we can encounter issues if $f^{(n+1)}(x)$ grows faster than the factorial $(n+1)!$ or if $\Psi_n(x)$ is large.

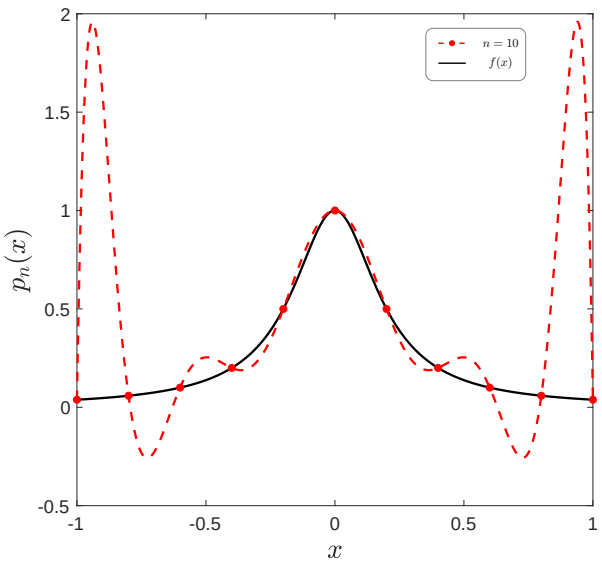
EXAMPLE 6.4. Examine the pointwise error of a fourth degree and eighth degree interpolating polynomial of the function $f(x) = x^2 \sin(\pi x)$ on the interval $[-1, 1]$ with uniformly spaced points. For the two interpolating polynomials the equally spaced nodes will be

$$\{x_i\}_{i=0}^{4} = \left\{-1, -\frac{1}{2}, 0, \frac{1}{2}, 1\right\} \quad \text{and} \quad \{x_i\}_{i=0}^{8} = \left\{-1, -\frac{3}{4}, -\frac{1}{2}, -\frac{1}{4}, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\right\}.$$

The higher order derivatives of the function $f(x)$ are nicely bounded and the pointwise error is dominated by the factorial term. Therefore, we expect the pointwise error to behave well in that its overall magnitude should shrink if we construct an interpolant with more points.



We examine this trend in the plots of the interpolating polynomial, the function $f(x)$, and the pointwise error above on the left for $n = 4$ and the right for $n = 8$. We observe that the pointwise error indeed becomes smaller with more interpolation nodes. Although, notice that the error remains the largest near the endpoints of the interval. ♦



Is this always the case that more *uniformly* spaced interpolation points yields a more accurate global interpolating polynomial to a function? No, this is not the case. To demonstrate this we consider the function

$$f(x) = \frac{1}{1 + 25x^2}$$

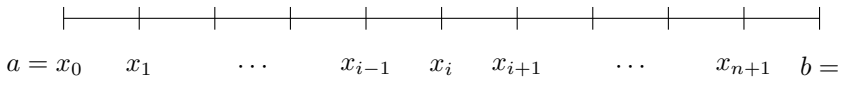
We see in the figure at right that near the endpoints the interpolating polynomial is a very poor approximation of the function $f(x)$ although in the interior it looks quite good. This observation is confirmed with the pointwise error where the interior is significantly more accurate than close to the boundary. The ringing near the boundary are *Runge phenomena*. One cause is the natural, oscillatory behaviors of a high order polynomial function. Next, we examine how to alleviate this spurious ringing and obtain better accuracy with interpolation near the boundary.

6.2 Piecewise polynomial interpolation

Previously, we constructed a global polynomial function, $p_n(x)$, to interpolate a function at a given set of data points. As we increase the number of interpolation points, the global polynomial becomes possesses a higher degree. In order to capture the given data this global polynomial oscillates, sometimes a great deal, in between the interpolation points even when the function $f(x)$ we want to approximate is *nonoscillatory*. For instance, recall the interpolation of the function $f(x) = (1 + 25x^2)^{-1}$.

To remove this oscillatory behavior of a high degree polynomial interpolant we alter the interpolation strategy and move from a global perspective to a **local** one. The motivation for this change comes from the observation that we get a poor global approximation with a low degree polynomial but no wild oscillations, whereas a high degree polynomial is accurate in parts of the interval but can have wild oscillations in another part of the interval.

