Computational Mathematics II (MATH2731)

Dr Andrew Krause & Dr Denis Patterson, Durham University

2025-06-01

Table of contents

Introduction

Welcome to Computational Mathematics II!

This course aims to help you build skills and knowledge in using modern computational methods to do and apply mathematics. It will involve a blend of hands-on computing work and mathematical theory—this theory will include aspects of numerical analysis, computational algebra, and other topics within scientific computing. These areas consist of studying the mathematical properties of the computational representations of mathematical objects (numerical values as well as symbolic manipulations). The computing skills developed in this module will be valuable in all subsequent courses in your degree at Durham and well beyond. We will also introduce you to the use (and abuse) of various computational tools invaluable for doing mathematics, such as AI and searchable websites. While we will encourage you throughout to use all the tools at your disposal, it is **imperative that you understand the details and scope of what you are doing!** You will also develop your communication, presentation, and group-work skills through the various assessments involved in the course – more on that below!

This module has **no final exam**. In fact, there are no exams of any kind. Instead, the summative assessment and associated final grade are entirely based on coursework undertaken during the term. This means that you should expect to spend more time on this course during the term relative to your other modules. We believe this workload distribution is a better way to train the skills we are trying to develop, and as a bonus, you will not need to worry about this course any further once the term ends!

Content

The module's content is divided into six chapters of roughly equal length; some will focus slightly more on theory, while others have a more practical and hands-on nature.

- **Chapter 1: Introduction to Computational Mathematics**
  - Programming basics (including GitHub, and numerical versus symbolic computation)
  - LaTeX, Overleaf, and presenting lab reports
  - Finite-precision arithmetic, rounding error, symbolic representations

- **Chapter 2: Continuous Functions**
  - Interpolation using polynomials – fitting curves to data (Lagrange polynomials, error estimates, convergence, and Chebyshev nodes)
  - Solving nonlinear equations (bisection, fixed-point iteration, Newton's method)
- **Chapter 3: Linear Algebra**
  - Solving linear systems numerically (LU decomposition, Gaussian elimination, conditioning) and symbolically
  - Applications: PageRank, computer graphics
- **Chapter 4: Calculus**
  - Numerical differentiation (finite differences)
  - Numerical integration (quadrature rules, Newton-Cotes formulae)
- **Chapter 5: Ordinary Differential Equations (ODEs)**
  - Numerically approximating solutions of ODEs
  - Timestepping: explicit and implicit methods
  - Stability and convergence order
- **Chapter 6: Selected Further Topics**
  - Intro. to random numbers and stochastic processes
  - Intro. to partial differential equations

Weekly workflow and summative assessment

The final grade for this module is determined as follows:

- **Weekly lab reports (weeks 1-6)** – 20%
- **Weekly e-assessments (weeks 1-6)** – 30%
- **Project (weeks 7-10)** – 50%

Lab reports

Each week for the first six weeks of the course, we will release a short set of exercises based on the lectures from the previous week. Students will be expected to submit a brief report (1-2 pages A4, including figures) with their solutions to the set of exercises – the report will consist of written answers and figures/plots. The reports will be evaluated for correctness and quality of the presentation and communication (quality of figures, clarity of argumentation, etc.).

The lab report for a given week will be due at noon on Monday of the following week (e.g., week one's lab report is due on Monday of week two and so on). Solutions and generalised feedback will be provided to the class on common mistakes and issues arising in each report. Students can also seek detailed feedback on their submission from the lecturers during drop-in sessions and office hours. There will be six lab reports in total, and **your mark is based on your four highest-scoring submissions.**

E-assessments

Each week for the first six weeks of the course, we will release an e-assessment based on the lectures from the previous week. These exercises are designed to complement the lab reports by focusing exclusively on coding skills. The e-assessments will involve submitting code auto-marked by an online grading tool, and hence give immediate feedback. As with the lab reports, **the e-assessment for a given week will be due at noon on Monday of the following week**. There will be six e-assessments in total, and **your mark is based on your four highest-scoring submissions.**

Project

The single largest component of the assessment for this module is the project. **Weeks 7-10 of this course focus exclusively on project work with lectures ending in Week 6.** We will be releasing more detailed instructions on the project submission format and assessment criteria separately, but briefly, the main aspects of the project are as follows:

- There will be approximately eight different project options to choose from across different areas of mathematics (e.g., pure, applied, probability, mathematical physics, etc.); each project has a distinct member of the Maths Department as supervisor.
- Students will submit their preferred project options (ranked choice preferences) in Week 4 of the term and be allocated to projects by the end of Week 6 (there are maximum subscription numbers for each option to ensure equity of supervision).
- Each project consists of two parts: a **guided component** that is completed as part of a small group and an **extension component** that is open-ended and completed as an individual. Group allocations will be done by the lecturers.
- Each group will jointly submit a five-page report for the guided component of the project, and this is worth 60% of the project grade.
- Each student will also submit a three-page report and a six-minute video presentation on their extension component. This submission is worth 40% of the project grade.

In Weeks 7-10 of the term, lectures will be replaced by project workshop sessions during which students can discuss their project with the designated supervisor. This will be an opportunity to discuss progress, ask questions, and seek clarification. Each student only needs to attend the one project drop-in weekly session relevant to their project. Computing drop-in sessions will continue as scheduled in the first six weeks to provide additional support for coding pertinent tasks for the projects – there will be two timetabled computing drop-ins per week and students are encouraged to attend at least one of them.

Lectures, computing drop-ins & project workshops

Lectures will primarily present, explain, and discuss new material (especially theory), but will also feature computer demonstrations of the algorithms and numerical methods. As such, students are encouraged to bring their laptops to lectures to run the examples themselves. Students must bring a laptop or device capable of running code to the computer drop-ins to work on the e-assessments and lab reports.

	Activities	Content
Week 1	Introductory lecture, 2 lectures	Chapter 1
Week 2	3 lectures, 1 computing drop-in	Chapter 2
Week 3	3 lectures, 1 computing drop-in	Chapter 3
Week 4	3 lectures, 1 computing drop-in	Chapter 4
Week 5	3 lectures, 1 computing drop-in	Chapter 5
Week 6	3 lectures, 1 computing drop-in	Chapter 5/6
Week 7	0 lectures, 1 project workshop	Project
Week 8	0 lectures, 1 project workshop	Project
Week 9	0 lectures, 1 project workshop	Project
Week 10	0 lectures, 1 project workshop	Project

Contact details and Reading Materials

If you have questions or need clarification on any of the above, please speak to us during lectures, drop-in sessions, or office hours. Alternatively, email one or both of us at denis.d.patterson@durham.ac.uk or andrew.krause@durham.ac.uk.

The lecture notes are designed to be sufficient and self-contained. Hence, students do not need to purchase a textbook to complete the course successfully. References for additional reading will also be given at the end of each chapter.

The following texts may be useful supplementary references for students wishing to read further into topics from the course:

- Burden, R. L., & Faires, J. D. (1997). *Numerical Analysis* (6th ed.). Pacific Grove, CA: Brooks/Cole Publishing Company.
- Süli, E., & Mayers, D. F. (2003). *An Introduction to Numerical Analysis*. Cambridge: Cambridge University Press.

Acknowledgements

We are indebted to Prof. Anthony Yeates (Durham) whose numerical analysis notes formed the basis of several chapters of the coures notes.

