MATH 227B: Mathematical Biology
Homework 2

Karthik Desingu

January 24, 2026

Code availability

All the code used for these problems is available publicly at this GitHub repository.

Problem 1

(a) Write a function to compute divided difference. Demonstrate correctness of your code.

We implemented a function `divided_difference` to compute the Newton divided differences for nodes $x_0, \ldots, x_n$ and values $f(x_0), \ldots, f(x_n)$. The divided differences are defined recursively by

$$f[x_i] = f(x_i), \qquad f[x_i, \ldots, x_{i+j}] = \frac{f[x_{i+1}, \ldots, x_{i+j}] - f[x_i, \ldots, x_{i+j-1}]}{x_{i+j} - x_i}.$$

These values form a triangular table whose first row provides the Newton coefficients, which can be used directly in the Newton-form interpolating polynomial.

Demonstration of correctness

Correctness was verified using Python unit tests (`unittest`) on functions of increasing degree:

- **Constant:** $f(x) = 5$, all higher-order differences vanish.
- **Linear:** $f(x) = 2x + 1$, the first-order divided difference equals the slope.
- **Quadratic:** $f(x) = x^2$, the second-order difference is constant.
- **Cubic:** $f(x) = x^3$, the third-order difference is constant.
- **Random quadratic:** $f(x) = 3 - 2x + x^2$, testing mixed coefficients.

- **Nonmonotone nodes:** $x = \{1, -1, 2\}$ with $f(x) = x^2$, testing general node ordering.

These cases confirm exactness for polynomials, vanishing of higher-order differences for lower-degree polynomials, and robustness with respect to arbitrary node ordering.

Visual verification

A dense divided-difference table was computed for the fifth-degree polynomial

$$f(x) = x^5 - 3x^4 + 2x^3 - x + 1$$

using seven equally spaced nodes. Figure 1 shows a heatmap of the table alongside a plot comparing the Newton-form evaluation against the exact polynomial.

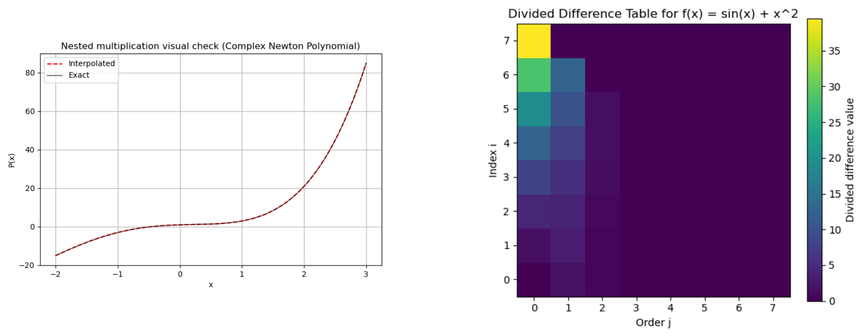


Figure 1: Verification of polynomial interpolation routines. Left: divided-difference table showing the expected triangular structure and smooth column-wise variation, confirming correct recursive construction. Right: Newton-form polynomial evaluation showing perfect overlap between the nested-multiplication result and the exact polynomial, demonstrating accurate polynomial reproduction.

Tabular verification

Representative results from the unit tests are shown in Table 1.

Python implementation

Listing 1: Python implementation of `divided_difference`.

```python
def divided_difference(x, fvals):
    """
    Compute the first row of the Newton divided-difference table.

    Parameters
    ----------
```

Test function	Nodes	Computed	Expected
$f(x) = 5$	$[-1, 0, 2]$	$(5, 0, 0)$	$(5, 0, 0)$
$f(x) = 2x + 1$	$[-1, 0, 1]$	$(-1, 2, 0)$	$(-1, 2, 0)$
$f(x) = x^2$	$[0, 1, 2]$	$(0, 1, 1)$	$(0, 1, 1)$
$f(x) = x^3$	$[0, 1, 2, 3]$	$(0, 1, 3, 1)$	$(0, 1, 3, 1)$
$f(x) = 3 - 2x + x^2$	$[0, 1, 2]$	$(3, -1, 1)$	$(3, -1, 1)$

Table 1: Verification of divided differences for selected unit tests. The computed Newton coefficients agree with analytical results to machine precision.

```
    x : list of floats
        Interpolation nodes [x0, x1, ..., xn]
    fvals : list of floats
        Function values [f(x0), f(x1), ..., f(xn)]

    Returns
    -------
    list of floats
        Newton coefficients [f[x0], f[x0,x1], ..., f[x0,...,xn]]
    """
    n = len(x)
    table = [[0.0] * n for _ in range(n)]

    # first column: f[x_i] = f(x_i)
    for i in range(n):
        table[i][0] = fvals[i]

    # build table of divided differences
    for j in range(1, n):
        for i in range(n - j):
            table[i][j] = (table[i+1][j-1] - table[i][j-1]) / (x[i+
                j] - x[i])

    # return first row, which contains the Newton coefficients
    return table[0]
```

Thus, correctness is established by agreement with analytical divided differences for polynomials of degree up to three, robustness to arbitrary coefficients and node ordering, and visual confirmation via the dense divided-difference heatmap.

(b) Write a function to compute the polynomial using nested multiplication. Demonstrate correctness of your code.

We implemented a function `compute_polynomial` to evaluate a Newton-form polynomial at a given point using *nested multiplication* (Horner's method). Given Newton coefficients $a_0, \dots, a_n$ and nodes $x_0, \dots, x_{n-1}$, the polynomial

$$P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n \prod_{i=0}^{n-1} (x - x_i)$$

is evaluated efficiently using the backward recurrence

$$\text{value} = a_n, \qquad \text{for } k = n-1, \ldots, 0: \ \text{value} \leftarrow \text{value}\,(x - x_k) + a_k,$$

which avoids repeated construction of products and improves numerical stability.

To demonstrate correctness, we designed several test polynomials of increasing complexity:

- **Constant polynomial:** $P(x) = 5$.
- **Linear polynomial:** $P(x) = 2 + 3(x - 1)$.
- **Quadratic polynomial:** $P(x) = 1 + 2(x - 0) + 3(x - 0)(x - 1)$.
- **Cubic polynomial:** $P(x) = 1 + 2(x - 0) + 3(x - 0)(x - 1) + 4(x - 0)(x - 1)(x - 2)$.
- **Random Newton-form polynomial:** $P(x) = 1 - 2(x - 0) + 0.5(x - 0)(x - 1) + 3(x - 0)(x - 1)(x + 1)$.