Suppose we still have an interval $[a, b]$ and a set of data points $\{x_i\}_{i=0}^n$. For simplicity we assume that these data points are uniformly spaced. Now, we will *subdivide* the interval $a = x_0 < x_1 < \cdots < x_n = b$



This produces n subintervals $[x_{i-1}, x_i]$ where $i = 1, \ldots, n$. Within each of these subintervals we then use a low degree polynomial to approximate the function $f(x)$. The global interpolant is then the union of these local, low degree polynomial approximations on the nonoverlapping subintervals.

But how do we construct these local approximations? We constrain this procedure with the requirement that the global approximation is *smooth* by enforcing continuity as well as continuity of some number of its derivatives. This is the principle behind constructing a *spline*.

Definition 6.8 (Spline interpolation). A function $s_m(x)$ is a spline of degree $2m+1$ when $s_m(x)$ is a union of piecewise polynomials of degree $2m+1$ such that $s_m(x)$ **and** its first $2m$ derivatives are continuous. When the spline is constructed such that $s_m(x_i) = f(x_i)$, $i = 0, 1, \ldots, n$ it is a *spline interpolant*. ◀

Remark 6.9. Two popular forms are the *linear spline* when $m = 0$ and the *cubic spline* when $m = 1$. ⋈

First, we discuss the linear spline interpolant. As such, suppose we have the set of data points $\{(x_i, f_i)\}_{i=0}^n$ that we want to interpolate. We produce subintervals $[x_{i-1}, x_i]$ and on each subinterval we approximate the function with a straight line. Thus, the linear spline $s_1(x)$ is composed of n line segments with the form

$$s_1(x) = a_i + b_i(x - x_{i-1}), \quad \text{where} \quad x_{i-1} \le x \le x_i.$$

We must determine the $2n$ coefficients of the different line segments. We enforce the continuity of the linear spline between the subintervals, which gives the conditions

$$a_i + b_i(x - x_{i-1}) = a_{i+1}, \quad \text{where} \quad i = 1, \ldots, n-1.$$

This leaves $2n - (n-1) = n+1$ degrees of freedom. To close the system and explicitly determine the spline coefficients we remove these degrees of freedom with the requirement that the linear spline take the given function values at the subinterval boundaries

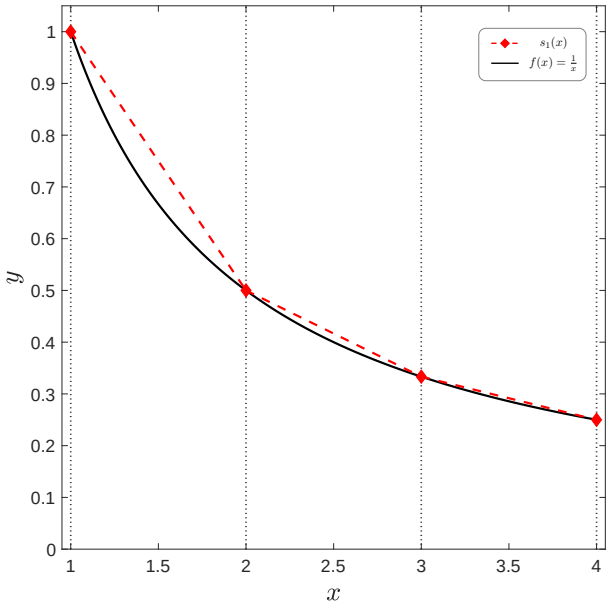
$$s_1(x_i) = f_i, \quad \text{where} \quad i = 0, \ldots, n.$$

This yields a general form for the linear spline in each subinterval

Definition 6.9 (Linear spline interpolant). Given a set of data points $\{(x_i, f_i)\}_{i=0}^n$ in the interval $[a, b]$ where the values $\{x_i\}_{i=0}^n$ are distinct, the linear interpolating spline can be written as

$$s_1(x) = f_{i-1} + \frac{f_i - f_{i-1}}{x_i - x_{i-1}}(x - x_{i-1}), \quad \text{where} \quad x_{i-1} \le x \le x_i$$

and $i = 1, \ldots, n$. ◀



Let us examine how the linear spline performs.