1 Floating Point Arithmetic

The goal of this chapter is to explore and begin to answer the following question:

How do we represent numbers on a computer?

Integers can be represented exactly, up to some maximum size.

If 1 bit (binary digit) is used to store the sign $\pm$, the largest possible number is

$$1 \times 2^{62} + 1 \times 2^{61} + \ldots + 1 \times 2^1 + 1 \times 2^0 = 2^{63} - 1.$$

In contrast to the integers, only a subset of real numbers within any given interval can be represented exactly.

Note

Some modern languages (such as Python) automatically promote large integers to arbitrary precision ("long"), but most statically-typed languages (C, Java, Matlab, etc.) do not; an **overflow** will occur and the type remains fixed.

Note

A statically typed language is one in which the type of every variable is determined before the program runs.

1.1 Fixed-point numbers

In everyday life, we tend to use a **fixed point** representation

$$x = \pm(d_1 d_2 \cdots d_{k-1}.d_k \cdots d_n)_\beta, \quad \text{where} \quad d_1, \ldots, d_n \in \{0, 1, \ldots, \beta - 1\}.$$

Here β is the base (e.g. 10 for decimal arithmetic or 2 for binary).

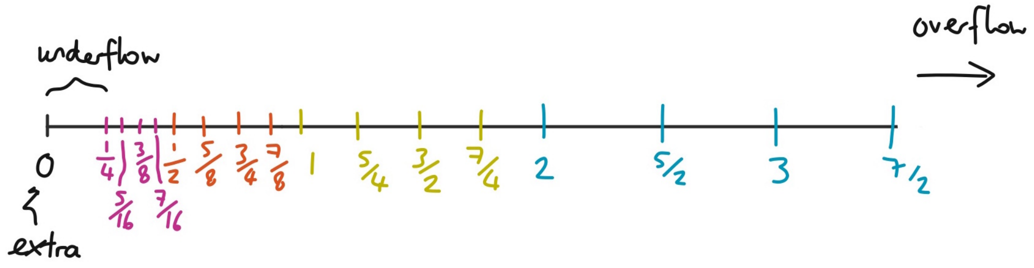
If we require that $d_1 \neq 0$ unless $k = 2$, then every number has a unique representation of this form, except for infinite trailing sequences of digits $\beta - 1$.

1.2 Floating-point numbers

Computers use a **floating-point** representation. Only numbers in a **floating-point number system** $F \subset \mathbb{R}$ can be represented exactly, where

$$F = \{ \pm (0.d_1 d_2 \cdots d_m)_\beta \beta^e \mid \beta, d_i, e \in \mathbb{Z},\ 0 \le d_i \le \beta - 1,\ e_{\min} \le e \le e_{\max} \}.$$

Here $(0.d_1 d_2 \cdots d_m)_\beta$ is called the **fraction** (or **significand** or **mantissa**), β is the base, and e is the **exponent**. This can represent a much larger range of numbers than a fixed-point system of the same size, although at the cost that the numbers are not equally spaced. If $d_1 \neq 0$ then each number in F has a unique representation and F is called **normalised**.



Note

Notice that the spacing between numbers jumps by a factor β at each power of β. The largest possible number is $(0.111)_2 2^2 = (\frac{1}{2} + \frac{1}{4} + \frac{1}{8})(4) = \frac{7}{2}$. The smallest non-zero number is $(0.100)_2 2^{-1} = \frac{1}{2}(\frac{1}{2}) = \frac{1}{4}$.

Here $\beta = 2$, and there are 52 bits for the fraction, 11 for the exponent, and 1 for the sign. The actual format used is

$$\pm(1.d_1 \cdots d_{52})_2 2^{e-1023} = \pm(0.1d_1 \cdots d_{52})_2 2^{e-1022}, \quad e = (e_1 e_2 \cdots e_{11})_2.$$

When $\beta = 2$, the first digit of a normalized number is always 1, so doesn't need to be stored in memory. The **exponent bias** of 1022 means that the actual exponents are in the range -1022 to 1025, since $e \in [0, 2047]$. Actually the exponents -1022 and 1025 are used to store ± 0 and $\pm\infty$ respectively.

The smallest non-zero number in this system is $(0.1)_2 2^{-1021} \approx 2.225 \times 10^{-308}$, and the largest number is $(0.1 \cdots 1)_2 2^{1024} \approx 1.798 \times 10^{308}$.

Note

IEEE stands for Institute of Electrical and Electronics Engineers. Matlab uses the IEEE 754 standard for floating point arithmetic. The automatic 1 is sometimes called the "hidden bit". The exponent bias avoids the need to store the sign of the exponent.

Numbers outside the finite set F cannot be represented exactly. If a calculation falls below the lower non-zero limit (in absolute value), it is called **underflow**, and usually set to 0. If it falls above the upper limit, it is called **overflow**, and usually results in a floating-point exception.

Note

Ariane 5 rocket failure (1996): The maiden flight ended in failure. Only 40 seconds after initiation, at altitude 3700m, the launcher veered off course and exploded. The cause was a software exception during data conversion from a 64-bit float to a 16-bit integer. The converted number was too large to be represented, causing an exception.

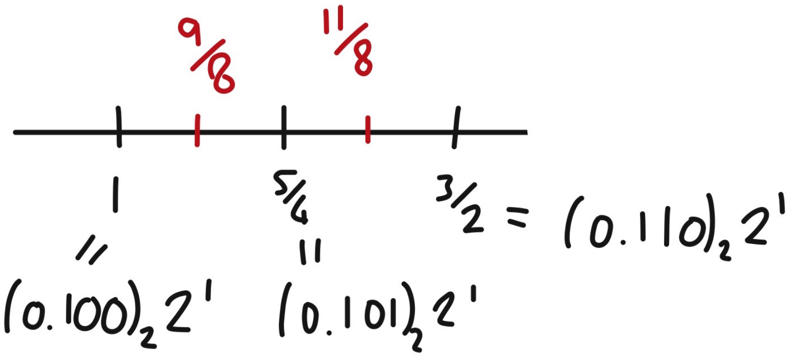
Note

In IEEE arithmetic, some numbers in the "zero gap" can be represented using $e = 0$, since only two possible fraction values are needed for ± 0. The other fraction values may be used with first (hidden) bit 0 to store a set of so-called **subnormal** numbers.

The mapping from $\mathbb{R}$ to F is called **rounding** and denoted $\mathrm{fl}(x)$. Usually it is simply the nearest number in F to x. If x lies exactly midway between two numbers in F, a method of breaking ties is required. The IEEE standard specifies *round to nearest even*—i.e., take the neighbour with last digit 0 in the fraction.

Note

This avoids statistical bias or prolonged drift.



$\frac{9}{8} = (1.001)_2$ has neighbours $1 = (0.100)_2 2^1$ and $\frac{5}{4} = (0.101)_2 2^1$, so is rounded down to 1.
$\frac{11}{8} = (1.011)_2$ has neighbours $\frac{5}{4} = (0.101)_2 2^1$ and $\frac{3}{2} = (0.110)_2 2^1$, so is rounded up to $\frac{3}{2}$.

Note

Vancouver stock exchange index: In 1982, the index was established at 1000. By November 1983, it had fallen to 520, even though the exchange seemed to be doing well. Explanation: the index was rounded *down* to 3 digits at every recomputation. Since the errors were always in the same direction, they added up to a large error over time. Upon recalculation, the index doubled!

