Math 4: Numerical Analysis

Nathan Warner

Northern Illinois University

Computer Science
Northern Illinois University
United States

Contents

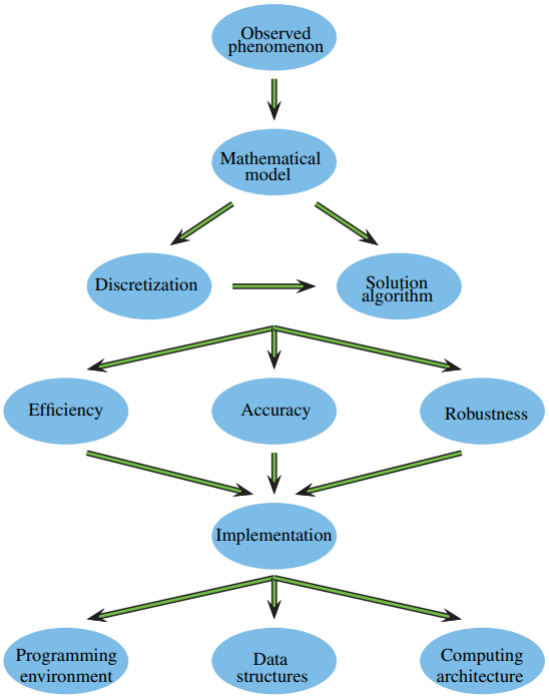
Numerical analysis with Julia

1.1 Numerical algorithms, roundoff errors, and nonlinear equations in one variable

- **Scientific computing**: Scientific computing is a discipline concerned with the development and study of **numerical algorithms** for solving mathematical problems that arise in various disciplines in science and engineering.

  Typically, the starting point is a given **mathematical model** which has been formulated in an attempt to explain and understand an observed phenomenon in biology, chemistry, physics, economics, or any other scientific or engineering discipline. We will concentrate on those mathematical models which are continuous (or piecewise continuous) and are difficult or impossible to solve analytically; this is usually the case in practice

  In order to solve such a model approximately on a computer, the continuous or piecewise continuous problem is approximated by a discrete one. Functions are approximated by finite arrays of values. Algorithms are then sought which approximately solve the mathematical problem efficiently, accurately, and reliably. This is the heart of scientific computing. **Numerical analysis** may be viewed as the theory behind such algorithms



- **Relative and absolute errors**: There are in general two basic types of measured error. Given a scalar quantity u and its approximation v:
  - **Absolute error**: The *absolute error* in v is

$$|u - v|$$

– **Relative error**: The *relative error*, assuming $u \neq 0$ is

$$\frac{|u - v|}{|u|}$$

Note: If we take the absolute error, $|u - v|$. Then it is clear it will be some percentage of u. In other words, some scaled version of u. Thus, we have $|u - v| = p\,|u|$.

The relative error is usually a more meaningful measure. This is especially true for errors in floating point representation. For example, we record absolute and relative errors for various hypothetical calculations in the following table

u	v	Absolute error	Relative error
1	0.99	0.01	0.01
1	1.01	0.01	0.01
-1.5	-1.2	0.3	0.2
100	99.99	0.01	0.0001
100	99	1	0.01

We expect the approximation in the last row of the above table to be similar in quality to the one in the first row. This expectation is borne out by the value of the relative error but is not reflected by the value of the absolute error

When the approximated value is small in magnitude, things are a little more delicate, and here is where relative errors may not be so meaningful. But let us not worry about this at this early point.

- **The sterling approximation**: The quantity

$$v = S_n = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Is called sterling's approximation and is used to approximate $u = n!$ for large n

- **Error types**: Knowing how errors are typically measured, we now move to discuss their source. There are several types of error that may limit the accuracy of a numerical calculation.

  1. **Errors in the problem to be solved**: These may be approximation errors in the mathematical model

     Another typical source of error in the problem is error in the input data. This may arise, for instance, from physical measurements, which are never infinitely accurate. Thus, it may be that after a careful numerical simulation of a given mathematical problem, the resulting solution would not quite match observations on the phenomenon being examined.

     At the level of numerical algorithms, which is the focus of our interest here, there is really nothing we can do about the above-described errors. Nevertheless, they should be taken into consideration, for instance, when determining the accuracy (tolerance with respect to the next two types of error mentioned below) to which the numerical problem should be solved.

  2. **Approximation errors**: Such errors arise when an approximate formula is used in place of the actual function to be evaluated.

We will often encounter two types of approximation errors:

(a) **Discretization errors** arise from discretizations of continuous processes, such as interpolation, differentiation, and integration.

(b) **Convergence errors** arise in iterative methods. For instance, nonlinear problems must generally be solved approximately by an iterative process. Such a process would converge to the exact solution in infinitely many iterations, but we cut it off after a finite (hopefully small!) number of such iterations. Iterative methods in fact often arise in linear algebra.

3. **Roundoff errorss:** Any computation with real numbers involves roundoff error. Even when no approximation error is produced (as in the direct evaluation of a straight line, or the solution by Gaussian elimination of a linear system of equations), roundoff errors are present. These arise because of the finite precision representation of real numbers on any computer, which affects both data representation and computer arithmetic.

- **Digits of accuracy**: If p is the relative error when v approximates u, then the digits of accuracy in the approximation v can be found with

$$\log_{10}\left(\frac{1}{p}\right) = -\log_{10}(p)$$

- **Catastrophic cancellation**: A numerical phenomenon that occurs when subtracting two nearly equal numbers in floating-point arithmetic. The result of this subtraction can lose significant digits, leading to a dramatic loss of precision in the computed result.

  In floating-point representation, numbers are stored with a finite number of significant digits (or bits). When two numbers are nearly equal, their leading digits cancel each other out during subtraction. The result is dominated by the remaining, less significant digits, which are more prone to rounding errors.