EXAMPLE 6.5. Construct a linear spline for the data points

$$\{(x_i, f_i)\}_{i=0}^{3} = \left\{(1,1)\ ,\ \left(2,\frac{1}{2}\right)\ ,\ \left(3,\frac{1}{3}\right)\ ,\ \left(4,\frac{1}{4}\right)\right\}$$

and compare it to the function $f(x) = x^{-1}$. We divide the interval $[1,4]$ into three subintervals and compute the linear spline to be

$$s_1(x) = \begin{cases} 1 + \frac{\frac{1}{2}-1}{2-1}(x-1) = 1 - \frac{1}{2}(x-1), & \text{if } 1 \le x \le 2 \\ \frac{1}{2} + \frac{\frac{1}{3}-\frac{1}{2}}{3-2}(x-2) = \frac{1}{2} - \frac{1}{6}(x-2), & \text{if } 2 \le x \le 3 \\ \frac{1}{3} + \frac{\frac{1}{4}-\frac{1}{3}}{4-3}(x-3) = \frac{1}{3} - \frac{1}{12}(x-3), & \text{if } 3 \le x \le 4 \end{cases}$$

We plot the linear spline and visually compare the approximation to the known function $f(x) = \frac{1}{x}$.

Additionally, we use vertical dotted lines to distinguish between the different subintervals. We see that the linear spline is, essentially, a systematic strategy to **connect-the-dots** between the different known function points. Note, we used very few subintervals to demonstrate the idea behind the linear spline. Thus, we see that the approximation is quite crude. ♦

Locally, within each of the subintervals, we use a linear polynomial to approximate the function. Therefore, we have the same pointwise error estimate from before inside each subinterval. We use this to state an error estimate for the linear spline **globally**. To do so, we denote the size of each subinterval to be

$$h_i = x_i - x_{i-1}$$

where $i = 1, \ldots, n$.

DEFINITION 6.10 (GLOBAL ERROR OF LINEAR SPLINE). For a function $f(x)$ that has two continuous derivatives on the interval $[a,b]$ the *global error* of the linear spline is bounded by

$$E_n^{s_1} = \max_{x\in[a,b]} |f(x) - s_1(x)| \le \frac{h^2}{8} \max_{x\in[a,b]} |f''(x)| \sim \mathcal{O}(h^2),$$

where $h = \max_{i=1,\ldots,n} \{h_i\}$ is the size of the largest subinterval. ◄

We see that the global error shrinks quadratically as we add more subintervals for the linear spline. This means that if we double the number of subintervals the error reduces by a factor of four. So, the linear spline is a second order method to approximate a suitably smooth function $f(x)$.

EXAMPLE 6.6. Verify that the linear spline converges at second order to the function $f(x) = \frac{1}{x}$ on the interval $[1,4]$. We note that the function is twice differentiable in the given interval, so the global error estimate is valid. We compute the error for increasing values of n on a uniform set of points $\{x_i\}_{i=0}^{n}$. In turn, this increases the number of subintervals and decreases the value of h.

Let us examine the ratio of two errors where the number of points differ by a factor of two. From above we have the global error estimates for the linear spline

$$E_n^{s_1} \approx h^2, \quad \text{and} \quad E_{2n}^{s_1} \approx \left(\frac{h}{2}\right)^2 = \frac{h^2}{2^2} \quad \Rightarrow \quad \frac{E_n^{s_1}}{E_{2n}^{s_1}} \approx 2^2$$

n	$E_n^{s_1}$	EOC
2	0.25000	—
4	0.08547	1.54
8	0.02666	1.68
16	0.00755	1.82
32	0.00202	1.90
64	0.00052	1.96

If we take the logarithm and manipulate we see that this error ratio can reveal the experimental order of convergence (EOC) for the approximation

$$EOC \equiv \frac{\log\left(\frac{E_n^{s1}}{E_{2n}^{s1}}\right)}{\log(2)} \approx 2$$

We collect values of the error for different numbers of subintervals as well as the EOC in the table above. Notice that the EOC manipulation above was always done approximately. That is why for very coarse resolutions of the linear spline the EOC is worse than for fine resolutions where it becomes closer to the expected value. ♦

We see that this *connect-the-dots* style of linear spline works but straight lines are a poor approximation for curves. Therefore, we can improve the accuracy of the spline interpolant by using a higher degree polynomial within each subinterval.