1.3 Significant figures

When doing calculations without a computer, we often use the terminology of **significant figures**. To count the number of significant figures in a number x, start with the first non-zero digit from the left, and count all the digits thereafter, including final zeros if they are after the decimal point.

To round x to n s.f., replace x by the nearest number with n s.f. An approximation $\hat{x}$ of x is "correct to n s.f." if both $\hat{x}$ and x round to the same number to n s.f.

1.4 Rounding error

If $|x|$ lies between the smallest non-zero number in F and the largest number in F, then

$$\mathrm{fl}(x) = x(1+\delta),$$

where the relative error incurred by rounding is

$$|\delta| = \frac{|\mathrm{fl}(x) - x|}{|x|}.$$

Note

Relative errors are often more useful because they are scale invariant. E.g., an error of 1 hour is irrelevant in estimating the age of this lecture theatre, but catastrophic in timing your arrival at the lecture.

Now x may be written as $x = (0.d_1 d_2 \cdots)_\beta \beta^e$ for some $e \in [e_{\min}, e_{\max}]$, but the fraction will not terminate after m digits if $x \notin F$. However, this fraction will differ from that of $\mathrm{fl}(x)$ by at most $\frac{1}{2}\beta^{-m}$, so

$$|\mathrm{fl}(x) - x| \le \tfrac{1}{2}\beta^{-m}\beta^e \quad \Longrightarrow \quad |\delta| \le \tfrac{1}{2}\beta^{1-m}.$$

Here we used that the fractional part of $|x|$ is at least $(0.1)_\beta \equiv \beta^{-1}$. The number $\epsilon_{\mathrm{M}} = \frac{1}{2}\beta^{1-m}$ is called the **machine epsilon** (or **unit roundoff**), and is independent of x. So the relative rounding error satisfies

$$|\delta| \leq \epsilon_{\mathrm{M}}.$$

Note

To check the machine epsilon value in Matlab you can just type 'eps' in the command line, which will return the value 2.2204e-16.

Note

The name "unit roundoff" arises because β^{1-m} is the distance between 1 and the next number in the system.

When adding/subtracting/multiplying/dividing two numbers in F, the result will not be in F in general, so must be rounded.

Let us multiply $x = \frac{5}{8}$ and $y = \frac{7}{8}$. We have

$$xy = \frac{35}{64} = \frac{1}{2} + \frac{1}{32} + \frac{1}{64} = (0.100011)_2.$$

This has too many significant digits to represent in our system, so the best we can do is round the result to $\mathrm{fl}(xy) = (0.100)_2 = \frac{1}{2}$.

Note

Typically additional digits are used during the computation itself, as in our example.

For $\circ = +, -, \times, \div$, IEEE standard arithmetic requires rounded exact operations, so that

$$\mathrm{fl}(x \circ y) = (x \circ y)(1 + \delta), \quad |\delta| \leq \epsilon_{\mathrm{M}}.$$

1.5 Loss of significance

You might think that the above guarantees the accuracy of calculations to within ϵ_{M}, but this is true only if x and y are themselves exact. In reality, we are probably starting from $\bar{x} = x(1 + \delta_1)$ and $\bar{y} = y(1 + \delta_2)$, with $|\delta_1|, |\delta_2| \leq \epsilon_{\mathrm{M}}$. In that case, there is an error even before we round the result, since

$$\begin{aligned} \bar{x} \pm \bar{y} &= x(1 + \delta_1) \pm y(1 + \delta_2) \\ &= (x \pm y)\left(1 + \frac{x\delta_1 \pm y\delta_2}{x \pm y}\right). \end{aligned}$$

If the correct answer $x \pm y$ is very small, then there can be an arbitrarily large relative error in the result, compared to the errors in the initial $\bar{x}$ and $\bar{y}$. In particular, this relative error can be much larger than ϵ_{M}. This is called **loss of significance**, and is a major cause of errors in floating-point calculations.

To 4 s.f., the roots are

$$x_1 = 28 + \sqrt{783} = 55.98, \quad x_2 = 28 - \sqrt{783} = 0.01786.$$

However, working to 4 s.f. we would compute $\sqrt{783} = 27.98$, which would lead to the results

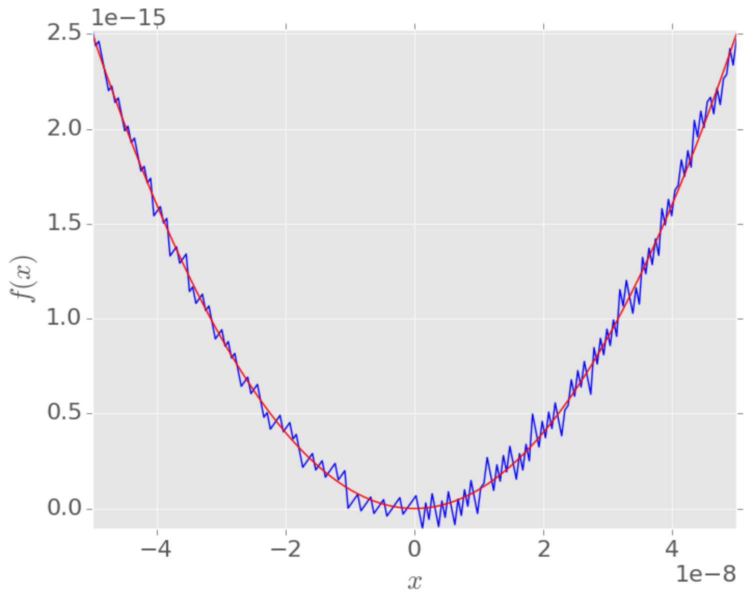
$$\bar{x}_1 = 55.98, \quad \bar{x}_2 = 0.02000.$$

The smaller root is not correct to 4 s.f., because of cancellation error. One way around this is to note that $x^2 - 56x + 1 = (x - x_1)(x - x_2)$, and compute x_2 from $x_2 = 1/x_1$, which gives the correct answer.

Note

Note that the error crept in when we rounded $\sqrt{783}$ to 27.98, because this removed digits that would otherwise have been significant after the subtraction.

Let us plot this function in the range $-5 \times 10^{-8} \leq x \leq 5 \times 10^{-8}$ – even in IEEE double precision arithmetic we find significant errors, as shown by the blue curve:



The red curve shows the correct result approximated using the Taylor series

$$\begin{aligned} f(x) &= \left(1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots\right) - \left(1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots\right) - x \\ &\approx x^2 + \frac{x^3}{6}. \end{aligned}$$

This avoids subtraction of nearly equal numbers.

Note

We will look in more detail at polynomial approximations in the next section.

Note that floating-point arithmetic violates many of the usual rules of real arithmetic, such as $(a+b)+c = a+(b+c)$.

$$\begin{aligned} \mathrm{fl}\big[(5.9+5.5)+0.4\big] &= \mathrm{fl}\big[\mathrm{fl}(11.4)+0.4\big] = \mathrm{fl}(11.0+0.4) = 11.0, \\ \mathrm{fl}\big[5.9+(5.5+0.4)\big] &= \mathrm{fl}\big[5.9+5.9\big] = \mathrm{fl}(11.8) = 12.0. \end{aligned}$$

In $\mathbb{R}$, the average of two numbers always lies between the numbers. But if we work to 3 decimal digits,

$$\mathrm{fl}\left(\frac{5.01+5.02}{2}\right) = \frac{\mathrm{fl}(10.03)}{2} = \frac{10.0}{2} = 5.0.$$

The moral of the story is that sometimes care is needed to ensure that we carry out a calculation accurately and as intended!

