# Polynomial Interpolation

Atonu Roy Chowdhury,
ext.atonu.roy@bracu.ac.bd

February 28, 2026

## 1 Representing functions on computers

### 1.1 How do we represent functions on a computer?

Computers store finite data, so to represent a function $f$ we usually replace it by a finite object: numbers, vectors, coefficients, or samples.

**Polynomials are a natural "finite" representation.** If $f$ is a polynomial of degree $n$,

$$f(x) = p_n(x) = a_0 + a_1 x + \cdots + a_n x^n, \tag{2.1}$$

then we only need to store the $n+1$ coefficients $a_0, \ldots, a_n$. Differentiation and integration are also easy:

$$p_n'(x) = a_1 + 2a_2 x + \cdots + n a_n x^{n-1}, \qquad \int p_n(x)\,dx = a_0 x + \frac{a_1}{2}x^2 + \cdots + \frac{a_n}{n+1}x^{n+1} + C.$$

**Intuition 1.1: Why polynomials show up everywhere**

Polynomials are the "assembly language" of numerical analysis:

- they are easy to store and evaluate,
- they are closed under algebraic operations (addition/multiplication),
- and smooth functions locally resemble polynomials (via Taylor series).

The main question is not whether polynomials are convenient, but how to choose a good one.

### 1.2 Approximating a general function by a polynomial

For continuous functions on a closed interval, polynomial approximation is always possible in principle.

**Theorem 1.1: Weierstrass Approximation Theorem (1885)**

For any $f \in C([0,1])$ and any $\varepsilon > 0$, there exists a polynomial $p$ such that

$$\max_{0 \le x \le 1} |f(x) - p(x)| \le \varepsilon.$$

The proof is outside the scope of this note, so we'll skip it. If you're interested, the proof uses a special class of polynomials, called Bernstein polynomials.

**Intuition 1.2: What Weierstrass guarantees (and what it does not)**

This theorem says: if $f$ is continuous on a bounded interval, polynomials are dense in the space of continuous functions. It does *not* tell us how to find the polynomial efficiently, nor does it guarantee that a naive method will be stable.

**Anecdote 1.1: When polynomials are not enough**

If $f$ is not continuous, then polynomials cannot model jump discontinuities well. If $f$ has asymptotic behaviour or singularities on the interval (e.g., a pole like $1/x$ near 0), then a single polynomial is often a poor model. In those cases, rational functions (ratios of polynomials) are a better language, but that theory is beyond our scope here.

**Interpolation: matching the function at finitely many points** In this chapter our main construction is **interpolation**: we choose nodes $x_0, \ldots, x_n$ and enforce

$$p_n(x_i) = f(x_i), \qquad i = 0, 1, \ldots, n.$$

Sometimes we also match derivatives (this leads to Hermite interpolation). In Chapter 6 we will see an alternative approach, appropriate for noisy data, where the overall error is minimised without requiring exact matching at selected points.

**Intuition 1.3: Interpolation vs. best fit**

Interpolation forces the polynomial to hit the data exactly at specified points. This is ideal when the data are exact (e.g., values from a known formula). If the data are noisy measurements, exact matching can be harmful; then we prefer least-squares fitting.

## 2 Taylor series: interpolation at one node (with derivatives)

A truncated Taylor series is the simplest interpolating polynomial because it uses only one node $x_0$, but compensates by also matching derivatives.

**Deriving the Taylor polynomial** Start from a general polynomial in powers of $(x - x_0)$:

$$f(x) \approx a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \cdots .$$

Differentiate term-by-term and evaluate at $x = x_0$:

$$f(x_0) = a_0, \qquad f'(x_0) = a_1, \qquad f''(x_0) = 2!a_2, \qquad f^{(3)}(x_0) = 3!a_3, \ \ldots$$

So $a_k = \dfrac{f^{(k)}(x_0)}{k!}$, and we obtain the degree-$n$ Taylor polynomial

$$p_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k.$$

**Example 2.1: Calculating $\sin(0.1)$ to 6 s.f. by Taylor series**

Approximate using the Taylor series about $x_0 = 0$:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots .$$

Evaluating successive truncations at $x = 0.1$:

$$1 \text{ term: } \ \sin(0.1) \approx 0.1,$$

$$\text{2 terms: } \sin(0.1) \approx 0.1 - \frac{0.1^3}{6} = 0.099833\ldots,$$

$$\text{3 terms: } \sin(0.1) \approx 0.1 - \frac{0.1^3}{6} + \frac{0.1^5}{120} = 0.09983341\ldots.$$

The next term is $-\dfrac{0.1^7}{7!} \approx -2 \times 10^{-11}$, so it does not affect the answer to 6 s.f.