  Suppose we want to compute $x - y$, where $x = 1.0000001$, and $y = 1.0000000$. The true result is

$$x - y = 0.0000001$$

  However, if x and y are represented with only 7 significant digits in a floating-point system, $x = 1.000000$, and $y = 1.000000$. Then, their subtraction gives

$$x - y = 0.000000$$

  The true value is completely lost because the subtraction eliminates all significant digits.

- **Numerical noise**: Numerical noise refers to small errors or inaccuracies that arise in numerical computations due to the limitations of floating-point arithmetic. These errors are often very small, but they can accumulate or become significant in certain situations, especially when the computations involve many steps or operations sensitive to precision.

  Computers represent real numbers in a finite number of bits (e.g., 64 bits for double-precision floats).

This representation cannot store every real number exactly, so numbers are rounded to the nearest representable value. These small rounding errors introduce "noise" into computations.

- **Machine epsilon**: The machine epsilon is the smallest positive number ε such that

$$1 + \varepsilon > 1$$

in a given floating-point system. It represents the upper bound on the relative error due to rounding in floating-point arithmetic.

  - **Single Precision (32-bit):**

$$\varepsilon_{\text{machine}} \approx 2^{-23} \approx 1.19 \times 10^{-7}$$

  - **Double Precision (64-bit):**

$$\varepsilon_{\text{machine}} \approx 2^{-52} \approx 2.22 \times 10^{-16}$$

- **Approximation Error (approximating the derivative)**: Consider the formula for the derivative of a differentiable function $f\colon \mathbb{R} \to \mathbb{R}$ at x_0:

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}.$$

It is therefore reasonable to approximate $f'(x_0)$ using

$$\frac{f(x_0 + h) - f(x_0)}{h}$$

for some small positive h. The error in this approximation is

$$\left| f'(x_0) - \frac{f(x_0 + h) - f(x_0)}{h} \right|$$

and is called a **discretization error**.

For example, consider $f(x) = \sin x$ at $x_0 = 1$. Note that $f'(x) = \cos x$. We have

$$f'(x_0) = \cos 1 = 0.5403023058681398\ldots$$

Let's now approximate this with

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

For $h = 10^{-1}, 10^{-2}, ..., 10^{-16}$. The following Julia code

```julia
using Printf

function deriv_approx(f, x0, fp)
    @printf("%6s %24s %12s %10s %8s\n", "h", "fpapprox",
    ↪  "abserr", "relerr", "digits")

    for k in 1:16
        h = 10.0^(-k)
        fpapprox = (f(x0+h) - f(x0))/h
        abserr = abs(fp - fpapprox)
        relerr = abserr/abs(fp)
        digits = -log10(relerr)
        @printf("%6.0e %24.16e %12.4e %10.2e %8.1f\n", h,
        ↪  fpapprox, abserr, relerr, digits)
    end

    return nothing
end

deriv_approx(f, 1.0, cos(1.0))
```

Yields the following output

h	fpapprox	abserr	relerr	digits
1e-01	4.9736375253538911e-01	4.2939e-02	7.95e-02	1.1
1e-02	5.3608598101186888e-01	4.2163e-03	7.80e-03	2.1
1e-03	5.3988148036032690e-01	4.2083e-04	7.79e-04	3.1
1e-04	5.4026023141862112e-01	4.2074e-05	7.79e-05	4.1
1e-05	5.4029809850586474e-01	4.2074e-06	7.79e-06	5.1
1e-06	5.4030188512133037e-01	4.2075e-07	7.79e-07	6.1
1e-07	5.4030226404044868e-01	4.1828e-08	7.74e-08	7.1
1e-08	5.4030230289825454e-01	2.9699e-09	5.50e-09	8.3
1e-09	5.4030235840940577e-01	5.2541e-08	9.72e-08	7.0
1e-10	5.4030224738710331e-01	5.8481e-08	1.08e-07	7.0
1e-11	5.4030113716407868e-01	1.1687e-06	2.16e-06	5.7
1e-12	5.4034554608506369e-01	4.3240e-05	8.00e-05	4.1
1e-13	5.3956838996782608e-01	7.3392e-04	1.36e-03	2.9
1e-14	5.4400928206632670e-01	3.7070e-03	6.86e-03	2.2
1e-15	5.5511151231257827e-01	1.4809e-02	2.74e-02	1.6
1e-16	0.0000000000000000e+00	5.4030e-01	1.00e+00	-0.0

Notice that when h is decreased by a factor of ten, the absolute error decreases by a factor of ten.

Further, notice that when $h = 10^{-k}$, for $k \in \{9, 10, 11, ..., 16\}$, the absolute error gets worse instead of better. Also, when $h = 10^{-16}$, we notice that the approximation reads zero. This is a result of round-off errors and the limitations of floating-point arithmetic in computers.