Knowledge checklist

Key topics:

1. Integer and floating point representations of real numbers on computers.
2. Overflow, underflow and loss of significance.

Key skills:

- Understanding and distinguishing integer, fixed-point, and floating-point representations.
- Analyzing the effects of rounding and machine epsilon in calculations.
- Diagnosing and managing rounding errors, overflow, and underflow.

2 Continuous Functions

The goal of this chapter is to explore and begin to answer the following question:

How do we represent mathematical functions on a computer?

Polynomial Interpolation: Motivation

If f is a polynomial of degree n,

$$f(x) = p_n(x) = a_0 + a_1 x + \ldots + a_n x^n,$$

then we only need to store the $n+1$ coefficients $a_0, \ldots, a_n$. Operations such as taking the derivative or integrating f are also convenient. The idea in this chapter is to find a polynomial that approximates a general function f. For a continuous function f on a bounded interval, this is always possible if you take a high enough degree polynomial:

Theorem 2.1: Weierstrass Approximation Theorem (1885)

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

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

Note

This may be proved using an explicit sequence of polynomials, called Bernstein polynomials.

If f is not continuous, then something other than a polynomial is required, since polynomials can't handle asymptotic behaviour.

Note

To approximate functions like $1/x$, there is a well-developed theory of rational function interpolation, which is beyond the scope of this course.

In this chapter, we look for a suitable polynomial p_n by **interpolation**—that is, requiring $p_n(x_i) = f(x_i)$ at a finite set of points x_i, usually called **nodes**. Sometimes we will also require the derivative(s) of p_n to match those of f. This type of function approximation where we want to match values of the function that we know at particular points is very natural in many applications. For example, weather forecasts involve numerically solving huge systems of partial differential equations (PDEs), which means actually solving them on a discrete grid of points. If we want weather predictions between grid points, we must **interpolate**. Figure Figure 2.1 shows the spatial resolutions of a range of current and past weather models produced by the UK Met Office.

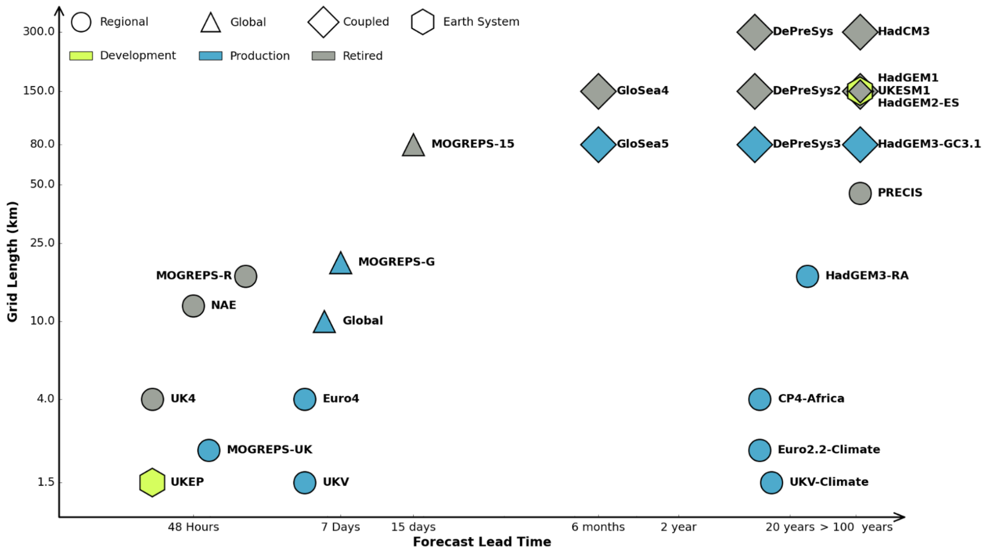


Figure 2.1: Chart showing a range of weather models produce by the UK Met Office. Even the highest spatial resolution models have more than 1.5km between grid point due to computational constraints.

Taylor series

A truncated Taylor series is (in some sense) the simplest interpolating polynomial since it uses only a single node x_0, although it does require p_n to match both f and some of its derivatives.

We can approximate this using a Taylor series about the point $x_0 = 0$, which is

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

This comes from writing

$$f(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \dots,$$

then differentiating term-by-term and matching values at x_0:

$$\begin{aligned} f(x_0) &= a_0, \\ f'(x_0) &= a_1, \\ f''(x_0) &= 2a_2, \\ f'''(x_0) &= 3(2)a_3, \\ &\vdots \\ \implies f(x) = f(x_0) + f'(x_0)(x-x_0) &+ \frac{f''(x_0)}{2!}(x-x_0)^2 + \frac{f'''(x_0)}{3!}(x-x_0)^3 + \dots . \end{aligned}$$

So

$$\begin{aligned} 1 \text{ term} &\implies f(0.1) \approx 0.1, \\ 2 \text{ terms} &\implies f(0.1) \approx 0.1 - \frac{0.1^3}{6} = 0.099833\dots, \\ 3 \text{ terms} &\implies f(0.1) \approx 0.1 - \frac{0.1^3}{6} + \frac{0.1^5}{120} = 0.09983341\dots . \end{aligned}$$

The next term will be $-0.1^7/7! \approx -10^{-7}/10^3 = -10^{-10}$, which won't change the answer to 6 s.f.

Note

The exact answer is $\sin(0.1) = 0.09983341$.

Mathematically, we can write the remainder as follows.

Theorem 2.2: Taylor's Theorem

Let f be $n+1$ times differentiable on (a,b), and let $f^{(n)}$ be continuous on $[a,b]$. If $x, x_0 \in [a,b]$ then there exists $\xi \in (a,b)$ 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 called the **Lagrange form** of the remainder. Note that the unknown number ξ depends on x.

For $f(x) = \sin(x)$, we found the Taylor polynomial $p_6(x) = x - x^3/3! + x^5/5!$, and $f^{(7)}(x) = -\sin(x)$. So we have

$$|f(x) - p_6(x)| = \left| \frac{f^{(7)}(\xi)}{7!}(x-x_0)^7 \right|$$

for some ξ between x_0 and x. For $x = 0.1$, we have

$$|f(0.1) - p_6(0.1)| = \frac{1}{5040}(0.1)^7 |f^{(7)}(\xi)| \quad \text{for some } \xi \in [0, 0.1].$$

Since $|f^{(7)}(\xi)| = |\sin(\xi)| \leq 1$, we can say, before calculating, that the error satisfies

$$|f(0.1) - p_6(0.1)| \leq 1.984 \times 10^{-11}.$$

Note

The actual error is 1.983×10^{-11}, so this is a tight estimate.

Since this error arises from approximating f with a truncated series, rather than due to rounding, it is known as **truncation error**. Note that it tends to be lower if you use more terms (larger n), or if the function oscillates less (smaller $f^{(n+1)}$ on the interval (x_0, x)).