### Taylor's theorem and the Lagrange remainder

**Theorem 2.1: Taylor's Theorem**

Let $f$ be $n+1$ times differentiable on $(a,b)$ and assume $f^{(n+1)}$ is continuous on $[a,b]$. If $x, x_0 \in [a,b]$, then there exists $\xi$ between $x_0$ and $x$ such that

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x-x_0)^k + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}.$$

The sum is called the Taylor polynomial of degree $n$, and the last term is the **Lagrange form of the remainder**. Note that the unknown number $\xi$ depends on $x$.

**Example 2.2: Using the Lagrange remainder to bound an error (and comparing with the true error)**

For $f(x) = \sin x$, the Taylor polynomial that we used earlier,

$$x - \frac{x^3}{3!} + \frac{x^5}{5!},$$

is actually the degree-6 Taylor polynomial (since the $x^6$ coefficient is 0), so let us denote it by $p_6$. Then $f^{(7)}(x) = -\sin x$, and Taylor's theorem gives

$$|f(0.1) - p_6(0.1)| = \left|\frac{f^{(7)}(\xi)}{7!}(0.1)^7\right| = \frac{(0.1)^7}{5040}\,|\sin(\xi)|, \qquad \xi \in (0, 0.1).$$

Since $|\sin(\xi)| \le 1$, we obtain the rigorous bound

$$|f(0.1) - p_6(0.1)| \le \frac{(0.1)^7}{5040} \approx 1.984 \times 10^{-11}.$$

**Comparison with the true error.** Using $\sin(0.1) \approx 0.09983341664682815$ and

$$p_6(0.1) = 0.1 - \frac{0.1^3}{6} + \frac{0.1^5}{120} \approx 0.0998334166468254,$$

the actual error is

$$|\sin(0.1) - p_6(0.1)| \approx 1.983 \times 10^{-11},$$

which is extremely close to the bound (so the estimate is tight here).

**Intuition 2.1: Truncation error (a recurring theme)**

The Taylor remainder is a **truncation error**: it exists even with exact arithmetic. Much of numerical analysis is about controlling truncation error by choosing good approximations and good discretisations.

# 3 Vandermonde interpolation

The classical problem of polynomial interpolation is to find a polynomial

$$p_n(x) = a_0 + a_1 x + \cdots + a_n x^n = \sum_{k=0}^{n} a_k x^k. \tag{3.1}$$

that interpolates a given function $f$ at a finite set of **nodes** (also called **interpolation points**)

$$\{x_0, x_1, \ldots, x_m\}.$$

Interpolation means

$$p_n(x_i) = f(x_i) \qquad \text{for each node } x_i.$$

Since $p_n$ has $n+1$ unknown coefficients, we typically need $n+1$ conditions, so we assume

$$m = n,$$

and the nodes $x_0, \ldots, x_n$ are distinct.

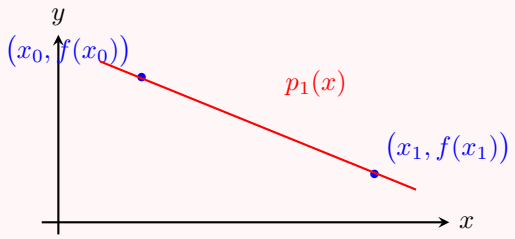
**Example 3.1: Linear interpolation ($n = 1$)**

With two nodes $x_0, x_1$ we look for $p_1(x) = a_0 + a_1 x$ and enforce

$$p_1(x_0) = a_0 + a_1 x_0 = f(x_0), \qquad p_1(x_1) = a_0 + a_1 x_1 = f(x_1).$$

Solving gives

$$p_1(x) = \frac{x_1 f(x_0) - x_0 f(x_1)}{x_1 - x_0} + \frac{f(x_1) - f(x_0)}{x_1 - x_0}\, x.$$




**Intuition 3.1: What linear interpolation is doing geometrically**

The line $p_1(x)$ is the unique straight line through the two points $(x_0, f(x_0))$ and $(x_1, f(x_1))$. It is the simplest polynomial model: easy to compute, usually robust, and often a good local approximation.