Next, we examine *cubic splines* where the function is approximated on the interval $[a, b]$ within each subinterval with a cubic polynomial with the general form

$$s_3(x) = a_i + b_i\left(\frac{x - x_{i-1}}{h_i}\right) + c_i\left(\frac{x - x_{i-1}}{h_i}\right)^2 + d_i\left(\frac{x - x_{i-1}}{h_i}\right)^3, \quad \text{where} \quad x_{i-1} \le x \le x_i$$

and $i = 1, \ldots, n$. So we must determine $4n$ coefficients to define the cubic spline. We enforce continuity of the spline as well as its first and second derivatives

$$\lim_{x \to x_i^+} s_3(x) = \lim_{x \to x_i^-} s_3(x), \quad \lim_{x \to x_i^+} s_3'(x) = \lim_{x \to x_i^-} s_3'(x), \quad \text{and} \quad \lim_{x \to x_i^+} s_3''(x) = \lim_{x \to x_i^-} s_3''(x)$$

for $i = 1, \ldots, n-1$ in the interval $[a, b]$. The continuity of the spline provides $3(n-1)$ conditions, so we have a remaining $4n - 3(n-1) = n + 3$ degrees of freedom. The interpolation requirements $s_3(x_i) = f_i$, $i = 0, \ldots, n$ are an additional $n + 1$ conditions. To remove the remaining two degrees of freedom we apply two *boundary conditions*. The simplest of which are the following:

Definition 6.11 (Natural boundary condition). We take interpolating cubic spline to satisfy

$$s_3''(x_0) = 0 \quad \text{and} \quad s_3''(x_n) = 0,$$

at the boundaries of the interval $[a, b]$. These boundary conditions are *natural* in the sense that the spline is assumed to be a straight line outside the interval and its second derivative vanishes. ◄

Remark 6.10. There are other types of boundary conditions available to "close" the spline. However, they often require additional knowledge about the derivative(s) of the function $f(x)$. ⋈

Remark 6.11. There are many strategies to determine the coefficients of the cubic spline interpolant. In the following, we present a construction strategy *that is different than the book*. ⋈

In order to build a cubic spline we first assume we have a set of n data points $\{(x_i, f_i)\}_{i=0}^n$ with the corresponding subintervals $[x_{i-1}, x_i]$, $i = 1, \ldots, n$. Next, we consider the second derivative of the spline $s_3''(x)$ that we know to be a linear polynomial within each of the subintervals. Further, we say that the second derivative of the cubic spline evaluates to $n + 1$ *yet to be determined* values

$$s_3''(x_i) = M_i, \quad \text{where} \quad i = 0, \ldots, n.$$

For this discussion we consider the arbitrary interval $[x_{i-1}, x_i]$. We impose that $s_3''(x)$ is continuous at the internal nodes to find

$$s_3''(x) = M_{i-1}\frac{(x_i - x)}{h_i} + M_i\frac{(x - x_{i-1})}{h_i}, \quad \text{in} \quad x_{i-1} \le x \le x_i$$

where $i = 1, \ldots, n$. We integrate twice to find the expression for the first derivative and the cubic spline

$$s_3'(x) = -M_{i-1}\frac{(x - x_i)^2}{2h_i} + M_i\frac{(x - x_{i-1})^2}{2h_i} + \gamma_{i-1}, \quad \text{in} \quad x_{i-1} \le x \le x_i$$

$$s_3(x) = M_{i-1}\frac{(x_i - x)^3}{6h_i} + M_i\frac{(x - x_{i-1})^3}{6h_i} + \gamma_{i-1}(x - x_{i-1}) + \tilde{\gamma}_{i-1}, \quad \text{in} \quad x_{i-1} \le x \le x_i$$

To eliminate the $2n$ constants of integration, γ_{i-1} and $\tilde{\gamma}_{i-1}$ with $i = 1, \ldots, n$, we apply the interpolation conditions as well as the continuity of the spline $s_3(x)$

$$s_3(x_0) = f_0, \quad s_3(x_n) = f_n, \quad \text{and} \quad \lim_{x \to x_i^+} s_3(x) = \lim_{x \to x_i^-} s_3(x) = f_i$$

for $i = 1, \ldots, n-1$. From this we determine that

$$\gamma_{i-1} = \frac{f_i - f_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1}), \quad \tilde{\gamma}_{i-1} = f_{i-1} - M_{i-1}\frac{h_i^2}{6}, \quad \text{where} \quad i = 1, \ldots, n.$$