Each polynomial was evaluated at multiple points across and beyond the node interval, and unit tests in Python (`unittest`) verified agreement with the exact formulas up to machine precision.

Visual verification

Figure 1 (right panel) provides a visual check for a Newton-form polynomial with five coefficients and four nodes: the curve generated by `compute_polynomial` is indistinguishable from the analytically derived expression over a dense grid, confirming the correctness of the implementation.

Tabular verification

Table 2 lists sample evaluation points for the cubic Newton-form polynomial

$$P(x) = 1 + 2(x - 0) + 3(x - 0)(x - 1) + 4(x - 0)(x - 1)(x - 2)$$

and shows that nested multiplication reproduces the exact values with absolute error below 10^{-15}.

Python implementation

Listing 2: Python implementation of `compute_polynomial` using nested multiplication.

```python
def compute_polynomial(a, x_nodes, x):
    """
    Evaluate a Newton-form interpolating polynomial using nested
        multiplication.

    Parameters
```

x	$P(x)$ (nested multiplication)	$P(x)$ (exact formula)	Absolute error
-1.0	-7.0	-7.0	$< 10^{-15}$
0.0	1.0	1.0	$< 10^{-15}$
0.5	3.875	3.875	$< 10^{-15}$
1.5	16.0	16.0	$< 10^{-15}$
3.0	91.0	91.0	$< 10^{-15}$

Table 2: Verification of `compute_polynomial` for a cubic Newton-form polynomial at selected points. The nested-multiplication implementation reproduces the exact values to within floating-point precision.

```
    ----------
    a : list of floats
        Newton coefficients [a0, a1, ..., an]
    x_nodes : list of floats
        Nodes used in Newton form [x0, x1, ..., x_{n-1}]
    x : float
        Point where the polynomial is evaluated

    Returns
    -------
    float
        P(x)
    """
    n = len(a)
    value = a[-1]
    for k in range(n - 2, -1, -1):
        value = value * (x - x_nodes[k]) + a[k]
    return value
```

This function, together with the unit tests and visual/table-based verification, demonstrates correctness for constant, linear, quadratic, cubic, and general Newton-form polynomials.

(c) Combine (a) and (b) to write a package for Lagrange polynomial interpolation. Discuss: (1) Use a flowchart to show how you make the combined code in modules to minimize changes when different functions are used for the package; (2) How you test the correctness of your overall implementation.

We combine the divided-difference routine in (a) and the nested-multiplication evaluator in (b) into a modular package for Lagrange (Newton-form) polynomial interpolation. The design goal is to ensure that only the user-supplied function $f(x)$ or data values change when a new interpolation problem is considered, while the numerical core remains untouched.

Modular structure and flowchart

The computation is organized into three modules:

1. `divided_difference`: computes the Newton coefficients from (x_i, f_i).
2. `compute_polynomial`: evaluates a Newton-form polynomial using nested multiplication.
3. `interpolate`: high-level interface that accepts either a callable function $f(x)$ or tabulated values and invokes the two lower-level modules.

Only the user-supplied function or data changes when interpolating a different function; the numerical routines are reused without modification, which improves maintainability and reusability.

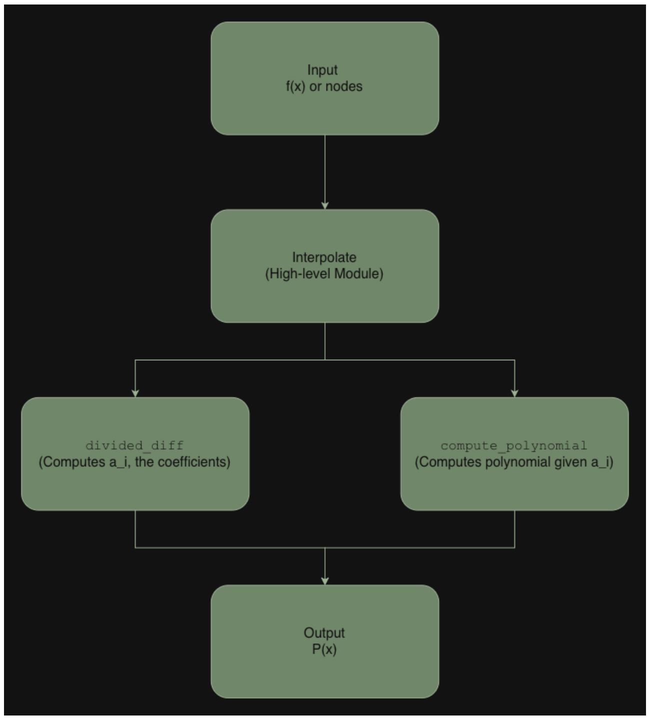


Figure 2: Flowchart of the interpolation package. The user provides nodes x_i or a given function $f(x)$. The program first computes divided differences, then evaluates the Newton polynomial via nested multiplication. The modular design ensures that changing $f(x)$ or the data values requires no modification to the underlying numerical routines.

Tabular verification

Correctness of the combined implementation was verified using structured Python unit tests. The tests cover exact polynomial reproduction, interpolation at data points, and symmetry properties; Table 3 summarizes the cases.

Test	Function / Data	Property verified
Exact quadratic	$3x^2 + 2x + 1$	Exact recovery
Exact cubic	x^3	Exact recovery
Interpolates nodes	Tabulated (x_i, f_i)	$P(x_i) = f_i$
Random polynomial	Quadratic with random coefficients	Reconstruction
Even symmetry	x^2	$P(x) = P(-x)$

Table 3: Summary of unit tests for the interpolation package.

These tests confirm both algebraic correctness (polynomials of degree $\leq n - 1$ are reproduced exactly in exact arithmetic) and structural correctness (the interpolant matches all data points).

Visual verification

In addition to unit tests, a non-polynomial function

$$f(x) = \sin(3x) + 0.3x^2$$

was interpolated on $[-1, 1]$ using equally spaced nodes. Figure 3 shows the interpolant and the corresponding pointwise error.