For general $n$, the interpolation conditions are

$$\begin{aligned} a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_n x_0^n &= f(x_0), \\ a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_n x_1^n &= f(x_1), \\ &\vdots \\ a_0 + a_1 x_n + a_2 x_n^2 + \cdots + a_n x_n^n &= f(x_n). \end{aligned} \tag{3.2}$$

This can be written as a linear system

$$\underbrace{\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}}_{\text{Vandermonde matrix } A} \underbrace{\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix}}_{\mathbf{a}} = \underbrace{\begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}}_{\mathbf{f}}. \tag{3.3}$$

We can then solve the system, by multiplying $A^{-1}$ to the left:

$$\mathbf{a} = A^{-1}\mathbf{f}. \tag{3.4}$$

Thus we obtain the coefficients of the interpolating polynomials.

**Intuition 3.2: Why this matrix appears**

The matrix entries are just the basis functions $1, x, x^2, \ldots, x^n$ evaluated at the nodes. So the Vandermonde matrix is what you get whenever you represent a polynomial in the *natural basis* $\{1, x, x^2, \ldots, x^n\}$.

**A key fact: the Vandermonde determinant.** One can show that

$$\det(A) = \prod_{0 \le i < j \le n} (x_j - x_i). \tag{3.5}$$

This is non-zero if and only if all nodes are distinct.

**Anecdote 3.1: Time complexity (why Vandermonde is expensive in practice).**

In numerical linear algebra, the best practice is:

- **Do not explicitly compute $A^{-1}$.** It is slower and typically less accurate.
- Instead, solve linear systems $A\mathbf{a} = \mathbf{f}$ using a factorization (e.g., LU with pivoting, or QR).

For a general dense $(n+1) \times (n+1)$ matrix, these direct methods cost $O(n^3)$. There are asymptotically faster algorithms for matrix multiplication/inversion (with complexity $O(n^\omega)$ for $\omega < 3$), but they are rarely used in everyday numerical computing because constant factors and numerical stability matter.
If we compute $p_n$ by explicitly forming the Vandermonde matrix $A$ and solving $A\mathbf{a} = \mathbf{f}$ by Gaussian elimination, the arithmetic cost is $O(n^3)$ (more precisely $O((n+1)^3)$) operations. This is already expensive for large $n$.
Even worse, the Vandermonde matrix is often ill-conditioned, so solving it can be *numerically unstable* in floating-point arithmetic. This is why practical interpolation uses alternative representations (Lagrange form, Newton form with divided differences, barycentric formula), which can reduce preprocessing to $O(n^2)$ and evaluation to $O(n)$, while being much more stable.

**Theorem 3.1: Existence and uniqueness of the interpolating polynomial**

Given $n+1$ distinct nodes $x_0, x_1, \ldots, x_n$, there is a unique polynomial $p_n \in P_n$ (the space of real polynomials of degree $\le n$) such that

$$p_n(x_i) = f(x_i), \qquad i = 0, 1, \ldots, n.$$

*Proof of Uniqueness.* Suppose $p_n$ and $q_n$ are both interpolating polynomials in $P_n$. Consider their difference $r_n = p_n - q_n$. Then $r_n \in P_n$ and for each node $x_i$,

$$r_n(x_i) = p_n(x_i) - q_n(x_i) = f(x_i) - f(x_i) = 0.$$

So $r_n$ has $n+1$ distinct roots. But a non-zero polynomial of degree $\le n$ can have at most $n$ roots. Therefore $r_n \equiv 0$, and hence $p_n = q_n$. □