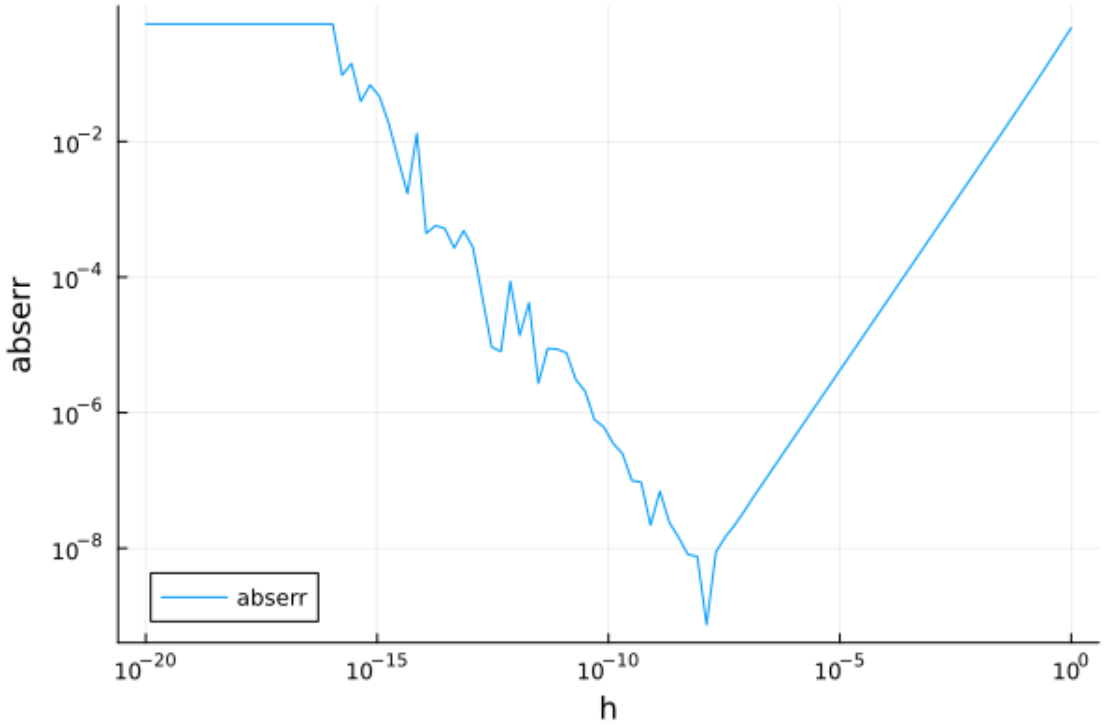
– **Catastrophic Cancellation:** The approximation formula involves subtracting two very close values $f(x_0 + h)$ and $f(x_0)$, for very small h

When h becomes very small, the values of $f(x_0 + h)$ and $f(x_0)$ are nearly identical. In floating-point arithmetic, this subtraction loses precision because the significant digits cancel out, leaving only the less accurate lower-order bits

– **Floating-Point Precision:** Floating-point numbers have limited precision. For typical 64-bit double-precision floating-point numbers, the relative precision is about 10^{-16} (The choice of h going up to 10^{-16} was no coincidence)

When h is smaller than 10^{-8}, the differences $f(x_0+h)-f(x_0)$ approach the limits of floating-point precision. Consequently, the computation becomes dominated by numerical noise, which introduces errors.

Observe the plot of h versus the absolute error in this approximation



- **Taylor series**: Assume that f is a function that is $(k+1)$-differentiable on an interval containing x_0 and $x_0 + h$. Then

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \cdots + \frac{h^k}{k!}f^{(k)}(x_0) + \frac{h^{k+1}}{(k+1)!}f^{(k+1)}(\xi),$$

for some $\xi \in (x_0, x_0 + h)$.

- **Proof that the discretization error decreases at the same rate as**: Solving for $f'(x_0)$ in the Taylor series expansion, we get

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - \left(\frac{h}{2}f''(x_0) + \frac{h^2}{6}f'''(x_0) + \cdots + \frac{h^{k-1}}{k!}f^{(k)}(\xi)\right).$$

Therefore,

$$\left| f'(x_0) - \frac{f(x_0+h) - f(x_0)}{h} \right| = \left| \frac{h}{2} f''(x_0) + \frac{h^2}{6} f'''(x_0) + \cdots + \frac{h^{k-1}}{k!} f^{(k)}(\xi) \right|.$$

If $f''(x_0) \neq 0$ and h is small, then the right-hand-side is dominated by $\frac{h}{2} f''(x_0)$. Thus,

$$\left| f'(x_0) - \frac{f(x_0+h) - f(x_0)}{h} \right| \approx \frac{h}{2} \left| f''(x_0) \right| = \mathcal{O}(h). \quad \blacksquare$$

Recall that $f(x) = \sin x$. Thus, $f''(x) = -\sin x$. Therefore,

$$\frac{|f''(x_0)|}{2} = \frac{-\sin 1}{2} = 0.42073549240394825\ldots$$

- **Roundoff error**: Numbers are stored in the computer using a finite precision representation. Roughly 16 digits of precision are possible using the 64-bit floating point format.

  Whenever an arithmetic operation takes place, the result must be rounded to roughly 16 digits of precision. Such an error is called roundoff error.

- **Accuracy**: As we have seen above, it is easy to write mathematically correct code that produces very inaccurate results.

  Accuracy is affected by the following two conditions:

  1. **Problem conditioning**: Some problems are highly sensitive to small changes in the input: we call such problems ill-conditioned. A problem that is not sensitive to small changes in the input is called well-conditioned. For example, computing $\tan(x)$ for x near $\frac{\pi}{2}$ is an ill-conditioned problem (Example 1.5 in Ascher-Greif).
  2. **Algorithm stability**: An algorithm is called stable if it is guaranteed to produce an exact answer to a slightly perturbed problem. (Example 1.6 in Ascher-Greif gives an example of an unstable algorithm).

     A "slightly perturbed problem" means a problem that has been altered by a small amount. For example, this could be small changes in the input data due to measurement errors or rounding errors.

     The algorithm is said to be stable if it provides the exact solution to this slightly perturbed problem. In other words, the output corresponds to what would happen if you solved the slightly modified problem exactly, rather than the original unmodified problem.

     A stable algorithm ensures that the effects of small input errors or numerical approximations (like rounding) do not grow uncontrollably during computations.