Substituting these values and enforcing the continuity of the first derivative of the cubic spline

$$\lim_{x \to x_i^+} s_3'(x) = \lim_{x \to x_i^-} s_3'(x),$$

gives us (after some algebraic manipulation)

$$\frac{h_i}{6}M_{i-1} + \frac{h_i}{3}M_i + \frac{f_i - f_{i-1}}{h_i} = -\frac{h_{i+1}}{3}M_i - \frac{h_{i+1}}{6}M_{i+1} + \frac{f_{i+1} - f_i}{h_{i+1}}, \quad \text{where} \quad i = 1, \ldots, n-1.$$

We separate the *still unknown* values of M_i from the known values and gather terms to find

$$\frac{h_i}{6}M_{i-1} + \frac{h_{i+1} + h_i}{6}M_i + \frac{h_{i+1}}{6}M_{i+1} = \frac{(f_{i+1} - f_i)}{h_{i+1}} - \frac{(f_i - f_{i-1})}{h_i}.$$

Then, we multiply through by the value $6/(h_{i+1} + h_i)$ and introduce the constants

$$\mu_i = \frac{h_i}{h_{i+1} + h_i}, \quad \sigma_i = \frac{h_{i+1}}{h_{i+1} + h_i}, \quad \text{and} \quad \eta_i = \frac{6}{h_{i+1} + h_i}\left(\frac{(f_{i+1} - f_i)}{h_{i+1}} - \frac{(f_i - f_{i-1})}{h_i}\right),$$

where $i = 1, \ldots, n-1$. This reveals that the problem to find the cubic spline coefficients is a *tridiagonal linear system*

$$\begin{pmatrix} \mu_1 & 2 & \sigma_1 & 0 & \cdots & \cdots & 0 \\ 0 & \mu_2 & 2 & \sigma_2 & 0 & \cdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 0 & \mu_{n-2} & 2 & \sigma_{n-2} & 0 \\ 0 & \cdots & \cdots & 0 & \mu_{n-1} & 2 & \sigma_{n-1} \end{pmatrix} \begin{pmatrix} M_0 \\ M_1 \\ \vdots \\ M_{n-1} \\ M_n \end{pmatrix} = \begin{pmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_{n-2} \\ \eta_{n-1} \end{pmatrix}.$$

This is a system of $n+1$ equations for $n-1$ unknown values. *Finally*, we impose the general boundary conditions $s_3''(x_0) = M_0 = f''(x_0)$ and $s_3''(x_n) = M_n = f''(x_n)$ that account for the curvature of the function. This yields a square system of equations

$$\begin{pmatrix} 2 & \sigma_1 & 0 & \cdots & 0 \\ \mu_2 & 2 & \sigma_2 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \mu_{n-2} & 2 & \sigma_{n-2} \\ 0 & \cdots & 0 & \mu_{n-1} & 2 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-2} \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} \eta_1 - M_0\mu_1 \\ \eta_2 \\ \vdots \\ \eta_{n-2} \\ \eta_{n-1} - M_n\sigma_{n-1} \end{pmatrix}.$$

For natural boundary conditions $M_0 = M_n = 0$, otherwise we require some information about the second derivative of the function we want to interpolate.

Remark 6.12. The system is nonsingular because $\mu_i, \sigma_i < 1$, so the tridiagonal matrix is strongly diagonally dominant. In fact this linear system is symmetric positive definite. Therefore, we can apply Gauss-Seidel or conjugate gradient to solve it. ⋈

Once we solve this square, nonsingular linear system we have all the necessary information for our cubic spline, even though it might not seem like it. We collect the pieces of the interpolating cubic spline:

DEFINITION 6.12 (NATURAL CUBIC SPLINE INTERPOLANT). Given the coefficients $\{M_i\}_{i=0}^n$ found by solving the tridiagonal system with natural boundary conditions then the *cubic spline interpolant* is given by

$$s_3(x) = M_{i-1}\frac{(x_i - x)^3}{6h_i} + M_i\frac{(x - x_{i-1})^3}{6h_i} + \gamma_{i-1}(x - x_{i-1}) + \tilde{\gamma}_{i-1}, \quad \text{in} \quad x_{i-1} \le x \le x_i$$

with the coefficients

$$\gamma_{i-1} = \frac{f_i - f_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1}), \quad \tilde{\gamma}_{i-1} = f_{i-1} - M_{i-1}\frac{h_i^2}{6},$$

where $i = 1, \ldots, n$. ◀

Oy, that was a lot work! But now we have a more sophisticated spline interpolant for a function $f(x)$.