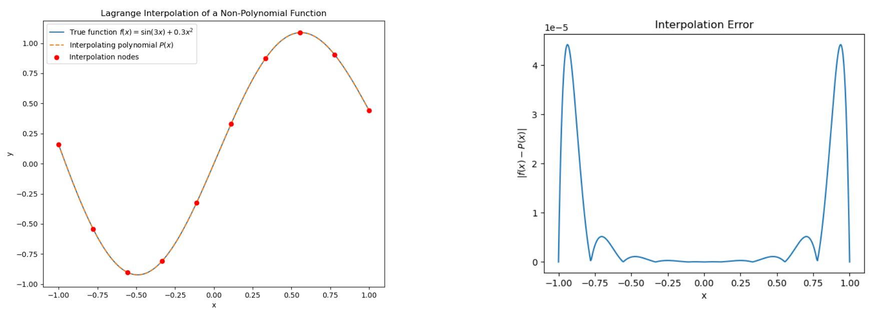


Figure 3: Left: true function $f(x) = \sin(3x) + 0.3x^2$ and its Lagrange interpolant. Right: absolute interpolation error $|f(x) - P(x)|$. The error is small in the interior and larger near the endpoints, indicating the onset of Runge-type oscillations for equally spaced nodes.

The error remains small in the interior and grows near the endpoints, reflecting oscillations associated with high-degree polynomial interpolation on equally spaced nodes (a manifestation of the Runge phenomenon).

Python implementation

Listing 3: Interpolation routine using divided differences and nested multiplication.

```python
def interpolate(x_nodes, f, x):
    """
    Lagrange interpolation at x using the Newton form.

    Parameters
    ----------
    x_nodes : list of floats
        Interpolation nodes.
    f : callable or list of floats
        Function f(x) or precomputed values f(x_i).
    x : float
        Evaluation point.

    Returns
    -------
    float
        Interpolated value P(x).
    """
    # Allow either f(x) or precomputed values
    if callable(f):
        fvals = [f(xi) for xi in x_nodes]
    else:
        fvals = f

    a = divided_difference(x_nodes, fvals)
    return compute_polynomial(a, x_nodes, x)
```

Together, the modular design, unit tests, and visual error analysis demonstrate the correctness and robustness of the combined Lagrange interpolation package.

(d) Use your package to obtain the Lagrange polynomial interpolation for the three given functions with n equally spaced points. Discuss how the errors between the interpolation function and $f(x)$ depend on n.

We now use the interpolation package to approximate three representative functions and study how the interpolation error depends on the number of equally spaced nodes n.

Test functions and setup

We consider the following functions and intervals:

1. Cubic polynomial: $f_1(x) = 3x^3 + 4x^2 + 2x + 1$ on $[-1, 1]$.
2. Trigonometric function: $f_2(x) = \sin(x)$ on $[0, 2\pi]$.
3. Runge function: $f_3(x) = \dfrac{1}{1 + 25x^2}$ on $[-1, 1]$.

For each f_j, we interpolate using n equally spaced nodes with n ranging from 2 to 13. At each n, the interpolant $P_n(x)$ is evaluated at 500 equally spaced test points in the interval, and the absolute error

$$E(x) = |f(x) - P_n(x)|$$

is computed.

Analytical error bounds

The theoretical error bound for Lagrange interpolation of a function f using n nodes $x_0, \ldots, x_{n-1}$ is

$$|f(x) - P_n(x)| \leq \frac{\max_{\xi \in [a,b]} |f^{(n)}(\xi)|}{n!} \prod_{i=0}^{n-1} |x - x_i|. \tag{1}$$

Here $\max_{\xi \in [a,b]} |f^{(n)}(\xi)|$ depends on the function, while the product encodes the geometry of the nodes. For our three test functions we use:

1. **Cubic polynomial $f_1(x)$:** The fourth and higher derivatives vanish identically, so for $n \geq 4$ the bound in (1) is exactly zero:

$$f_1^{(n)}(x) = 0 \quad \Rightarrow \quad |f_1(x) - P_n(x)| \leq 0.$$

2. **Sine function $f_2(x)$:** The n-th derivative of $\sin(x)$ is either $\sin(x)$ or $\cos(x)$ up to a sign, hence

$$\max_{x \in [0, 2\pi]} |f_2^{(n)}(x)| = 1,$$

and the bound reduces to

$$|f_2(x) - P_n(x)| \leq \frac{1}{n!} \prod_{i=0}^{n-1} |x - x_i|.$$

3. **Runge function $f_3(x)$:** The derivatives of f_3 grow rapidly with n and are largest near $x = 0$, with

$$\max_{\xi \in [-1,1]} |f_3^{(n)}(\xi)| \sim n!\, 25^n.$$

Consequently, the bound grows with n, especially near the interval endpoints, and captures the Runge phenomenon.

Implementation in code. In the Python function `analytical_error_bound`, we implemented (1) using a log-space computation to avoid overflow when evaluating $\prod |x - x_i|$. The key design choices are:

1. *Maximum derivative term:* `max_f_deriv` is set to 0 for the polynomial, 1 for $\sin(x)$, and $n!\, 25^n$ for the Runge function.

2. *Factorial normalization:* $n!$ is computed using `math.factorial`.
3. *Node product term:* the product $\prod |x - x_i|$ is evaluated via
$$\prod_{i=0}^{n-1} |x - x_i| = \exp\left(\sum_{i=0}^{n-1} \log |x - x_i|\right),$$
replacing exact zeros by a small ε to avoid $\log(0)$.
4. *Special handling for exact polynomials:* for f_1 and $n > 3$, the error bound is set to 0 directly to avoid spurious NaNs.