- **Unstable algorithm example**: Let

$$y_n = \int_0^1 \frac{x^n}{x+10} dx.$$

Then

$$y_n + 10y_{n-1} = \int_0^1 \frac{x^n + 10x^{n-1}}{x+10} dx = \int_0^1 x^{n-1} dx = \frac{1}{n} x^n \Big|_0^1 = \frac{1}{n}$$

and

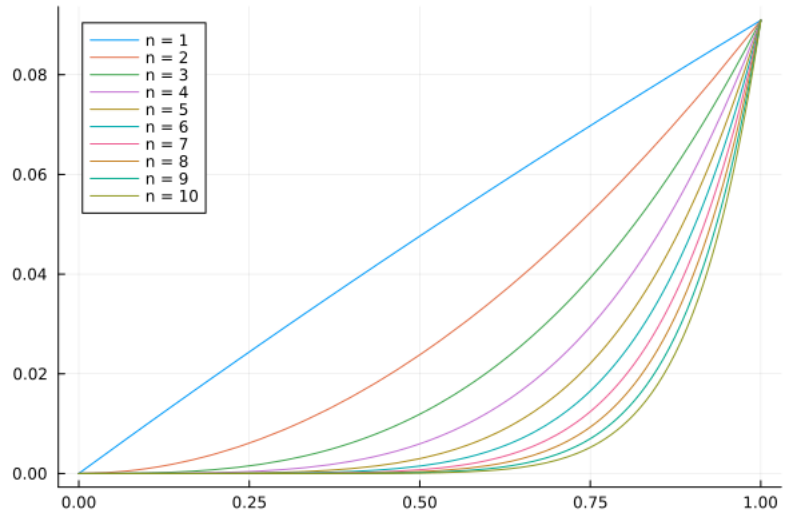
$$y_0 = \int_0^1 \frac{1}{x+10} dx = \ln|x+10| \Big|_0^1 = \ln(11) - \ln(10).$$

Then use these formulas to numerically compute y_{30}.

First, let's look at the functions $\frac{x^n}{x+10}$ as n grows. Using the Julia code

```
plot()
for n in 1:10
    plot!(x -> x^n/(x+10), 0, 1, label="n = $n")
end
plot!()
```

We get the graphs



So it appears the area under the curve is approaching zero. First, let's look at the integral result and error from a known stable algorithm (for $n = 30$)

```
using QuadGK

n = 30
integral, error = quadgk(x -> x^n/(x + 10), 0, 1)

#(0.002940928704861327, 8.45119305817703e-12)
```

The Julia code that uses the derived algorithm gives

```
        y0 = log(11) - log(10)

        yvals = zeros(30)
        yvals[1] = 1 - 10*y0
        for n = 2:30
            yvals[n] = 1/n - 10*yvals[n-1]
        end
        yvals

        # Out
        30-element Vector{Float64}:
       0.04689820195675232
       0.031017980432476833
       0.023153529008564988
       0.01846470991435012
       0.015352900856498819
       0.013137658101678468
       0.011480561840358172
       0.010194381596418278
       0.009167295146928323
       0.00832704853071678
       0.007638605601923115
       0.0069472773141021765
       0.007450303782055162

     916.9927348292546
   -9169.877348292546
    91698.82110197308
 -916988.1655651854
       9.169881699130116e6
      -9.169881694963449e7
       9.169881695363449e8
      -9.169881695324987e9
       9.169881695328691e10
      -9.169881695328334e11
       9.16988169532837e12
      -9.169881695328366e13
```

Thus, This algorithm is *very* **unstable**. The reason is the computation of y_0 using the log function. The log function introduces some roundoff error. Continuously using the results of the previous introduces more and more roundoff error.

- **Efficiency**: The efficiency of a code is affected by many factors:
  1. the rate of convergence of the method
  2. the number of arithmetic operations performed
  3. how the data in memory is accessed
- **Robustness (Reliability)**: We want to ensure that our code works under *all possible inputs*, and generates the clear warnings when it is not possible to produce an accurate result for some input.

1.2 Roundoff errors

- **Real numbers stored on a computer**: Real numbers are stored on a computer following the IEEE floating-point standard:

  1. **half precision:** using 16 bits (Julia type: 'Float16')
  2. **single precision:** using 32 bits (Julia type: 'Float32')
  3. **double precision:** using 64 bits (Julia type: 'Float64')

  Julia also has an *arbitrary precision* floating-point data type called 'BigFloat'. It is excellent if you need more precision, but it is also much slower.

  Julia has the type *AbstractFloat*, which is a subtype of the class Real, and is an abstract supertype for all floating point numbers.

```
AbstractFloat <: Real

> subtypes(AbstractFloat)
 5-element Vector{Any}:
 BigFloat
 Core.BFloat16
 Float16
 Float32
 Float64
```

- **Description of the IEEE Float64**: Suppose x is a floating-point number stored in the following 64-bits:

1	2	$\cdots$	12	13	$\cdots$	64
s	e_{10}	$\cdots$	e_0	f_1	$\cdots$	f_{52}

  Where

  – 1 bit s represents the **sign**

  – 11 bits $e_{10}\cdots e_0$ represent the **exponent**

  – 52 bits $f_1\cdots f_{52}$ represent the **fraction** (a.k.a. the mantissa or significand)

  Then

$$x = (-1)^s[1.f_1\cdots f_{52}]_2 \times 2^{(e-1023)}.$$

  Notes:

  – x is **normalized** to have its first digit nonzero.

  – $e = [e_{10}\cdots e_0]_2 = e_{10}2^{10} + \cdots + e_1 2^1 + e_0 2^0 \in \left[0, 2^{11}-1\right] = [0, 2047]$

  – $e = 0$ and $e = 2047$ are reserved for special floating-point values, so

$$e \in [1, 2046]$$

  The "-1023" in the exponent is called the **bias**: $e - 1023 \in [-1022, 1023]$

  Also,

$$[1.f_1\cdots f_{52}]_2 = 1 + \frac{f_1}{2^1} + \frac{f_2}{2^2} + \cdots + \frac{f_{52}}{2^{52}}$$

For example, suppose

$$\begin{aligned} x &= -[1.101101]_2 \times 2^{(1026-1023)} \\ &= -[1.101101]_2 \times 2^3 \\ &= -[1101.101]_2 \\ &= -\left(1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + 1 \cdot \frac{1}{8}\right) \\ &= -13.625 \end{aligned}$$

Even if a number can be represented exactly in base-10 with a finite number of digits, it may require an infinite number of digits in base-2.

$$0.1 = [0.000110011001\ldots]_2 = \left[1.\overline{1001}\right]_2 \times 2^{-4}$$

Therefore, 0.1 cannot be represented exactly as a floating-point number.