EXAMPLE 6.7. Construct a natural cubic spline interpolant for the data points

$$\{(x_i, f_i)\}_{i=0}^3 = \left\{(1,1)\ ,\ \left(2, \frac{1}{2}\right)\ ,\ \left(3, \frac{1}{3}\right)\ ,\ \left(4, \frac{1}{4}\right)\right\},$$

and compare it to the function $f(x) = x^{-1}$. From the natural boundary conditions we know that $M_0 = M_3 = 0$ and because the nodes are uniform we have that $\mu_i = \sigma_i = \frac{1}{2}$ and the right hand side values are

$$\eta_1 = \frac{6}{2}\left(\frac{1}{3} - \frac{1}{2} - \left(\frac{1}{2} - 1\right)\right) = 3\left(-\frac{1}{6} + \frac{1}{2}\right) = 1, \quad \eta_2 = \frac{6}{2}\left(\frac{1}{4} - \frac{1}{3} - \left(\frac{1}{3} - \frac{1}{2}\right)\right) = 3\left(-\frac{1}{12} + \frac{1}{6}\right) = \frac{1}{4}.$$

This simplifies the tridiagonal matrix into a the 2×2 linear system

$$\begin{pmatrix} 2 & \frac{1}{2} \\ \frac{1}{2} & 2 \end{pmatrix}\begin{pmatrix} M_1 \\ M_2 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{1}{4} \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} M_1 \\ M_2 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix}.$$

We also compute the other necessary constants needed for the cubic spline

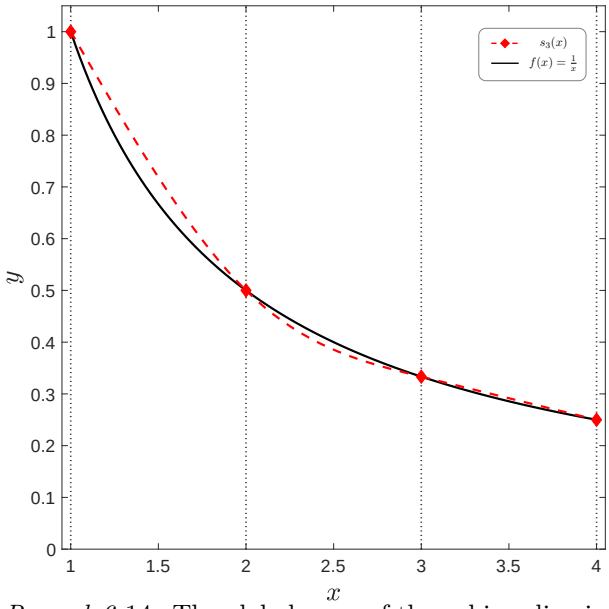
$$\begin{aligned}
&i = 1: \quad \gamma_0 = f_1 - f_0 - \frac{1}{6}(M_1 - M_0) = \frac{1}{2} - 1 - \frac{1}{6}\left(\frac{1}{2} - 0\right) = -\frac{7}{12}, \quad \tilde{\gamma}_0 = f_0 - M_0\frac{h_i^2}{6} = 1 - 0 = 1 \\
&i = 2: \quad \gamma_1 = -\frac{1}{12}, \quad \tilde{\gamma}_1 = \frac{5}{12} \\
&i = 3: \quad \gamma_2 = -\frac{1}{12}, \quad \tilde{\gamma}_2 = \frac{5}{12}
\end{aligned}$$

We then assemble the complete cubic spline interpolant

$$s_3(x) = \begin{cases} \frac{1}{12}(x-1)^3 - \frac{7}{12}(x-1) + 1, & \text{if } 1 \le x \le 2 \\ -\frac{1}{12}(x-3)^3 - \frac{1}{12}(x-2) + \frac{5}{12}, & \text{if } 2 \le x \le 3 \\ -\frac{1}{12}(x-3) + \frac{1}{3}, & \text{if } 3 \le x \le 4 \end{cases}$$

We plot the cubic interpolating spline on the left at the top of the next page. ⧫

Remark 6.13. Note that in the example we used uniform nodes, but the splines (either linear or cubic) can use nonuniform nodes to define the subintervals. In the case of the cubic spline interpolant it simply makes the linear system to determine the coefficients slightly more complicated, but still solvable. ⋈



We see from the previous example that the coarse cubic spline interpolant resembles the linear spline interpolant for the function $f(x) = x^{-1}$. So what advantage does the more sophisticated cubic spline interpolant give us? In general, the local error within each subinterval will be smaller because the local cubic function can take the curvature of the function into account. Therefore, we can create a bound for the global error of the cubic spline interpolant.