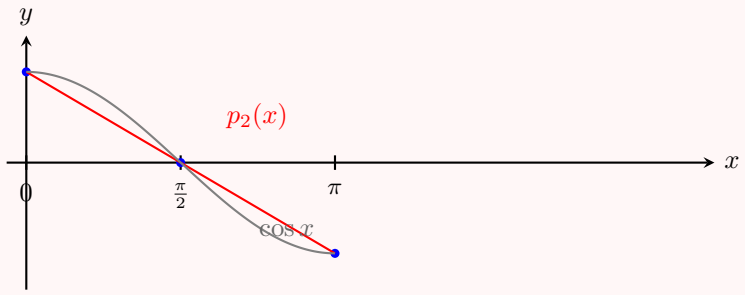
**Intuition 3.3: Where existence comes from**

Existence can be proved either by showing the Vandermonde matrix is invertible when nodes are distinct (since $\det(A) \neq 0$), or by explicitly constructing $p_n$ (Lagrange and Newton forms). The explicit constructions are more useful numerically and conceptually.

**Degree can be lower than $n$.** Even though we *allow* degree up to $n$, the unique interpolant through $n+1$ points can have degree $< n$ (for example if the highest coefficient happens to be $a_n = 0$).

**Example 3.2: Interpolating $\cos x$ at $\{0, \frac{\pi}{2}, \pi\}$**

Let $f(x) = \cos x$ and nodes $x_0 = 0$, $x_1 = \frac{\pi}{2}$, $x_2 = \pi$. Then $f(x_0) = 1$, $f(x_1) = 0$, $f(x_2) = -1$.




The unique interpolant in $p_2$ is actually a straight line:

$$p_2(x) = 1 - \frac{2}{\pi}x.$$

If instead we take nodes $\{0, 2\pi, 4\pi\}$ then $f$ equals 1 at all nodes, and the interpolant is the constant polynomial $p_2(x) = 1$.

**Anecdote 3.2: Why we avoid solving the Vandermonde system directly**

In theory you could compute $p_n$ by solving the linear system (e.g., Gaussian elimination). In practice, this is often numerically unstable for moderate/large $n$ because the Vandermonde matrix can be extremely ill-conditioned. That is why we usually switch to a better basis or a better formula (Lagrange/Newton), and we are careful about node choice.

# 4 Lagrange interpolation (Lagrange form)

The Vandermonde approach represents $p_n$ in the coefficient basis $\{1, x, x^2, \ldots, x^n\}$. Lagrange interpolation instead uses a special basis in which the interpolation conditions become the identity matrix.

**The key idea: basis functions that "pick out" nodes.** We seek basis polynomials $\ell_0, \ell_1, \ldots, \ell_n$ such that

$$\ell_k(x_i) = \begin{cases} 1, & \text{if } i = k, \\ 0, & \text{if } i \neq k. \end{cases} \tag{4.1}$$

If we have such a basis, then the interpolating polynomial is immediate:

$$p_n(x) = \sum_{k=0}^{n} f(x_k)\, \ell_k(x). \tag{4.2}$$

Indeed, evaluating at a node $x_i$ gives

$$p_n(x_i) = \sum_{k=0}^{n} f(x_k)\, \ell_k(x_i) = f(x_i).$$

**Intuition 4.1: Why this is like an "identity matrix"**

Think of the coefficients as the values $f(x_k)$. The condition $\ell_k(x_i) = \delta_{ik}$ means that, when you evaluate at $x_i$, only one term survives. So the interpolation equations reduce from "solve a linear system" to "read off the right value".

**Definition of the Lagrange basis polynomials.** For distinct nodes $x_0, \dots, x_n$, the Lagrange polynomials are defined by

$$\ell_k(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j}. \tag{4.3}$$

By construction, each factor is 0 at $x = x_j$ for $j \neq k$, so $\ell_k(x_j) = 0$ when $j \neq k$. At $x = x_k$ every factor equals 1, so $\ell_k(x_k) = 1$.

**Example 4.1: Linear interpolation revisited**

With nodes $x_0, x_1$, the Lagrange basis is

$$\ell_0(x) = \frac{x - x_1}{x_0 - x_1}, \qquad \ell_1(x) = \frac{x - x_0}{x_1 - x_0}.$$

So the interpolant can be written as

$$p_1(x) = f(x_0)\,\ell_0(x) + f(x_1)\,\ell_1(x),$$

which is the same line as before, but now the role of the function values is explicit.

**Anecdote 4.1: A small historical note**

These polynomials are called "Lagrange polynomials" today, but the formula was discovered earlier: Edward Waring (1776) and Euler (1783) both had versions before Lagrange published it in 1795.