- **16-bit and 32-bit IEEE representation**
  - **16-bit (half-precision)**: The IEEE 754 half-precision floating-point format consists of:
    - 1 bit for the sign (s)
    - 5 bits for the exponent (e_4, e_3, e_2, e_1, e_0)
    - 10 bits for the fraction/mantissa ($f_1, f_2, \ldots, f_{10}$)

1	2	3	4	5	6	$\cdots$	16	
s	e_4	e_3	e_2	e_1	e_0	f_1	$\cdots$	f_{10}

  - **32-bit (Single precision)**: The IEEE 754 single-precision floating-point format consists of:
    - 1 bit for the sign (s)
    - 8 bits for the exponent ($e_7, e_6, e_5, e_4, e_3, e_2, e_1, e_0$)
    - 23 bits for the fraction/mantissa ($f_1, f_2, \ldots, f_{23}$)

1	2	3	4	5	6	7	8	9	10	$\cdots$	32
s	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	f_1	$\cdots$	f_{23}

- **Bias calculation**: The bias used above is calculated as

$$\text{Bias} = 2^{(E-1)} - 1$$

where E is the number of bits allocated to the exponent field.

  - **16-bit half precision**: Exponent is allowed 5 bits, thus

$$\text{Bias} = 2^{5-1} - 1 = 15$$

  - **32-bit single precision**: Exponent is allowed 8 bits, thus

$$\text{Bias} = 2^7 - 1 = 127$$

  - **64-bit double precision**: Exponent is allowed 11 bits, thus

$$\text{Bias} = 2^{10} - 1 = 1023$$

- **Convert real to binary representation**: So we now know how to convert a binary representation of a float to its decimal representation.

Consider the base ten real -13.625. To convert a base ten real into its binary representation, we first

Convert the integer part to binary: Following the standard algorithm to convert 13 to binary

$$\begin{aligned} 13 &= 2(6) + 1 : \ 1_2 \\ 6 &= 2(3) + 0 : \ 0_2 \\ 3 &= 2(1) + 1 : \ 1_2 \\ 1 &= 2(0) + 1 : \ 1_2 \end{aligned}$$

13_{10} is therefore 1101_2

Convert the fractional part: Convert the fractional part (0.625) by repeatedly multiplying by 2 and recording the whole number parts. Stop when the fractional part becomes 0. We build the resulting representation in the opposite way of the integer algorithm (top down)

$$\begin{aligned} 0.625 \cdot 2 &= 1.25 : \ 1_2 \\ 0.25 \cdot 2 &= 0.5 : \ 0_2 \\ 0.5 \cdot 2 &= 1 : \ 1_2 \end{aligned}$$

The result is therefore 101_2

Combine the integer and fractional parts: We have

$$13.625_{10} = 1101.101_2$$

Normalize the Binary Number: Normalize the binary number to the form

$$1.\text{mantissa} \cdot 2^{\text{exponent}}$$

For 1101.101_2, shift the decimal point left by three places to get 1.101101_2. The exponent is therefore three because

$$1101.101 = 1.101101 \cdot 2^3$$

Determine the Sign Bit: The sign bit is:

- 0 for positive numbers.
- 1 for negative numbers.

Since -13.625 is negative, the sign bit is 1

Encode the Exponent: The exponent is stored in "biased" form, for 64-bit double precision the bias is 1023. We add the bias to the actual exponent to get the biased to get the biases exponent

$$3 + 1023 = 1026$$

Then, convert the biased exponent to binary

$$1026_{10} = 10000000010_2$$

Encode the Mantissa: The mantissa is the fractional part of the normalized binary number (1.mantissa), excluding the leading 1.

From 1.101101_2, the mantissa is 101101. We pad with zeros to make it the required 52 bits

Combine the parts: The IEEE 754 representation is formed by combining

- **Sign bit (1 bit):** 1
- **Exponent (11 bits):** 10000000010
- **Mantissa (52 bits):** $1011010000000000000000...0_{52}$

- **Finding decimal value**: In a binary string of all ones, we have a mathematical formula to compute its value... Specifically

$$1 + 2 + 4 + 8 + ... + 2^n = 2^{n+1} - 1$$

For example (unsigned), $1111 = 2^{3+1} - 1 = 15$. But what about for binary decimals numbers, where we instead divide by two to find its value. Consider the binary string

$$0.11111_2 = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + ...$$

Notice this a geometric series

$$S = \sum_{k=1}^{n} \frac{1}{2^k}$$

The sum of the geometric series is given by

$$S = \frac{a(1 - r^n)}{1 - r}$$

Where a is the first term, r is the common ratio, and n is the number of terms. Therefore, for $a = \frac{1}{2}, r = \frac{1}{2}$, we have

$$\begin{aligned} S &= \frac{1}{2} \cdot \frac{1 - \left(\frac{1}{2}\right)^n}{1 - \left(\frac{1}{2}\right)} \\ &= 1 - \frac{1}{2^n} \end{aligned}$$

Thus, consider 0.1111, where $n = 4$. Note that n is not 3, like in the formula described for integers, this is because the sum starts at $n = 1$, instead of $n = 0$ like the integer formula. Thus,

$$0.1111 = 0 + 1 - \frac{1}{2^n} = 1 - \frac{1}{2^4} = 0.9375$$

And

$$1.1111 = 1 + 1 - \frac{1}{2^n} = 1 - \frac{1}{2^4} = 1 + 0.9375 = 1.9375$$

- **More on normalized float**: To be normalized in the context of IEEE 754 floating-point numbers means that the number is represented in a standardized form where:
  - The leading digit of the significand (mantissa) is always 1 (except for special cases like subnormal numbers).
  - This representation ensures there is only one unique way to represent each number.