Error estimates like the Lagrange remainder play an important role in numerical analysis and computation, so it is important to understand where it comes from. The number ξ will ultimately come from Rolle's theorem, which is a special case of the mean value theorem from first-year calculus:

Theorem 2.3: Rolle's Theorem

If f is continuous on $[a, b]$ and differentiable on (a, b), with $f(a) = f(b) = 0$, then there exists $\xi \in (a, b)$ with $f'(\xi) = 0$.

Note

Note that Rolle's Theorem does not tell us what the value of ξ might actually be, so in practice we must take some kind of worst case estimate to get an error bound, e.g. calculate the max value of $f'(\xi)$ over the range of possible ξ values.

2.1 Polynomial interpolation

The classical problem of **polynomial interpolation** is to find a polynomial

$$p_n(x) = a_0 + a_1 x + \ldots + a_n x^n = \sum_{k=0}^{n} a_k x^k$$

that interpolates our function f at a finite set of nodes $\{x_0, x_1, \ldots, x_m\}$. In other words, $p_n(x_i) = f(x_i)$ at each of the nodes x_i. Since the polynomial has $n + 1$ unknown coefficients, we expect to need $n + 1$ distinct nodes, so let us assume that $m = n$.

Here we have two nodes x_0, x_1, and seek a polynomial $p_1(x) = a_0 + a_1 x$. Then the interpolation conditions require that

$$\begin{cases} p_1(x_0) = a_0 + a_1 x_0 = f(x_0) \\ p_1(x_1) = a_0 + a_1 x_1 = f(x_1) \end{cases} \implies \quad p_1(x) = \frac{x_1 f(x_0) - x_0 f(x_1)}{x_1 - x_0} + \frac{f(x_1) - f(x_0)}{x_1 - x_0} x.$$

For general n, the interpolation conditions require

$$\begin{array}{llllll} a_0 & +a_1 x_0 & +a_2 x_0^2 & + \ldots & +a_n x_0^n & = f(x_0), \\ a_0 & +a_1 x_1 & +a_2 x_1^2 & + \ldots & +a_n x_1^n & = f(x_1), \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ a_0 & +a_1 x_n & +a_2 x_n^2 & + \ldots & +a_n x_n^n & = f(x_n), \end{array}$$

so we have to solve

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}.$$

This is called a **Vandermonde matrix**. The determinant of this matrix is

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

which is non-zero provided the nodes are all distinct. This establishes an important result, where $\mathcal{P}_n$ denotes the space of all real polynomials of degree $\le n$.

Theorem 2.4: Existence/uniqueness

Given $n+1$ distinct nodes $x_0, x_1, \ldots, x_n$, there is a unique polynomial $p_n \in \mathcal{P}_n$ that interpolates $f(x)$ at these nodes.

We may also prove uniqueness by the following elegant argument.

Proof (Uniqueness part of Existence/Uniqueness Theorem):
Suppose that in addition to p_n there is another interpolating polynomial $q_n \in \mathcal{P}_n$. Then the difference $r_n := p_n - q_n$ is also a polynomial with degree $\le n$. But we have

$$r_n(x_i) = p_n(x_i) - q_n(x_i) = f(x_i) - f(x_i) = 0 \quad \text{for } i = 0, \ldots, n,$$

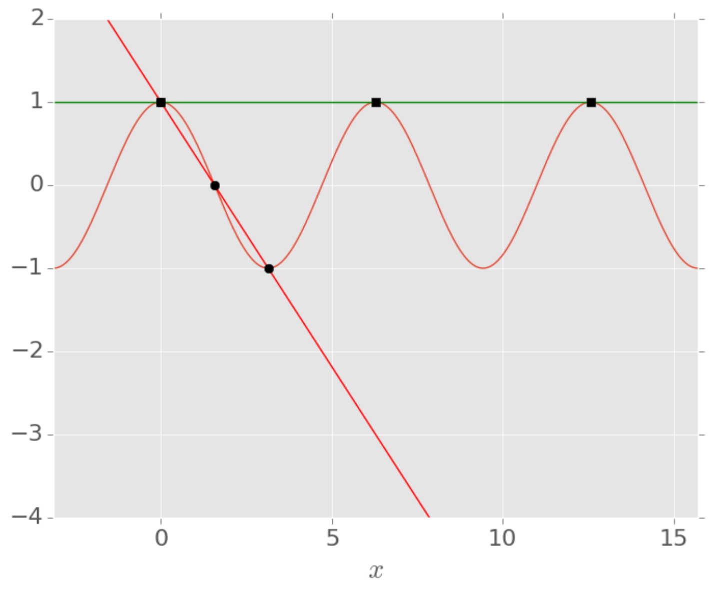
so $r_n(x)$ has $n+1$ roots. From the Fundamental Theorem of Algebra, this is possible only if $r_n(x) \equiv 0$, which implies that $q_n = p_n$.

Note

Note that the unique polynomial through $n+1$ points may have degree $< n$. This happens when $a_0 = 0$ in the solution to the Vandermonde system above.

We have $x_0 = 0$, $x_1 = \frac{\pi}{2}$, $x_2 = \pi$, so $f(x_0) = 1$, $f(x_1) = 0$, $f(x_2) = -1$. Clearly the unique interpolant is a straight line $p_2(x) = 1 - \frac{2}{\pi}x$.

If we took the nodes $\{0, 2\pi, 4\pi\}$, we would get a constant function $p_2(x) = 1$.



One way to compute the interpolating polynomial would be to solve the Vandermonde system above, e.g. by Gaussian elimination. However, this is not recommended. In practice, we choose a different basis for p_n; there are two common and effective choices due to Lagrange and Newton.

Note

The Vandermonde matrix arises when we write p_n in the **natural basis** $\{1, x, x^2, \ldots\}$, but we could also choose to work in some other basis…

2.1.1 Lagrange Polynomials

This uses a special basis of polynomials $\{\ell_k\}$ in which the interpolation equations reduce to the identity matrix. In other words, the coefficients in this basis are just the function values,

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

Example 2.1: Linear interpolation again.

We can re-write our linear interpolant to separate out the function values:

$$p_1(x) = \underbrace{\frac{x - x_1}{x_0 - x_1}}_{\ell_0(x)} f(x_0) + \underbrace{\frac{x - x_0}{x_1 - x_0}}_{\ell_1(x)} f(x_1).$$

Then ℓ_0 and ℓ_1 form the necessary basis. In particular, they have the property that

$$\ell_0(x_i) = \begin{cases} 1 & \text{if } i = 0, \\ 0 & \text{if } i = 1, \end{cases} \qquad \ell_1(x_i) = \begin{cases} 0 & \text{if } i = 0, \\ 1 & \text{if } i = 1, \end{cases}$$

For general n, the $n+1$ **Lagrange polynomials** are defined as a product

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

By construction, they have the property that

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

From this, it follows that the interpolating polynomial may be written as above.

i Note

By the Existence/Uniqueness Theorem, the Lagrange polynomials are the *unique* polynomials with this property.

Example 2.2: Compute the quadratic interpolating polynomial to $f(x) = \cos(x)$ with nodes $\{-\frac{\pi}{4}, 0, \frac{\pi}{4}\}$ using Lagrange polynomials.