**Example 4.2: Quadratic Lagrange interpolant for $\cos x$ on $\{-\frac{\pi}{4}, 0, \frac{\pi}{4}\}$**

Let $x_0 = -\frac{\pi}{4}$, $x_1 = 0$, $x_2 = \frac{\pi}{4}$ and $f(x) = \cos x$. Then $f(x_0) = f(x_2) = \frac{1}{\sqrt{2}}$ and $f(x_1) = 1$.
The degree-2 Lagrange polynomials are

$$\ell_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} = \frac{x\left(x - \frac{\pi}{4}\right)}{\left(-\frac{\pi}{4}\right)\left(-\frac{\pi}{2}\right)},$$

$$\ell_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} = \frac{\left(x + \frac{\pi}{4}\right)\left(x - \frac{\pi}{4}\right)}{\left(\frac{\pi}{4}\right)\left(-\frac{\pi}{4}\right)},$$

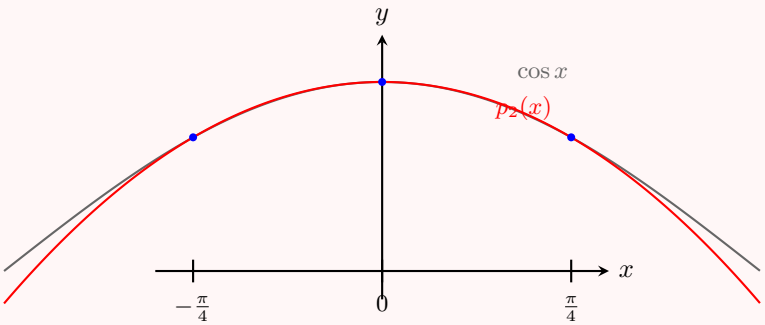
$$\ell_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} = \frac{x\left(x + \frac{\pi}{4}\right)}{\left(\frac{\pi}{2}\right)\left(\frac{\pi}{4}\right)}.$$

So

$$p_2(x) = f(x_0)\ell_0(x) + f(x_1)\ell_1(x) + f(x_2)\ell_2(x).$$

If you simplify the expression, you obtain a particularly clean form:

$$p_2(x) = 1 + \frac{16}{\pi^2}\left(\frac{1}{\sqrt{2}} - 1\right)x^2.$$



**Anecdote 4.2: What happens when $n$ grows?**

When $n$ is small (say $n = 1, 2, 3$), almost any method works and the formulas feel "cheap". But in scientific computing we often need dozens or hundreds of nodes, and then both cost and stability matter.

**Vandermonde: one big solve.** If you build $A$ and solve $A\mathbf{a} = \mathbf{f}$ with a dense factorization, you pay $O(n^3)$. If you then want to evaluate $p_n(x)$ at many points, Horner evaluation is $O(n)$ per point.

**Lagrange: no solve, but naive evaluation is expensive.** The Lagrange formula avoids solving a linear system, but if you evaluate it naively you end up computing many products. Roughly speaking:

- building all $\ell_k(x)$ symbolically is expensive,
- evaluating $p_n(x) = \sum f(x_k)\ell_k(x)$ from scratch can take $O(n^2)$ operations per evaluation point.

**Intuition 4.2: The right takeaway**

Lagrange form is the cleanest *mathematical* representation and is perfect for proofs. For computation, we usually rewrite it into a numerically stable and fast evaluation scheme (barycentric interpolation), or we use Newton's form (divided differences), especially if we want to add nodes incrementally.

**Anecdote 4.3: Updating nodes: Lagrange vs Vandermonde**

Suppose you add a new data point $(x_{n+1}, f(x_{n+1}))$.

- In the Vandermonde approach, you must solve a new, larger linear system (essentially start over).
- In the naive Lagrange approach, you must rebuild all basis polynomials as well (again, essentially start over).

Newton's form will later give us a much more natural "append one node" update.