This is akin to scientific notation, where we always write numbers like 3.25×10^2 instead of 32.5×10^1 or 0.325×10^3. In IEEE 754, normalized numbers always take the form:

$$1.\text{fraction} \times 2^{(\text{exponent}-\text{bias})}$$

- **Exponent of all ones**: All ones in the exponent is reserved for infinity and NaN. Thus, the largest exponent to work with in calculations is $11111111110 = 2046$
- **Limits of floating point numbers**: The largest Float64 is $(2 - 2^{-52}) \times 2^{1023} \approx 1.97769 \times 10^{308} \approx 2 \times 10^{308}$

  The largest float is when the sign bit is zero, all exponent bits are one, and all fraction bits are one. The exponent is then $2^{10+1} - 1 - 2047$, However, this value is reserved for infinity (and NaN). The largest finite exponent is 2046. The exponent bias is 1023, so the largest actual exponent $2046 - 1023 = 1023$. is The fractional part is $1.111...1_{52} = 1 + 1 - \frac{1}{2^{52}} = 2 - \frac{1}{2^{52}} = 2$. Thus, we get

  $$(-1)^0 \cdot 2 \cdot 2^{1023} = 2^{1024} \approx 2 \times 10^{308}$$

  Thus, the largest possible float64 is

  $$0\ 11111111110\ 111...1_{64}$$

  The smallest positive possible normalized float64 is $2^{-1022} \approx 2 \times 10^{-308}$, and it occurs when the sign bit is zero, the exponent is 00000000001 (all zeros reserved), and the fractional part is $1.000000...0_{52} = 1.0_{10}$. Thus, we get

  $$(-1)^0 \cdot 1.0 \cdot 2^{1-1023} = 2^{-1022} \approx 2.225 \times 10^{-308} \approx 2 \times 10^{-308}$$

  The smallest negative possible normalized float64 is then when the sign bit is one, we have

  $$(-1)^1 \cdot 1.0 \cdot 2^{-1022} \approx -2.225 \times 10^{-308} \approx -2 \times 10^{-308}$$

- **Finding these values in julia**: In julia, we have the functions

```
floatmax(T = Float64)
floatmin(T = Float64)
typemax(T)
```

- **Float overflow and underflow**: Floating-point overflow occurs when a calculation produces a number larger than the maximum representable value in the floating-point system. In IEEE 754 double precision (Float64), the largest finite number is $\approx 1.79769 \times 10^{308} \approx 2 \times 10^{308}$

  If an operation results in a number greater than this limit, IEEE 754 rules dictate that the result is represented as positive infinity

If the result is negative, it becomes negative infinity

- **De-normalized (subnormal) floating-point numbers**: The IEEE floating-point standard also allows de-normalized numbers that are smaller than $\pm 2^{-1022}$. De-normalized floats are represented by $e = 0$. Also note that subnormal floats have mantissa non-zero. Subnormal is therefore represented as

$$(-1)^s \cdot [0.f_1 f_2 ...]_2 \cdot 2^{-1022}$$

Note that the exponent is -1022 instead of -1023. This is due to the IEE convention for subnormal numbers to insure there is no gap between the largest subnormal number and the smallest normal number

The smallest positive subnormal float that is not zero is therefore

$$0\ 00000000000\ 000...01$$

And is equal to

$$(-1)^0 \frac{1}{2^{52}} \cdot 2^{-1022} \approx 4.94 \times 10^{-324} \approx 5 \times 10^{-324}$$

- **Other special floats**:
  - **0.0 and -0.0**:

$$e_{10}...e_0 = 0...0 \text{ and } f_1...f_{52} = 0...0$$

    If a very small negative number is rounded to zero, then it becomes -0.0. If a very small positive number rounds to zero, it becomes 0.0
  - **Inf and -Inf**:

$$e_{10}...e_0 = 1...1 \text{ and } f_1...f_{52} = 0...0$$

  - **Nan**:

$$e_{10}...e_0 = 1...1 \text{ and } f_1...f_{52} \neq 0$$

Note: $0.0, -0.0, \infty, -\infty, NaN$ are neither normal or subnormal

Also, from Julia

  - Finite numbers are ordered in the usual manner.
  - Positive zero is equal but not greater than negative zero.
  - Inf is equal to itself and greater than everything else except NaN.
  - -Inf is equal to itself and less then everything else except NaN.
  - NaN is not equal to, not less than, and not greater than anything, including itself.

- **Summary (Float64)**
  - **0.0 and -0.0**:

$$0\ 00000000000\ 000...01\ 00000000000\ 000...0$$

- **Smallest positive normal**

$$0\ 00000000001\ 000...00$$

And has value

$$(-1)^0[1.000...0]_2 \cdot 2^{1-1023} = 1.0 \cdot 2^{-1022} \approx 2.225 \cdot 10^{-308}$$

- **Largest positive normal**: Occurs when

$$0\ 11111111110\ 111...11$$

And has value

$$(-1)^0[1.111...11]_2 \cdot 2^{2^{10+1}-1-1-1023}$$
$$= 1 + 1 - \frac{1}{2^{52}} \cdot 2^{2046-1023} = 2 - \frac{1}{2^{52}} \cdot 2^{1023} \approx 1.797 \times 10^{308}$$

- **Smallest positive subnormal**

$$0\ 00000000000\ 000...01$$

And has value

$$(-1)^0 \cdot \frac{1}{2^{52}} \cdot 2^{-1022} \approx 4.94 \times 10^{-324}$$

- **Largest positive subnormal**

$$0\ 00000000000\ 111..11$$

$$(-1)^0 \cdot 1 - \frac{1}{2^{52}} \cdot 2^{-1022} \approx 2.225 \times 10^{-308}$$

- ∞ **and** $-\infty$

$$0\ 11111111111\ 000...00$$
$$1\ 11111111111\ 000...00$$

- **NaN**: Any sign bit, exponent all ones, any nonzero combination of fractional bits.