Listing 4: Analytical Lagrange interpolation error bound computation.

```python
def analytical_error_bound(f_name, n, x_nodes, x_test):
    """
    Analytical Lagrange bound using log-space to avoid overflow.

    Parameters
    ----------
    f_name : {'poly', 'sin', 'runge'}
        Name of the test function.
    n : int
        Number of interpolation nodes.
    x_nodes : array-like
        Interpolation nodes.
    x_test : array-like
        Test points where the bound is evaluated.

    Returns
    -------
    np.ndarray
        Array of error-bound values at x_test.
    """
    if f_name == 'poly':
        max_f_deriv = 0.0  # polynomial of degree <= n -> exact
    elif f_name == 'sin':
        max_f_deriv = 1.0  # max |sin^(n)(x)| = 1
    elif f_name == 'runge':
        max_f_deriv = math.factorial(n) * 25**n  # bound near x = 0
    else:
        raise ValueError("Unknown function")

    fact = math.factorial(n)
    x_nodes_arr = np.array(x_nodes)
    error_bound = []
    eps = 1e-16

    for x in x_test:
        # special case for polynomials: when degree <= n, error is
            0
        if f_name == 'poly' and n > 3:
            error_bound.append(0.0)
            continue

        diff = x - x_nodes_arr
        diff = np.where(diff == 0, eps, diff)
        log_omega = np.sum(np.log(np.abs(diff)))
        log_bound = np.log(max_f_deriv + 1e-16) + log_omega - np.
            log(fact)
        error_bound.append(np.exp(log_bound))

    return np.array(error_bound)
```

A more general approach would be to estimate $\max|f^{(n)}|$ via automatic differentiation, but in practice `autograd` and `jax` encountered memory and runtime issues for the Runge function, so analytical estimates were used instead.

Error analysis

For each function we generated two main types of plots:

- Maximum and RMS error versus n, together with the analytical error bound.
- Interpolated polynomial versus the exact function, with interpolation nodes shown as red dots.

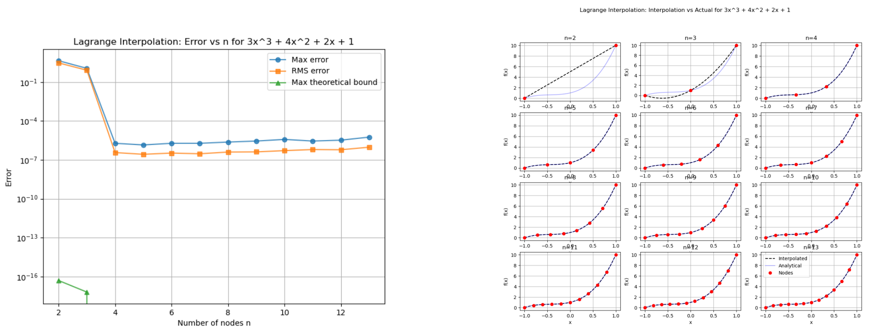


Figure 4: Lagrange interpolation analysis for $f_1(x) = 3x^3 + 4x^2 + 2x + 1$. Left: maximum and RMS errors as a function of n compared to the theoretical bound. Right: interpolating polynomial (dashed black), exact function (blue), and nodes (red dots).

Cubic polynomial $f_1(x)$. For $n > 3$, the analytical error is zero, and the numerical errors plateau around 10^{-6} due to floating-point round-off. The flat error curves for $n \geq 4$ confirm that increasing the number of nodes beyond the polynomial degree does not reduce the error further in finite precision.

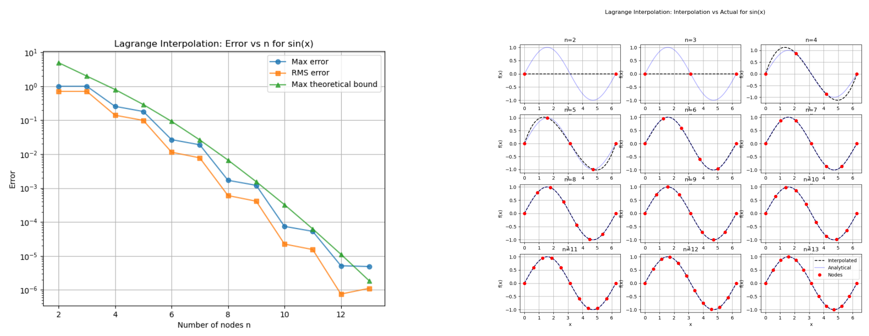


Figure 5: Lagrange interpolation analysis for $f_2(x) = \sin(x)$. Left: maximum and RMS errors versus n compared to the theoretical bound. Right: interpolant (dashed black), exact function (blue), and nodes (red dots).

Sine function $f_2(x)$**.** For $\sin(x)$, both maximum and RMS errors decrease rapidly as n increases, in line with the bound based on $\max |f_2^{(n)}| = 1$. The error curves show mild oscillations in n, but the overall trend is decreasing, reflecting accurate convergence for this smooth periodic function.

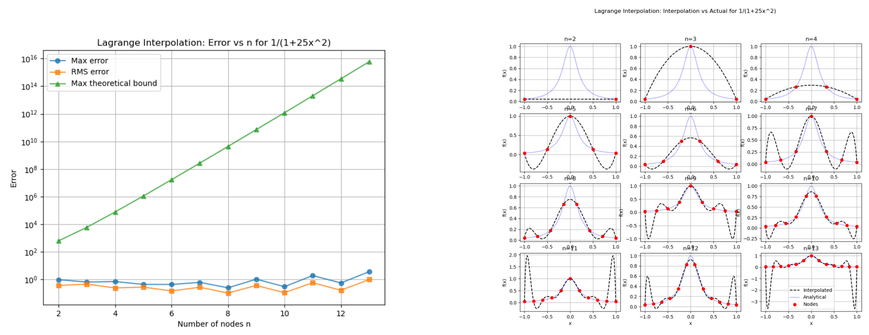


Figure 6: Lagrange interpolation analysis for $f_3(x) = \dfrac{1}{1 + 25x^2}$ (Runge function). Left: maximum and RMS errors as a function of n compared to the theoretical bound. Right: interpolant (dashed black), exact function (blue), and nodes (red dots).

Runge function $f_3(x)$**.** For the Runge function, both the theoretical bound and the observed maximum error increase near the interval edges as n grows, illustrating the Runge phenomenon: high-degree interpolation with equally spaced nodes produces large endpoint oscillations. The right panel shows noticeable overshoots near $x = \pm 1$ that become more pronounced for larger n, while the interpolant remains accurate near the center.

Observations and special cases

1. **Cubic polynomial (f_1):** The error bound is exactly zero for $n > 3$, and the numerical error saturates at a small, nonzero level ($\approx 10^{-6}$) because of floating-point limitations.
2. **Sine function (f_2):** Maximum and RMS errors both decrease with n, and their trend is consistent with the analytical bound $\frac{1}{n!}\prod |x - x_i|$, indicating good convergence for a smooth, bounded function.
3. **Runge function (f_3):** The errors grow at the interval endpoints as n increases, while remaining moderate in the interior. The qualitative behavior matches the growth of the analytical bound and clearly exhibits the classical Runge phenomenon for equally spaced nodes.