# 5 Newton's divided-difference form

The Newton form is designed so that adding a new node contributes exactly one new term. Suppose we already have the degree-$n$ interpolant $p_n$. To increase the degree we write

$$p_{n+1}(x) = p_n(x) + g_{n+1}(x), \tag{5.1}$$

where $g_{n+1}$ is a $(n+1)$-degree polynomial. From the interpolation conditions,

$$g_{n+1}(x_i) = p_{n+1}(x_i) - p_n(x_i) = f(x_i) - f(x_i) = 0, \qquad i = 0, \ldots, n, \tag{5.2}$$

so $g_{n+1}$ must have the $n+1$ distinct roots $x_0, \ldots, x_n$. Hence

$$g_{n+1}(x) = a_{n+1}(x - x_0)\cdots(x - x_n). \tag{5.3}$$

The remaining condition at $x_{n+1}$ determines the single unknown coefficient:

$$a_{n+1} = \frac{f(x_{n+1}) - p_n(x_{n+1})}{(x_{n+1} - x_0)\cdots(x_{n+1} - x_n)}. \tag{5.4}$$

The polynomial $(x - x_0)\cdots(x - x_n)$ is called a **Newton polynomial**. These form a basis

$$n_0(x) = 1, \qquad n_k(x) = \prod_{j=0}^{k-1}(x - x_j), \quad k > 0. \tag{5.5}$$

Therefore the Newton form of the interpolant is

$$p_n(x) = \sum_{k=0}^{n} a_k n_k(x), \qquad a_0 = f(x_0), \qquad a_k = \frac{f(x_k) - p_{k-1}(x_k)}{(x_k - x_0)\cdots(x_k - x_{k-1})}, \quad k > 0. \tag{5.6}$$

Notice that $a_k$ depends only on $x_0, \ldots, x_k$, so we can build the coefficients sequentially. This is the key practical advantage: if a new node arrives, we keep the old coefficients and compute just one new $a_{n+1}$.

## 5.1 Divided differences

Define the **divided difference** $f[x_0, x_1, \dots, x_k]$ to be the coefficient of $x^k$ in the degree-$k$ interpolating polynomial through the nodes $x_0, \dots, x_k$. Then

$$f[x_0, x_1, \dots, x_k] = a_k. \tag{5.7}$$

For the first two orders,

$$f[x_0] = f(x_0), \tag{5.8}$$

$$f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}. \tag{5.9}$$

Continuing one step,

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}. \tag{5.10}$$

In general we have the recursion.

**Theorem 5.1: Divided-difference recursion**

For $k > 0$,

$$f[x_i, x_{i+1}, \dots, x_{i+k}] = \frac{f[x_{i+1}, \dots, x_{i+k}] - f[x_i, \dots, x_{i+k-1}]}{x_{i+k} - x_i}. \tag{5.11}$$

*Idea of proof.* Write the degree-$k$ interpolant through $x_0, \dots, x_k$ as

$$p_k(x) = \frac{(x_k - x)q_{k-1}(x) + (x - x_0)\tilde{q}_{k-1}(x)}{x_k - x_0},$$

where $q_{k-1}$ interpolates $f$ at $x_0, \dots, x_{k-1}$ and $\tilde{q}_{k-1}$ interpolates $f$ at $x_1, \dots, x_k$. Matching the coefficient of $x^k$ yields the recursion. □

**Example 5.1**

Nodes $\{-1, 0, 1, 2\}$ with data $\{5, 1, 1, 11\}$ give the divided differences below. The coefficients appear at the left edge of each level, so it is natural to visualize the computation as a tree:

$$\begin{array}{llllll}
x_0 = -1 & f[x_0] = 5 & & & & \\
 & & f[x_0, x_1] = -4 & & & \\
x_1 = 0 & f[x_1] = 1 & & f[x_0, x_1, x_2] = 2 & & \\
 & & f[x_1, x_2] = 0 & & f[x_0, x_1, x_2, x_3] = 1 & \\
x_2 = 1 & f[x_2] = 1 & & f[x_1, x_2, x_3] = 5 & & \\
 & & f[x_2, x_3] = 10 & & & \\
x_3 = 2 & f[x_3] = 11 & & & &
\end{array}$$