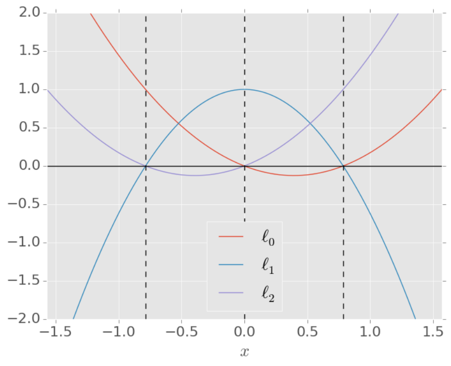
The Lagrange polynomials of degree 2 for these nodes are

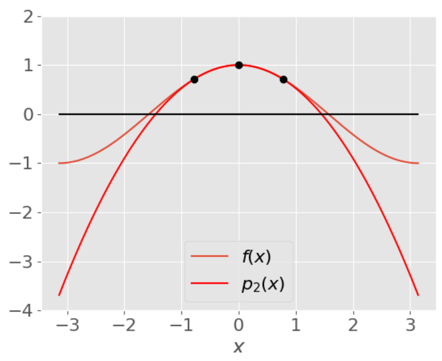
$$\begin{aligned}
\ell_0(x) &= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} = \frac{x(x-\frac{\pi}{4})}{\frac{\pi}{4}\cdot\frac{\pi}{2}},\\
\ell_1(x) &= \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} = \frac{(x+\frac{\pi}{4})(x-\frac{\pi}{4})}{-\frac{\pi}{4}\cdot\frac{\pi}{4}},\\
\ell_2(x) &= \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}\\
&= \frac{x(x+\frac{\pi}{4})}{\frac{\pi}{2}\cdot\frac{\pi}{4}}.
\end{aligned}$$

So the interpolating polynomial is

$$\begin{aligned}
p_2(x) &= f(x_0)\ell_0(x) + f(x_1)\ell_1(x) + f(x_2)\ell_2(x)\\
&= \frac{1}{\sqrt{2}}\frac{8}{\pi^2}x(x-\tfrac{\pi}{4}) - \frac{16}{\pi^2}(x+\tfrac{\pi}{4})(x-\tfrac{\pi}{4}) + \frac{1}{\sqrt{2}}\frac{8}{\pi^2}x(x+\tfrac{\pi}{4}) = \frac{16}{\pi^2}\left(\frac{1}{\sqrt{2}}-1\right)x^2 + 1.
\end{aligned}$$

The Lagrange polynomials and the resulting interpolant are shown below:





Note

Lagrange polynomials were actually discovered by Edward Waring in 1776 and rediscovered by Euler in 1783, before they were published by Lagrange himself in 1795; a classic example of Stigler's law of eponymy!

The Lagrange form of the interpolating polynomial is easy to write down, but expensive to evaluate since all of the ℓ_k must be computed. Moreover, changing any of the nodes means that the ℓ_k must all be recomputed from scratch, and similarly for adding a new node (moving to higher degree).

2.1.2 Newton/Divided-Difference Polynomials

It would be easy to increase the degree of p_n if

$$p_{n+1}(x) = p_n(x) + g_{n+1}(x), \quad \text{where } g_{n+1} \in \mathcal{P}_{n+1}.$$

From the interpolation conditions, we know that

$$g_{n+1}(x_i) = p_{n+1}(x_i) - p_n(x_i) = f(x_i) - f(x_i) = 0 \quad \text{for } i = 0, \ldots, n,$$

so

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

The coefficient a_{n+1} is determined by the remaining interpolation condition at x_{n+1}, so

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

The polynomial $(x - x_0)(x - x_1) \cdots (x - x_n)$ is called a **Newton polynomial**. These form a new basis

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

The **Newton form** of the interpolating polynomial is then

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

Notice that a_k depends only on $x_0, \ldots x_k$, so we can construct first a_0, then a_1, etc.

It turns out that the a_k are easy to compute, but it will take a little work to derive the method. We define the **divided difference** $f[x_0, x_1, \ldots, x_k]$ to be the coefficient of x^k in the polynomial interpolating f at nodes $x_0, \ldots, x_k$. It follows that

$$f[x_0, x_1, \ldots, x_k] = a_k,$$

where a_k is the coefficient in the Newton form above.

Example 2.3: Compute the Newton interpolating polynomial at two nodes.

$$\begin{aligned} f[x_0] &= a_0 = f(x_0), \\ f[x_0, x_1] &= a_1 = \frac{f(x_1) - p_0(x_1)}{x_1 - x_0} = \frac{f(x_1) - a_0}{x_1 - x_0} = \frac{f[x_1] - f[x_0]}{x_1 - x_0}. \end{aligned}$$

So the **first-order** divided difference $f[x_0, x_1]$ is obtained from the **zeroth-order** differences $f[x_0]$, $f[x_1]$ by subtracting and dividing, hence the name "divided difference".

Example 2.4: Compute the Newton interpolating polynomial at three nodes.

Continuing from the previous example, we find

$$\begin{aligned} f[x_0,x_1,x_2] = a_2 &= \frac{f(x_2)-p_1(x_2)}{(x_2-x_0)(x_2-x_1)} = \frac{f(x_2)-a_0-a_1(x_2-x_0)}{(x_2-x_0)(x_2-x_1)} \\ &= \ldots = \frac{1}{x_2-x_0}\left(\frac{f[x_2]-f[x_1]}{x_2-x_1} - \frac{f[x_1]-f[x_0]}{x_1-x_0}\right) \\ &= \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0}. \end{aligned}$$

So again, we subtract and divide.

In general, we have the following.

Theorem 2.5

For $k>0$, the divided differences satisfy

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

Proof:
Without loss of generality, we relabel the nodes so that $i=0$. So we want to prove that

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

The trick is to write the interpolant with nodes $x_0,\ldots,x_k$ in the form

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

where $q_{k-1}\in\mathcal{P}_{k-1}$ interpolates f at the subset of nodes $x_0,x_1,\ldots,x_{k-1}$ and $\tilde{q}_{k-1}\in\mathcal{P}_{k-1}$ interpolates f at the subset $x_1,x_2,\ldots,x_k$. If this holds, then matching the coefficient of x^k on each side will give the divided difference formula, since, e.g., the leading coefficient of q_{k-1} is $f[x_0,\ldots,x_{k-1}]$. To see that p_k may really be written this way, note that

$$\begin{aligned} p_k(x_0) &= q_{k-1}(x_0) = f(x_0), \\ p_k(x_k) &= \tilde{q}_{k-1}(x_k) = f(x_k), \\ p_k(x_i) &= \frac{(x_k-x_i)q_{k-1}(x_i)+(x_i-x_0)\tilde{q}_{k-1}(x_i)}{x_k-x_0} = f(x_i) \quad \text{for } i=1,\ldots,k-1. \end{aligned}$$

Since p_k agrees with f at the $k+1$ nodes, it is the unique interpolant in $\mathcal{P}_k$.

Theorem above gives us our convenient method, which is to construct a **divided-difference table**.

Example 2.5: Construct the Newton polynomial at the nodes $\{-1, 0, 1, 2\}$ and with corresponding function values $\{5, 1, 1, 11\}$

We construct a divided-difference table as follows.