Summary

The Lagrange interpolation package successfully:

- reproduces polynomials of degree $\leq n$ exactly in exact arithmetic,
- shows decreasing errors with increasing n for smooth bounded functions such as $\sin(x)$, and
- illustrates the limitations of high-degree interpolation with equally spaced nodes for functions like the Runge function, where endpoint errors increase with n.

The theoretical bound

$$|f(x) - P_n(x)| \leq \frac{\max_{\xi \in [a,b]} |f^{(n)}(\xi)|}{n!} \prod_{i=0}^{n-1} |x - x_i|$$

captures these trends and the numerical analysis results above corroborate the theoretical limit.

Problem 2

Demonstrate the order of accuracy computationally for the cubic spline. Construct at least two examples using non-trivial functions to show order of accuracy. Please try two different boundary conditions implemented.

Here, we computationally demonstrate the order of accuracy of cubic spline interpolation. We use multiple non-trivial functions to illustrate the convergence of the spline approximation under two different boundary conditions: the *complete spline* (where the first derivatives at the endpoints are specified using the exact function) and the *not-a-knot spline* (where additional smoothness is enforced at interior nodes instead of prescribing endpoint derivatives).

Choice of Functions and Intervals

We selected three functions for analysis to illustrate the convergence behavior of cubic splines under different scenarios:

1. $f_1(x) = \sin(x)$ on $[0, 2\pi]$. This function is smooth, periodic, and oscillatory. The interval $[0, 2\pi]$ contains exactly one full period, which lets us see how well the spline captures complete oscillations. Sine is infinitely differentiable, so it satisfies the smoothness assumptions of the cubic spline error theorem. Using this interval also ensures that the nodes are spread over regions of both high and low slope, which helps show how the spline adapts to changing curvature.
2. $f_2(x) = x^2(1-x)^2$ on $[-1, 1]$. This function is a smooth polynomial with zeros at the endpoints and a peak in the interior. It is simple but non-trivial: it is neither linear nor purely oscillatory. The chosen interval $[-1, 1]$ forces the spline to capture the interior curvature accurately while respecting the zero boundary behavior. Since the function is fourth-degree, it allows us to see the exactness property of cubic splines for lower-degree polynomials and to test error trends when the function is smooth but not overly simple.
3. $f_3(x) = \exp(x)$ on $[-1, 1]$. The exponential function is smooth and strictly increasing with convex curvature. The interval $[-1, 1]$ gives a range where the function grows moderately and has enough variation in slope and curvature to show how the spline behaves on a non-polynomial function. This example highlights how cubic splines handle non-polynomial growth while still showing the expected convergence rates.

For all three functions, the chosen intervals include both regions of low and high curvature, so the spline error is meaningful across the whole interval. These functions also satisfy the smoothness assumptions required for the theoretical error bounds of cubic splines, which lets us observe the expected order of accuracy under different boundary conditions.

Implementation Details

We implemented cubic splines using two boundary conditions:

- **Complete (clamped) spline:** The first derivatives at the endpoints are specified using the exact derivatives of the function, i.e.,

  $$s'(a) = f'(a), \quad s'(b) = f'(b).$$

  This forces the spline to match the slope of the underlying function at the endpoints. In Python, this is implemented using the *clamped* boundary condition, with endpoint derivatives computed from the given function using *autograd*.

- **Not-a-knot spline:** The spline is constrained so that the third derivative is continuous at the second and penultimate nodes,

$$s'''(x_1) = s'''(x_0) \quad \text{and} \quad s'''(x_{n-1}) = s'''(x_n),$$

  effectively reducing the influence of the endpoints and providing a smooth cubic interpolation without explicitly specifying endpoint derivatives.

For each function, we evaluated spline interpolations using nodes ranging from $n = 5$ to $n = 53$ in steps of 2. Test points for evaluating interpolation error were uniformly spaced, with 500 points across each interval.

For each n, we computed two error measures:

- **Maximum error:**

$$E_{\max} = \max_{x \in \text{test points}} |f(x) - s(x)|,$$

  which captures the worst-case deviation between the spline and the true function.

- **Root-mean-square (RMS) error:**

$$E_{\text{RMS}} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} |f(x_i) - s(x_i)|^2},$$

  which gives a sense of the overall error distribution across the interval.

We also computed errors for the first and second derivatives using the derivatives of the spline $s'(x)$ and $s''(x)$, and the exact derivatives $f'(x)$ and $f''(x)$:

$$E^{(1)}_{\max} = \max_i |f'(x_i) - s'(x_i)|, \quad E^{(1)}_{\text{RMS}} = \sqrt{\frac{1}{N} \sum_i |f'(x_i) - s'(x_i)|^2},$$

$$E^{(2)}_{\max} = \max_i |f''(x_i) - s''(x_i)|, \quad E^{(2)}_{\text{RMS}} = \sqrt{\frac{1}{N} \sum_i |f''(x_i) - s''(x_i)|^2}.$$

Theoretical Error Bound

For a sufficiently smooth function $f \in C^4[a, b]$, the classical cubic spline error theorem states that the complete (clamped) spline $s(x)$ satisfies

$$|f(x) - s(x)| \le K h^4, \quad x \in [a, b],$$

where $h = \max_i (x_{i+1} - x_i)$ is the maximum spacing between nodes, and K is a constant depending on the fourth derivative of f but independent of h.

The theorem assumes three key conditions:

1. f has at least four continuous derivatives on $[a, b]$ ($f \in C^4[a, b]$),
2. The spline is a **complete spline** with the first derivatives at the endpoints correctly specified, and
3. The mesh of nodes is sufficiently regular (no extremely clustered or widely spaced nodes).

For other boundary conditions, such as not-a-knot, the same exact bound does not strictly apply, but the asymptotic trend $O(h^4)$ is typically observed in practice. Similarly, the first and second derivative errors satisfy

$$|f'(x) - s'(x)| = O(h^3), \quad |f''(x) - s''(x)| = O(h^2).$$

Analysis

For each function and boundary condition, we generate the following analyses:

1. **Log-log error vs. h plots:** We plot $E_{\max}$ and E_{RMS} for f, f', and f'' against h using logarithmic axes. The x-axis shows $\log(h)$ and the y-axis shows $\log(E)$. Dashed lines show the RMS error in the same color as the corresponding maximum error. Reference lines indicate the expected slopes from theory,

$$\text{slope 4 for } f, \quad 3 \text{ for } f', \quad 2 \text{ for } f'',$$

so we can visually check whether the observed convergence rates match the theory.