The coefficients of $p_3$ are the first entries in each column, so

$$p_3(x) = 5 - 4(x + 1) + 2x(x + 1) + x(x + 1)(x - 1).$$

If we add a new node $x_4 = -2$ with $f(x_4) = 5$, only the entries that involve this node change. Those new

entries are shown in BrickRed in the updated tree below:

$$
\begin{array}{lllll}
f[x_0]=5 & & & & \\
 & f[x_0,x_1]=-4 & & & \\
f[x_1]=1 & & f[x_0,x_1,x_2]=2 & & \\
 & f[x_1,x_2]=0 & & f[x_0,x_1,x_2,x_3]=1 & \\
f[x_2]=1 & & f[x_1,x_2,x_3]=5 & & f[x_0,x_1,x_2,x_3,x_4]=-\frac{1}{12} \\
 & f[x_2,x_3]=10 & & f[x_1,x_2,x_3,x_4]=\frac{4}{3} & \\
f[x_3]=11 & & f[x_2,x_3,x_4]=-2 & & \\
 & f[x_3,x_4]=0 & & & \\
f[x_4]=5 & & & &
\end{array}
$$

Therefore,

$$p_4(x) = p_3(x) - \frac{1}{12}(x-x_0)(x-x_1)(x-x_2)(x-x_3). \tag{5.12}$$

**Intuition 5.1: Time complexity and updates**

Building the full divided-difference table for $n+1$ nodes requires about $1+2+\cdots+n = O(n^2)$ arithmetic operations. Once the table (or just its first column of coefficients) is available, evaluating the Newton form at a single point costs $O(n)$ using nested multiplication. If a new node is appended, only one new entry per level is needed, so the update cost is $O(n)$ rather than recomputing the entire table.

If we append new nodes, only new rows are added to the table; existing entries are unchanged. This is the key computational advantage of Newton's form.

## 5.2 2.5 Interpolation error

The goal is to estimate the error $|f(x)-p_n(x)|$ when we approximate $f$ by its interpolating polynomial $p_n$. The error depends on $x$ and on the node locations.

**Example 5.2: Quadratic interpolant for $f(x)=\cos x$ on $\{-\frac{\pi}{4},0,\frac{\pi}{4}\}$**

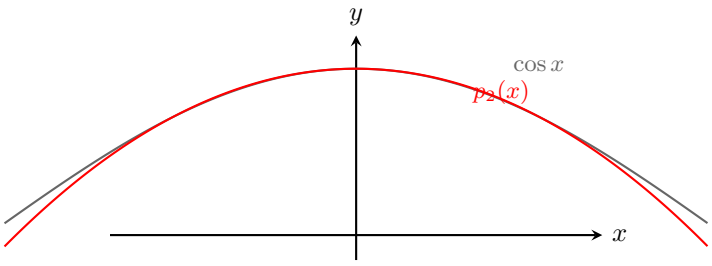
From Section 2.3, the interpolant is

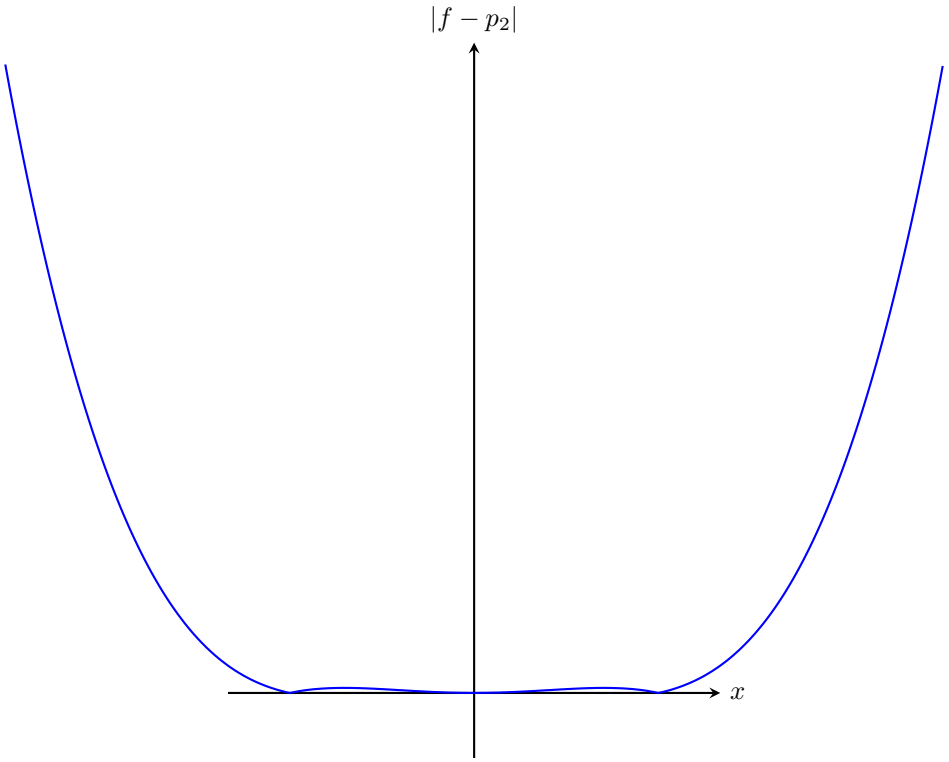
$$p_2(x) = 1 + \frac{16}{\pi^2}\left(\frac{1}{\sqrt{2}}-1\right)x^2.$$