DEFINITION 6.13 (GLOBAL ERROR OF CUBIC SPLINE). If $s_3(x)$ is an interpolatory cubic spline constructed with natural boundary conditions then its *global error* is bounded by

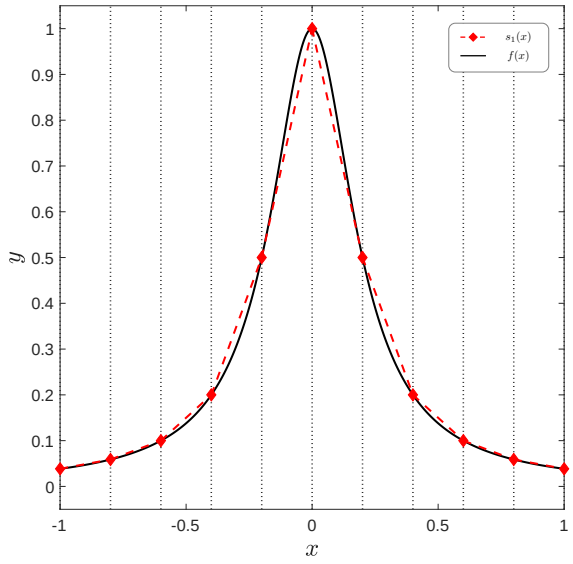
$$E_n^{s_3} = \max_{x\in[a,b]} |f(x) - s_3(x)| \leq \frac{5h^4}{384} \max_{x\in[a,b]} |f^{(4)}(x)| \sim \mathcal{O}(h^4)$$

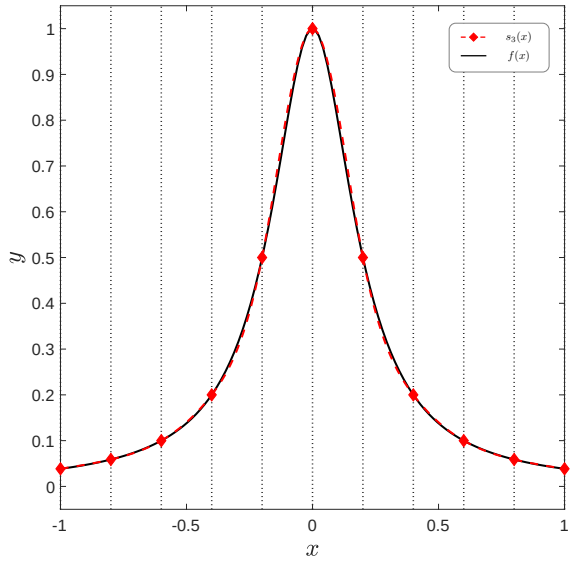
provided the function $f(x)$ has four continuous derivatives. ◀

Remark 6.14. The global error of the cubic spline interpolant also depends on the boundary conditions used to create it. In fact, the accuracy and convergence rate of the cubic spline can be improved if boundary conditions that account for derivative information of the underlying function $f(x)$ are used. ⋈

We see that creating a spline interpolant with higher degree polynomials increases the global accuracy as well as increases how quickly the spline interpolant converges to the function $f(x)$. In fact, we see that if we double the number of subintervals the global error of the cubic spline interpolant reduces by a factor of sixteen. So, the cubic spline is a fourth order method to approximate a suitably smooth function $f(x)$.

But what about our original goal of *removing* the Runge phenomena which can occur for the global interpolation strategy? We revisit creating an interpolant of the function $f(x) = (1 + 25x^2)^{-1}$ but this time use a linear spline (show below on the left) and a cubic spline (shown below on the right) each using $n = 10$ subintervals with a *uniform* size. Again, we delineate between the subintervals using dotted lines.





We see that the piecewise interpolation methods **both** avoid the generation of those spurious oscillations near the boundaries of the interval. Visually, we see that for the function $f(x) = (1 + 25x^2)^{-1}$, which possesses a larger amount of curvature, is better approximated by the cubic spline for the same number of subintervals.