Therefore, we can also derive the largest and smallest negative normals and subnormals

- **Largest negative normal**

$$1\ 00000000001\ 000...00$$

And has value

$$(-1)^1[1.000...0]_2 \cdot 2^{1-1023} = (-1)1.0 \cdot 2^{-1022} \approx -2.225 \cdot 10^{-308}$$

- **Smallest negative normal**: Occurs when

$$1\ 11111111110\ 111...11$$

And has value

$$(-1)^1[1.111...11]_2 \cdot 2^{2^{10+1}-1-1-1023}$$
$$= (-1)1 + 1 - \frac{1}{2^{52}} \cdot 2^{2046-1023} = (-1)2 - \frac{1}{2^{52}} \cdot 2^{1023} \approx -1.797 \times 10^{308}$$

- **Largest negative subnormal**

  $$1\ 00000000000\ 000...01$$

  And has value

  $$(-1)^1 \cdot \frac{1}{2^{52}} \cdot 2^{-1022} \approx -4.94 \times 10^{-324}$$

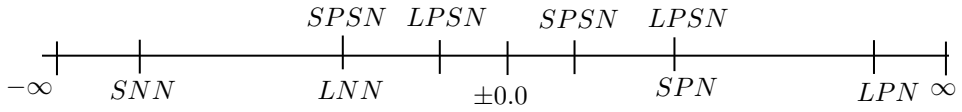
- **Smallest negative subnormal**

  $$1\ 00000000000\ 111..11$$

  $$(-1)^1 \cdot 1 - \frac{1}{2^{52}} \cdot 2^{-1022} \approx -2.225 \times 10^{-308}$$

  Notes: Float underflow takes you to -inf, float overflow takes you to +inf. A negative number rounded to zero will be -0.0, a positive number rounded to zero will be 0.0

  Behold



- **nextfloat and prevfloat in Julia**: In Julia, we can find the next float and the previous float with

```
nextfloat(f)
prevfloat(f)
```

- **Bounding of significand in a normalized number**: The significand is always in the range

  $$1 \leqslant 1.b_1 b_2 b_3 ... b_{t-1} < 2$$

  since the leading bit is always 1 in a normalized number.

  This means that x satisfies

  $$2^e \leqslant |x| < 2^{e+1}$$

  We sometimes take $|x| \approx 2^e$

- **Machine epsilon**: In the IEEE 754 floating-point standard, machine epsilon (denoted as ϵ_{mach}) is the smallest positive number that, when added to 1, results in a different representable floating-point number. It represents the upper bound on relative error due to rounding in floating-point arithmetic. That is

  $$1 + \epsilon \neq 1$$

For a floating-point system with t bits in the significand (mantissa) (including the implicit leading 1), the machine epsilon is:

$$e_{\text{mach}} = 2^{-(t-1)}$$

For the IEEE 754 double precision (64-bit) format, the mantissa has 52 fractional bits and one implicit leading bit. Thus, the machine epsilon is

$$2^{-(53-1)} = 2^{-52} \approx 2.2204 \times 10^{-16}$$

In Julia, we can find the machine epsilon with

```
eps(Float64)
```

Any rounding error in IEEE floating-point arithmetic is bounded above by ϵ_{mach}

- **Understanding machine epsilon**: In an IEEE 754 floating-point number with t bits in the significand (including the implicit 1), the machine epsilon is defined as

$$\epsilon_{\text{mach}} = 2^{-(t-1)} = \frac{1}{2^{t-1}}$$

A floating-point number is stored in normalized form

$$x = 1.b_1 b_2 b_3 ... b_{t-1} \times 2^e$$

The gap between two consecutive representable floating-point numbers is determined by the last bit of the significand

$$\text{Unit gap } = 2^{e-(t-1)}$$

This is because the smallest possible difference in the mantissa is $2^{-t(t-1)}$, which gets scaled by 2^e

When rounding to the nearest floating-point number, the maximum error occurs when x falls exactly between two consecutive representable numbers.

Since the gap between two consecutive numbers is

$$2^{e-(t-1)}$$

the maximum absolute rounding error is

$$\frac{1}{2} \times 2^{e-(t-1)}$$

because the number is rounded to the nearest representable value.

The relative error is given by

$$\frac{\text{max absolute rounding error}}{|x|}$$

Since $|x| \approx 2^e$ in normalized form

$$\frac{\frac{1}{2}2^{e-(t-1)}}{e} = \frac{1}{2} \times 2^{-(t-1)}$$

Since $\epsilon_{\text{mach}} = 2^{-(t-1)}$, we conclude

$$\text{Relative rounding error} \leqslant \frac{1}{2}\epsilon_{\text{mach}}$$

- **Unit roundoff** η: We define the unit roundoff $\eta = \frac{\epsilon}{2.0}$, and it is the largest possible relative error due to roundoff

$$\eta = 2^{-53} \approx 1.1 \times 10^{-16}$$

The unit roundoff represents the maximum rounding error in a floating-point system when using round-to-nearest mode

(the default in IEEE 754). This is because rounding introduces an error at most half the distance between two consecutive representable floating-point numbers.

- **Roundoff error example**: Suppose we are using a base-10 floating-point system with 4 significant digits, using 'RoundNearest':

$$\begin{aligned}\left(1.112 \times 10^1\right) \times \left(1.112 \times 10^2\right) &= 1.236544 \times 10^3 \\ &\to 1.237 \times 10^3 = 1237\end{aligned}$$

The absolute error is $1237 - 1236.544 = 0.456$.