So the pointwise error is

$$|f(x)-p_2(x)| = \left|\cos(x) - 1 - \frac{16}{\pi^2}\left(\frac{1}{\sqrt{2}}-1\right)x^2\right|.$$

The error vanishes at the nodes themselves, and it is typically smallest near the middle of the node set.






**Theorem 5.2: Cauchy interpolation error**

Let $p_n \in P_n$ be the unique polynomial interpolating $f$ at distinct nodes $x_0, x_1, \ldots, x_n \in [a, b]$, and assume $f$ is continuous on $[a, b]$ with $n + 1$ continuous derivatives on $(a, b)$. Then for each $x \in [a, b]$ there exists $\xi \in (a, b)$ such that

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

**Intuition 5.2: What the formula tells us**

The error vanishes at every node, since the product $(x - x_0)\cdots(x - x_n)$ is zero there. It also shows two main drivers of error: (i) the size of the $(n+1)$st derivative (how "wiggly" $f$ is), and (ii) the size of the node polynomial $\prod(x - x_i)$. Increasing $n$ does *not* automatically reduce the error; the node placement matters.

**Intuition 5.3**

When does the interpolation error $\frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n)$ become 0?

1. when $x = x_i$, i.e. when $x$ is one of the nodes. In other words, the error vanishes at the nodes.
2. when $f^{(n+1)} = 0$, i.e. when $f$ is a $n$-degree polynomial.
3. when $n \to \infty$, $(n+1)!$ blows up very rapidly, so $\frac{1}{(n+1)!}$ goes to 0. However, $(x-x_0)(x-x_1)\cdots(x-x_n)$ gets very wiggly as $n \to \infty$.

*Proof sketch.* Define the auxiliary function

$$g(t) = f(t) - p_n(t) - M\prod_{i=0}^{n}(t - x_i),$$

and choose $M$ so that $g(x) = 0$. Then $g$ has $n + 2$ distinct roots: $x_0, \ldots, x_n, x$. Repeatedly applying Rolle's theorem yields a point $\xi$ with $g^{(n+1)}(\xi) = 0$. Differentiating gives $g^{(n+1)}(t) = f^{(n+1)}(t) - M(n+1)!$, which leads to the formula above. □

**Example 5.3: Bounding the error for the cosine example**

For $n = 2$ the theorem gives

$$f(x) - p_2(x) = \frac{f^{(3)}(\xi)}{6}\, x\left(x + \frac{\pi}{4}\right)\left(x - \frac{\pi}{4}\right) = \frac{\sin(\xi)}{6}\, x\left(x + \frac{\pi}{4}\right)\left(x - \frac{\pi}{4}\right),$$

for some $\xi \in [-\frac{\pi}{4}, \frac{\pi}{4}]$. Using $|\sin(\xi)| \leq 1$ gives a simple bound once $x$ is fixed. On $[-1, 1]$, the polynomial $w(x) = x(x + \frac{\pi}{4})(x - \frac{\pi}{4})$ has extrema at $x = \pm\frac{\pi}{4\sqrt{3}}$, and $\max_{[-1,1]} |w(x)| \approx 0.383$, so

$$|f(x) - p_2(x)| \leq \frac{1}{6}(0.383) \approx 0.0638.$$

The plot above shows the true error is smaller than this bound (as it must be).