2. **Scaled error plots:** We plot the scaled errors

$$\frac{E_{\max}}{h^p}, \quad \frac{E_{\text{RMS}}}{h^p}, \quad p = 4, 3, 2$$

for f, f', and f'' respectively. If the spline actually achieves the expected order of convergence, the scaled error should flatten out and approach a horizontal line as $h \to 0$. These plots help identify pre-asymptotic regions and give a sense of the constant K.

3. **Reference line plot:** We plot $E_{\max}$ for f together with a dashed line representing Kh^4, where K is chosen from the finest-grid error. This lets us see how closely the observed errors track the asymptotic h^4 behavior and gives a quantitative feel for the convergence constant.

4. **Fit plots:** For each n, we plot the spline $s(x)$ and the exact function $f(x)$ on the same axes, with interpolation nodes shown as red points. Watching these fits as n increases gives a more intuitive picture of how the spline converges and where the remaining errors are located (for example, near endpoints or near regions of high curvature).

Taken together, these plots give a fairly complete picture of cubic-spline convergence, both through error norms and through visual inspection of the actual fits.

Results

We assess cubic-spline accuracy using the three test functions

$$f_1(x) = \sin(x), \quad x \in [0, 2\pi]; \qquad f_2(x) = x^2(1-x)^2, \quad x \in [-1, 1]; \qquad f_3(x) = \exp(x), \quad x \in [-1, 1].$$

For each, we compare **complete (clamped)** and **not-a-knot** boundary conditions. Each function is analyzed through four complementary plots: (1) log-log errors, (2) scaled errors, (3) reference-line comparison, and (4) interpolated fits, so we can see both asymptotic convergence and endpoint behavior.

Function $f_1(x) = \sin(x)$ on $[0, 2\pi]$

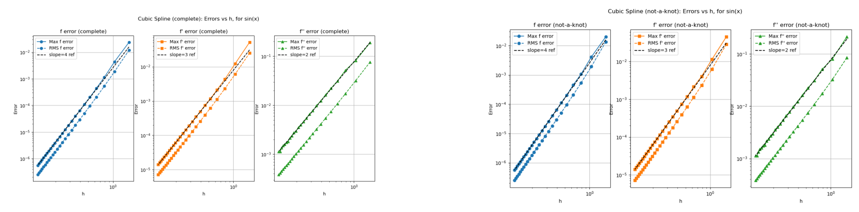


Figure 7: Complete (left) and not-a-knot (right) spline: log-log maximum and RMS errors for f, f', f''; reference slopes $4, 3, 2$ shown. Maximum error decreases roughly as $O(h^4)$, consistent with theory. Endpoint differences in f'' are very small because $\sin(x)$ is smooth and periodic.

Both splines reproduce f with fourth-order accuracy in h. First and second derivative errors follow the expected $O(h^3)$ and $O(h^2)$ trends. RMS errors are slightly smaller than maximum errors, as expected, and deviations at large h are clearly pre-asymptotic behavior rather than a failure of convergence.

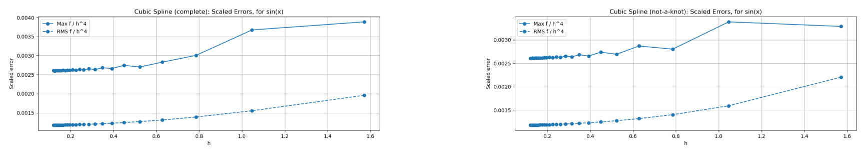
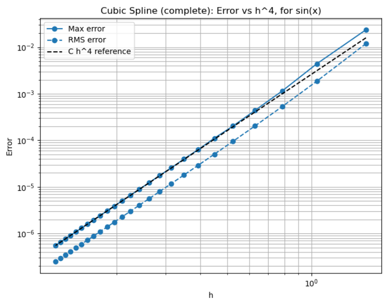


Figure 8: Complete (left) and not-a-knot (right) spline scaled errors E/h^p for f_1. For the complete spline, the scaled error stays roughly constant at small h, indicating that we are in the asymptotic regime. The not-a-knot spline shows similarly flat behavior, with only minor deviations at coarse grids.

Function $f_2(x) = x^2(1-x)^2$ on $[-1, 1]$



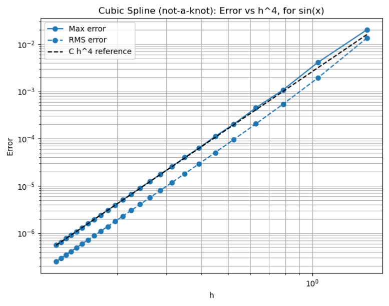
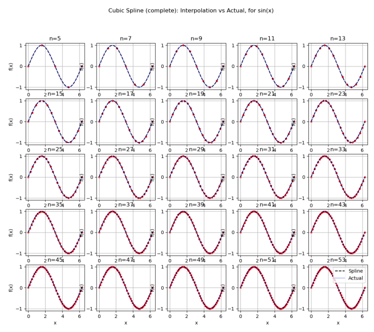


Figure 9: Complete (left) and not-a-knot (right) spline maximum and RMS errors versus the reference line Ch^4 for f_1. The complete spline tracks the Ch^4 line very closely, confirming the predicted convergence rate. The not-a-knot spline behaves almost identically over the full range of grid sizes.



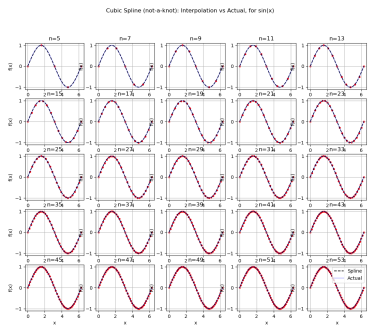


Figure 10: Complete (left) and not-a-knot (right) spline interpolants for f_1 as n increases. The complete spline converges smoothly, with nodes indicated in red. The not-a-knot spline gives visually indistinguishable fits for this periodic function, which explains why both boundary conditions perform almost the same here.