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

The coefficients of the p_3 lie at the top of each column, so

$$\begin{aligned}
p_3(x) = f[x_0] &+ f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) \\
&+ f[x_0, x_1, x_2, x_3](x - x_0)(x - x_1)(x - x_2) \\
&= 5 - 4(x + 1) + 2x(x + 1) + x(x + 1)(x - 1).
\end{aligned}$$

Now suppose we add the extra nodes $\{-2, 3\}$ with data $\{5, 35\}$. All we need to do to compute p_5 is add two rows to the bottom of the table — there is no need to recalculate the rest. This gives

$$\begin{array}{ccccccc}
-1 & 5 & & & & & \\
 & & -4 & & & & \\
0 & 1 & & 2 & & & \\
 & & 0 & & 1 & & \\
1 & 1 & & 5 & & -\frac{1}{12} & \\
 & & 10 & & \frac{13}{12} & & 0 \\
2 & 11 & & \frac{17}{6} & & -\frac{1}{12} & \\
 & & \frac{3}{2} & & \frac{5}{6} & & \\
-2 & 5 & & \frac{9}{2} & & & \\
 & & 6 & & & & \\
3 & 35 & & & & &
\end{array}$$

The new interpolating polynomial is

$$p_5(x) = p_3(x) - \tfrac{1}{12}x(x + 1)(x - 1)(x - 2).$$

Note

Notice that the x^5 coefficient vanishes for these particular data, meaning that they are consistent with $f \in \mathcal{P}_4$.

Note

Note that the value of $f[x_0, x_1, \ldots, x_k]$ is independent of the order of the nodes in the table. This follows from the uniqueness of p_k.

Divided differences are actually approximations for *derivatives* of f. In the limit that the nodes all coincide, the Newton form of $p_n(x)$ becomes the Taylor polynomial.

2.1.3 Interpolation Error

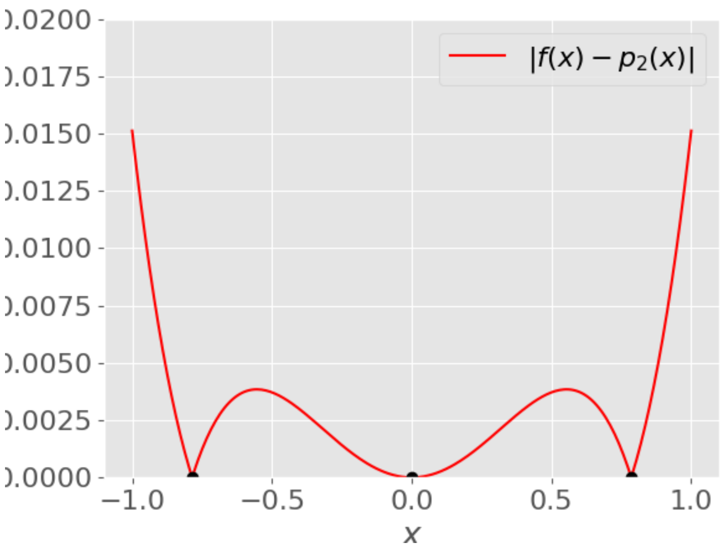
The goal here is to estimate the error $|f(x) - p_n(x)|$ when we approximate a function f by a polynomial interpolant p_n. Clearly this will depend on x.

Example 2.6: Quadratic interpolant for $f(x) = \cos(x)$ with $\{-\frac{\pi}{4}, 0, \frac{\pi}{4}\}$.

From Section 2.1.1, we have $p_2(x) = \frac{16}{\pi^2}\left(\frac{1}{\sqrt{2}} - 1\right)x^2 + 1$, so the error is

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

This is shown below:



Clearly the error vanishes at the nodes themselves, but note that it generally does better near the middle of the set of nodes — this is quite typical behaviour.

We can adapt the proof of Taylor's theorem to get a quantitative error estimate.

Theorem 2.6: Cauchy's Interpolation Error Theorem

Let $p_n \in \mathcal{P}_n$ be the unique polynomial interpolating $f(x)$ at the $n+1$ distinct nodes $x_0, x_1, \ldots, x_n \in [a,b]$, and let f be 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

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

This looks similar to the error formula for Taylor polynomials (see Taylor's Theorem). But now the error vanishes at multiple nodes rather than just at x_0.

From the formula, you can see that the error will be larger for a more "wiggly" function, where the derivative $f^{(n+1)}$ is larger. It might also appear that the error will go down as the number of nodes n increases; we will see in Section 2.1.4 that this is not always true.

Note

As in Taylor's theorem, note the appearance of an undetermined point ξ. This will prevent us knowing the error exactly, but we can make an estimate as before.

Example 2.7: Quadratic interpolant for $f(x) = \cos(x)$ with $\{-\frac{\pi}{4}, 0, \frac{\pi}{4}\}$.

For $n = 2$, Cauchy's Interpolation Error Theorem says that

$$f(x) - p_2(x) = \frac{f^{(3)}(\xi)}{6}x(x+\tfrac{\pi}{4})(x-\tfrac{\pi}{4}) = \tfrac{1}{6}\sin(\xi)x(x+\tfrac{\pi}{4})(x-\tfrac{\pi}{4}),$$

for some $\xi \in [-\frac{\pi}{4}, \frac{\pi}{4}]$.
For an upper bound on the error at a particular x, we can just use $|\sin(\xi)| \leq 1$ and plug in x.
To bound the maximum error within the interval $[-1,1]$, let us maximise the polynomial $w(x) = x(x+\frac{\pi}{4})(x-\frac{\pi}{4})$. We have $w'(x) = 3x^2 - \frac{\pi^2}{16}$ so turning points are at $x = \pm\frac{\pi}{4\sqrt{3}}$.
We have

$$w(-\tfrac{\pi}{4\sqrt{3}}) = 0.186\ldots, \quad w(\tfrac{\pi}{4\sqrt{3}}) = -0.186\ldots, \quad w(-1) = -0.383\ldots, \quad w(1) = 0.383\ldots.$$