For the not-a-knot spline, the small-h increase in E/h^4 can be understood from the expansion

$$E = Ch^4 + Dh^5 + \dots,$$

where the effective constant C becomes larger when the endpoint derivatives are not enforced. In contrast, the complete spline uses the exact endpoint slopes $f'(x_0)$ and $f'(x_N)$, which keeps C smaller and avoids this effect.

Function $f_3(x) = \exp(x)$ on $[-1, 1]$

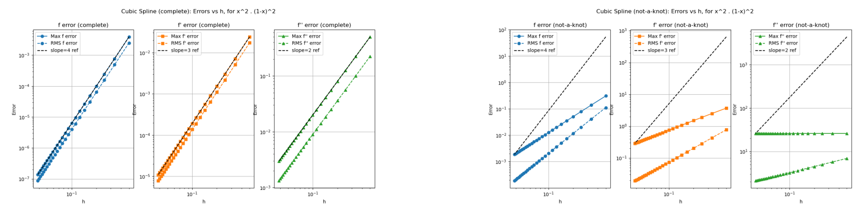


Figure 11: Complete (left) and not-a-knot (right) spline log–log error plots for f_2. The complete spline shows $O(h^4)$ behavior in the maximum error for f and captures the quartic curvature well. The not-a-knot spline has similar asymptotic slopes, but the fine-grid errors are slightly larger because endpoint derivatives are not directly controlled.

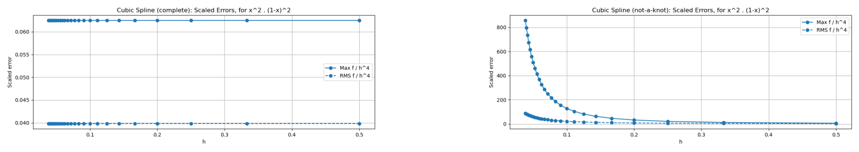


Figure 12: Complete (left) and not-a-knot (right) spline scaled errors for f_2. The complete spline yields nearly flat scaled errors at small h, indicating stable h^4 behavior. For the not-a-knot spline, the scaled errors grow slightly at the finest grids, reflecting a larger effective constant due to endpoint mismatch.

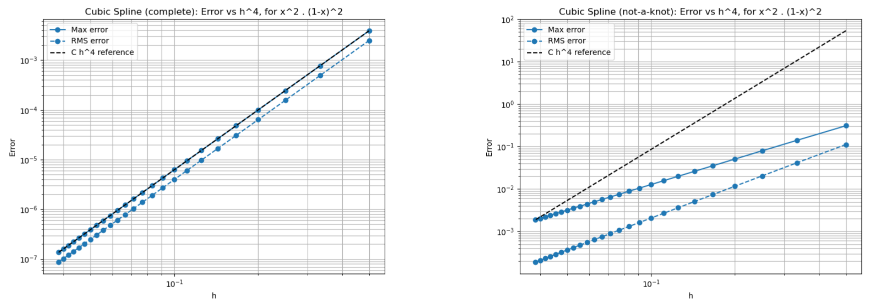


Figure 13: Complete (left) and not-a-knot (right) spline errors versus the reference line Ch^4 for f_2. The complete spline aligns smoothly with the reference line for almost all grid sizes. The not-a-knot spline follows the same trend but sits slightly above the reference line on the finest grids, again due to relaxed endpoint constraints.

Discussion and Conclusion

The computational results give a detailed picture of the convergence properties of cubic splines across the three test functions. Looking at both complete

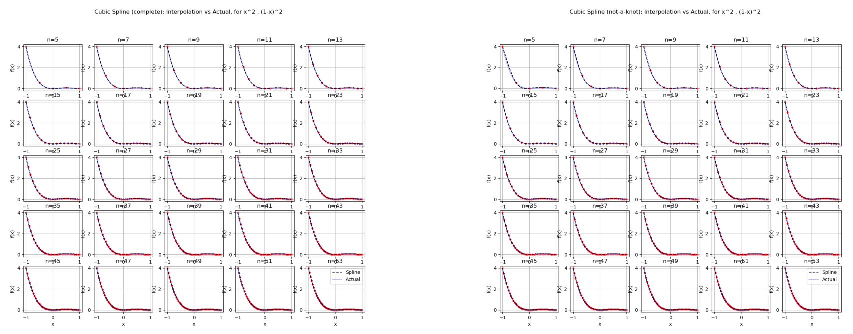


Figure 14: Complete (left) and not-a-knot (right) spline interpolants for f_2 as n increases. The complete spline quickly locks onto the quartic shape. The not-a-knot spline also converges but shows slightly larger deviations near the endpoints for small n, consistent with the error plots.

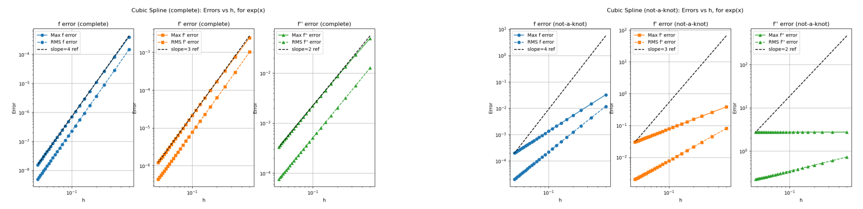


Figure 15: Complete (left) and not-a-knot (right) spline log–log error plots for f_3. The complete spline shows clear $O(h^4)$ convergence in the function error. The not-a-knot spline has similar slopes but exhibits slightly larger errors, especially near the right endpoint where $\exp(x)$ grows fastest.

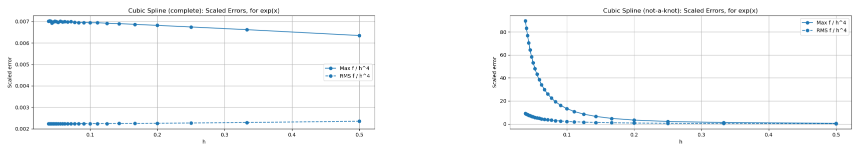


Figure 16: Complete (left) and not-a-knot (right) spline scaled errors for f_3. The complete spline yields nearly flat scaled errors for small h, in line with the expected convergence rates. For the not-a-knot spline, E/h^4 increases slightly as h becomes very small, reflecting the sensitivity of the exponential near $x = 1$ when endpoint derivatives are not enforced.

(clamped) and not-a-knot boundary conditions allows us to separate "pure" order-of-accuracy effects from boundary-condition effects, especially near the endpoints.

For the **complete (clamped) spline**, the behavior is very close to what

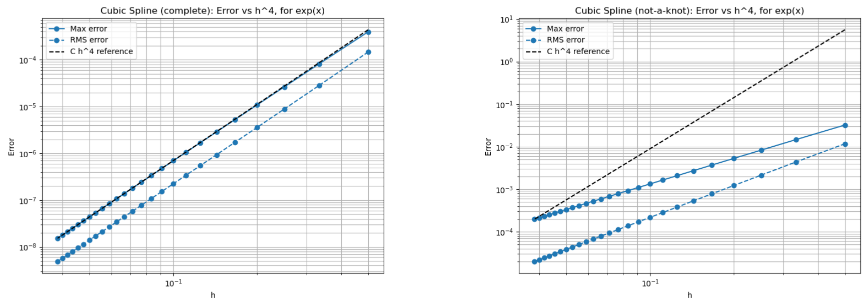


Figure 17: Complete (left) and not-a-knot (right) spline errors versus the reference line Ch^4 for f_3. The complete spline lines up well with the theoretical h^4 reference over several grid refinements. The not-a-knot spline follows the same slope but remains at a slightly higher level, especially on the finest grids.

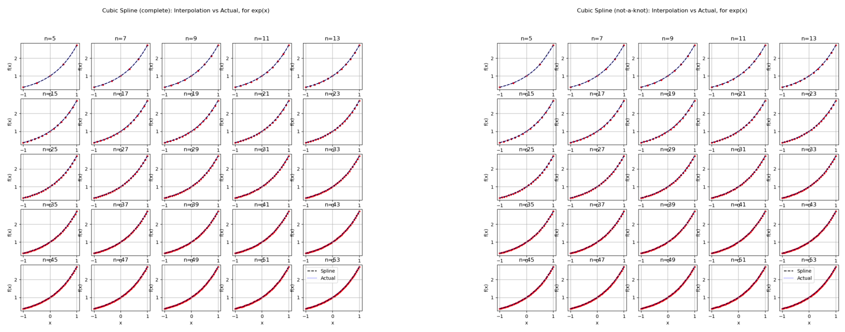


Figure 18: Complete (left) and not-a-knot (right) spline interpolants for f_3 as n increases. The complete spline converges smoothly and accurately captures the exponential growth. The not-a-knot spline also converges, but small endpoint differences persist for moderate n, in agreement with the slightly larger error constants seen earlier.

the standard theory predicts. The maximum error in the function, $\|f - s\|_\infty$, decays like $O(h^4)$, while the RMS error shows a similar trend with slightly smaller values because it averages over the domain. The first derivative error, $\|f' - s'\|_\infty$, decays like $O(h^3)$, and the second derivative error, $\|f'' - s''\|_\infty$, like $O(h^2)$. The interpolated fits back this up visually: as n increases, the splines capture the oscillations in $f_1(x) = \sin(x)$, the interior peak of $f_2(x) = x^2(1-x)^2$, and the rapid growth of $f_3(x) = \exp(x)$ very well. Because complete splines enforce

$$s'(a) = f'(a), \quad s'(b) = f'(b),$$

the endpoint behavior is particularly stable, and the scaled errors E/h^p stay essentially flat once we are in the asymptotic regime.

For the **not-a-knot spline**, the observed orders of convergence for f, f', and f'' are still $O(h^4)$, $O(h^3)$, and $O(h^2)$, respectively, especially away from the endpoints. However, for functions with more interesting endpoint behavior, such as f_2 and f_3, the scaled errors E/h^p tend to rise slightly on the finest grids instead of staying perfectly flat. This can be explained by the fact that not-a-knot conditions do not explicitly enforce f' or f'' at the endpoints. As h gets very small, even small differences in the endpoint derivatives can dominate the scaled error, and in the expansion

$$E \sim Ch^4 + Dh^5 + \dots,$$

the effective constant C becomes larger. Complete splines avoid that by pinning down the endpoint slopes. For the smooth periodic function $f_1(x) = \sin(x)$, these issues are much less visible: the not-a-knot spline behaves almost indistinguishably from the complete spline, since the natural cubic smoothness lines up well with the periodic structure.

A few further points are worth mentioning:

- **Pre-asymptotic behavior on coarse grids:** For large h, the slopes of the log-log curves do not yet match the asymptotic values $4, 3, 2$; higher-order terms in the error expansion are still important. This is particularly noticeable for f_2 and f_3, where the curvature is not uniform across the interval.

- **Round-off limits at very fine grids:** For very small h, the errors eventually plateau instead of continuing to decrease, which is a sign that floating-point round-off dominates. This effect is strongest for the second derivative, where small differences of similar-sized numbers are amplified.

- **RMS vs. maximum errors:** RMS errors are systematically smaller than maximum errors, which is expected because localized peaks (e.g., near endpoints or regions of large curvature) influence the maximum norm but are "diluted" in the RMS norm.

- **Role of boundary conditions:** The choice of boundary condition shows up mainly in the endpoint behavior and the constants in the error estimates. Complete splines give more controlled endpoint derivatives, which keeps the scaled errors flatter and the constants smaller. Not-a-knot splines still achieve the same formal orders, but their endpoint behavior can be slightly less accurate, especially for functions with non-zero slope or strong curvature near the boundaries.

- **Visual fits:** The fit plots confirm that both boundary conditions give good approximations in the interior of the domain as n grows. The main visual differences appear near the endpoints for f_2 and f_3 at moderate n, which matches what we see in the error and scaled-error plots.

Overall, the experiments support the classical error estimates

$$\|f - s\|_\infty = O(h^4), \quad \|f' - s'\|_\infty = O(h^3), \quad \|f'' - s''\|_\infty = O(h^2),$$

for cubic splines, with RMS errors showing the same orders. They also highlight that boundary conditions matter in practice: complete (clamped) splines give cleaner asymptotic behavior and slightly smaller constants, while not-a-knot splines can introduce small endpoint discrepancies even though they retain the same nominal convergence rates.

Code availability

All the code used for these problems is available publicly at this GitHub repository.