So our error estimate for $x \in [-1,1]$ is

$$|f(x) - p_2(x)| \leq \tfrac{1}{6}(0.383) = 0.0638\ldots$$

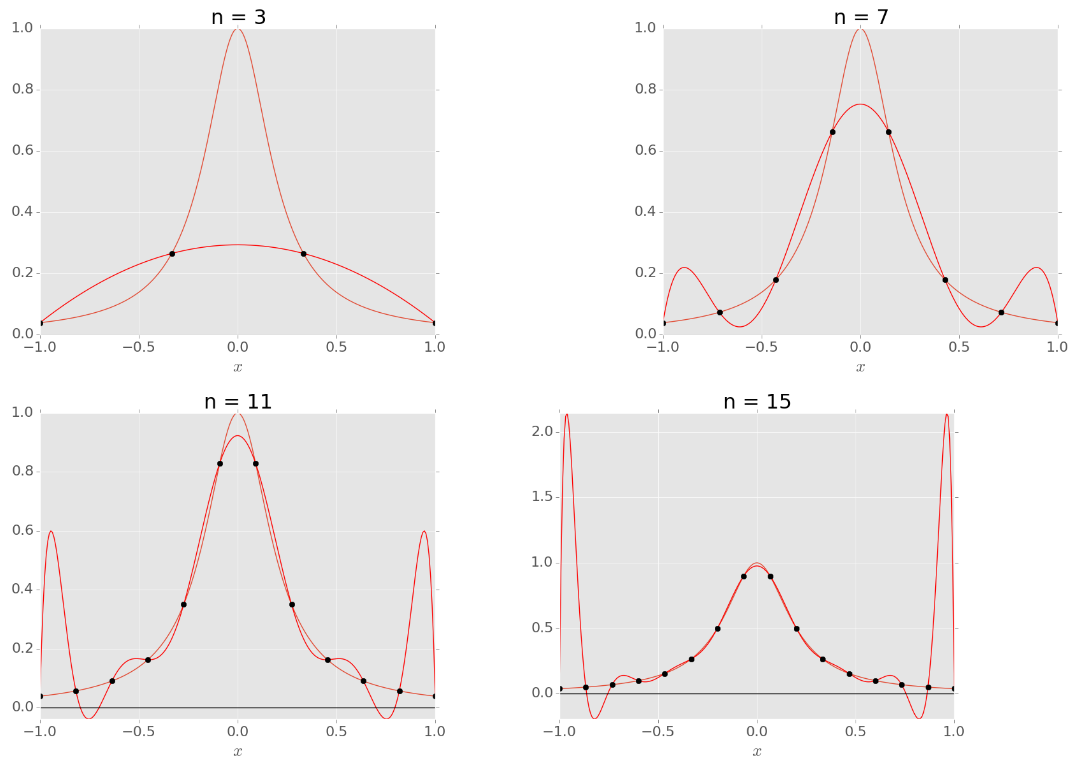
From the plot earlier, we see that this bound is satisfied (as it has to be), although not tight.

2.1.4 Node Placement: Chebyshev nodes

You might expect polynomial interpolation to *converge* as $n \to \infty$. Surprisingly, this is not the case if you take **equally-spaced** nodes x_i. This was shown by Runge in a famous 1901 paper.

Example 2.8: The Runge function $f(x) = 1/(1 + 25x^2)$ on $[-1, 1]$.

Here are illustrations of p_n for increasing n:



Notice that p_n is converging to f in the middle, but diverging more and more near the ends, even within the interval $[x_0, x_n]$. This is called the **Runge phenomenon**.

Note

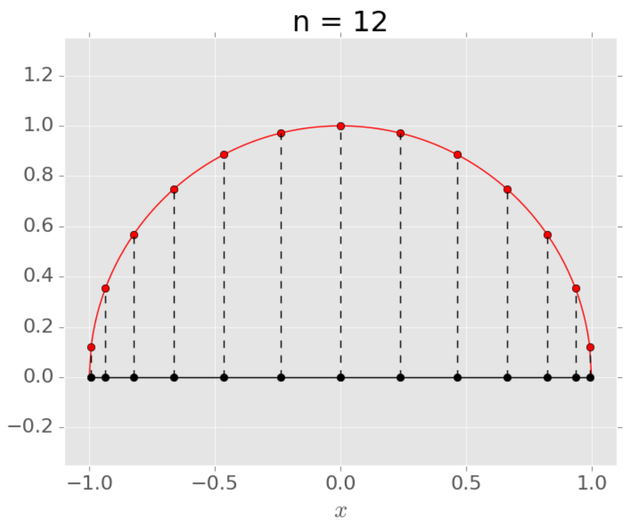
A full mathematical explanation for this divergence usually uses complex analysis — see Chapter 13 of *Approximation Theory and Approximation Practice* by L.N. Trefethen (SIAM, 2013). For a more elementary proof, see this StackExchange post.

The problem is (largely) coming from the interpolating polynomial

$$w(x) = \prod_{i=0}^{n} (x - x_i).$$

We can avoid the Runge phenomenon by choosing different nodes x_i that are **not uniformly spaced**.

Since the problems are occurring near the ends of the interval, it would be logical to put more nodes there. A good choice is given by taking equally-spaced points on the unit circle $|z| = 1$, and projecting to the real line:



The points around the circle are

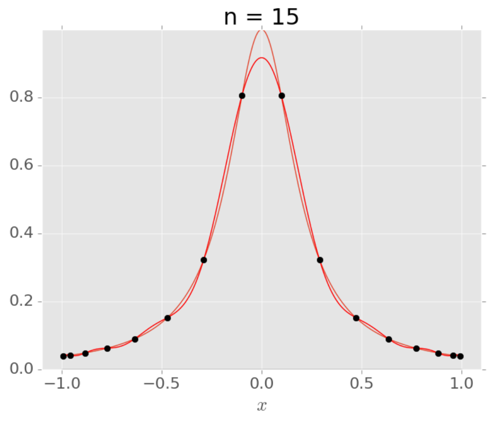
$$\phi_j = \frac{(2j+1)\pi}{2(n+1)}, \quad j = 0, \ldots, n,$$

so the corresponding **Chebyshev nodes** are

$$x_j = \cos\left[\frac{(2j+1)\pi}{2(n+1)}\right], \quad j = 0, \ldots, n.$$

Example 2.9: The Runge function $f(x) = 1/(1+25x^2)$ on $[-1, 1]$ using the Chebyshev nodes.

For $n = 3$, the nodes are $x_0 = \cos(\frac{\pi}{8})$, $x_1 = \cos(\frac{3\pi}{8})$, $x_2 = \cos(\frac{5\pi}{8})$, $x_3 = \cos(\frac{7\pi}{8})$.
Below we illustrate the resulting interpolant for $n = 15$:



Compare this to the example with equally spaced nodes.

In fact, the Chebyshev nodes are, in one sense, an optimal choice. To see this, we first note that they are zeroes of a particular polynomial.

The Chebyshev points $x_j = \cos\left[\frac{(2j+1)\pi}{2(n+1)}\right]$ for $j = 0, \ldots, n$ are zeroes of the Chebyshev polynomial

$$T_{n+1}(t) := \cos\left[(n+1)\arccos(t)\right]$$

Note

The Chebyshev polynomials are denoted T_n rather than C_n because the name is transliterated from Russian as "Tchebychef" in French, for example.

In choosing the Chebyshev nodes, we are choosing the error polynomial $w(x) := \prod_{i=0}^{n}(x - x_i)$ to be $T_{n+1}(x)/2^n$. (This normalisation makes the leading coefficient 1) This is a good choice because of the following result.

Theorem 2.7: Chebyshev interpolation

Let $x_0, x_1, \ldots, x_n \in [-1, 1]$ be distinct. Then $\max_{[-1,1]} |w(x)|$ is minimized if

$$w(x) = \frac{1}{2^n}T_{n+1}(x),$$

where $T_{n+1}(x)$ is the **Chebyshev polynomial** $T_{n+1}(x) = \cos\Big((n+1)\arccos(x)\Big)$.

Having established that the Chebyshev polynomial minimises the maximum error, we can see convergence as $n \to \infty$ from the fact that

$$|f(x) - p_n(x)| = \frac{|f^{(n+1)}(\xi)|}{(n+1)!}|w(x)| = \frac{|f^{(n+1)}(\xi)|}{2^n(n+1)!}|T_{n+1}(x)| \leq \frac{|f^{(n+1)}(\xi)|}{2^n(n+1)!}.$$

If the function is well-behaved enough that $|f^{(n+1)}(x)| < M$ for some constant whenever $x \in [-1, 1]$, then the error will tend to zero as $n \to \infty$.

2.2 Solving Nonlinear Equations