The relative error is

$$\frac{0.456}{1236.544} \approx 0.0004 = 0.04\%$$

The default rounding mode is 'RoundNearest' (round to the nearest floating-point number). This implies that

$$\frac{|x - \text{fl}(x)|}{|x|} \leqslant \eta.$$

If 'RoundToZero' is used (a.k.a. **chopping**), then

$$\frac{|x - \text{fl}(x)|}{|x|} \leqslant 2\eta.$$

'RoundNearest' is used since it produces smaller roundoff errors.

- **Roundoff error accumulation**: When performing arithmetic operations on floats, extra **guard digits** are used to ensure **exact rounding**. This guarantees that the relative error of a floating-point operation (**flop**) is small. More precisely, for floating-point numbers x and y, we have

$$\begin{aligned}\text{fl}(x \pm y) &= (x \pm y)(1 + \varepsilon_1) \\ \text{fl}(x \times y) &= (x \times y)(1 + \varepsilon_2) \\ \text{fl}(x \div y) &= (x \div y)(1 + \varepsilon_3)\end{aligned}$$

where $|\varepsilon_i| \leqslant \eta$, for $i = 1, 2, 3$, where η is the unit roundoff.

Although the relative error of each flop is small, it is possible to have the roundoff error accumulate and create significant error in the final result. If E_n is the error after n flops, then:

  - **Linear roundoff error accumulation** is when $E_n \approx c_0 n E_0$
  - **Exponential roundoff error accumulation** is when $E_n \approx c_1^n E_0$, for some $c_1 > 1$

In general, linear roundoff error accumulation is unavoidable. On the other hand, exponential roundoff error accumulation is not acceptable and is an indication of an **unstable algorithm**. (See Example 1.6 in Ascher-Greif for an example of exponential roundoff error accumulation, and see Exercise 5 in Section 1.4 for a numerically stable method to accomplish the same task.)

- **General advice**:
  1. Adding $x + y$ when $|x| \gg |y|$ can cause the information in y to be "lost" in the summation.
  2. Dividing by very small numbers or multiplying by very large numbers can **magnify error**.
  3. Subtracting numbers that are almost equal produces **cancellation error**.
  4. An **overflow** occurs when the result is too large in magnitude to be representable as a float. The result will become either `Inf` or `-Inf`. Overflows should be avoided.
  5. An **underflow** occurs when the result is too small in magnitude to be representable as a float. The result will become either `0.0` or `-0.0`.
- **Information lose example**: This example shows that the summation order can make a difference. Consider the sum

$$s = \sum_{n=1}^{1,000,000} \frac{1}{n}$$

Let's do this sum in two different ways. First, from largest to smallest, then from smallest to largest.

Largest to smallest

```
sum = 0
for i in 1:1000000
    sum+=1/i
end
# 14.392726722864989
```

Smallest to largest

```
sum = 0
for i in 1000000:-1:1
    sum += 1/i
end
# 14.392726722865772
```

We can do the computation with a BigFloat to see which one is more precise

```
bfsum::BigFloat = 0
for i in 1:1000000
    sum+=BigFloat(1)/i
end
# 14.392726722865723631381127493188587676644800013744311653418433045812958507511 94
```

So we see the smallest to largest sum is slightly more accurate. Why is this? When summing a sequence of numbers, the order in which the numbers are added affects how rounding errors accumulate. Adding smaller numbers to a large sum can cause the smaller values to be "swallowed" due to the limitations of floating-point precision.

When summing from large terms to small terms, the large values dominate early in the computation. Because floating-point numbers have limited precision, adding much smaller numbers later may result in those numbers being effectively ignored (i.e., they contribute little due to rounding errors).

When summing from small terms to large terms, the intermediate sum stays small for longer, allowing more precise accumulation of the smaller values before reaching the larger ones. This reduces the impact of rounding errors.

- **Cancellation error example**: We consider

$$\ln\left(x-\sqrt{x^2-1}\right) = -\ln\left(x+\sqrt{x^2-1}\right)$$

First, we show that these expressions are actually equivalent

$$\begin{aligned} x-\sqrt{x^2-1} &= x-\sqrt{x^2-1}\left(\frac{x+\sqrt{x^2-1}}{x+\sqrt{x^2-1}}\right) \\ &= \frac{x^2-\sqrt{x^2-1}+\sqrt{x^2-1}-\left(\sqrt{x^2-1}\right)^2}{x+\sqrt{x^2-1}} \\ &= \frac{1}{x+\sqrt{x^2-1}} = \left(x+\sqrt{x^2-1}\right)^{-1} \end{aligned}$$

Thus,

$$\begin{aligned} \ln\left(x-\sqrt{x^2-1}\right) &= \ln\left(\left(x+\sqrt{x^2-1}\right)^{-1}\right) \\ &= -\ln\left(x+\sqrt{x^2-1}\right) \end{aligned}$$

Let's see which one is more accurate in numerical computations.

```
x = 1e6
fl = log(x-sqrt(x^2 -1))
fr = -log(x+sqrt(x^2 -1))
fl,fr

# (-14.50865012405984, -14.508657738523969)
```

If we examine the quantities x, and $\sqrt{x^2-1}$, we see that the are very close to each other

```
x, sqrt(x^2-1)
# (1.0e6, 999999.9999995)
```

The first expression $\ln\left(x-\sqrt{x^2-1}\right)$ gives cancellation error, which happens when two nearly equal floating-point numbers are subtracted, leading to significant loss of precision.

When two very close numbers are subtracted, the leading digits cancel out, leaving only a small result with much fewer significant digits. This makes the result inaccurate.

- **Example: Avoiding overflow**: Overflow is possible when squaring a large number. This needs to be avoided when computing the Euclidean norm (a.k.a. the 2-norm) of a vector x:

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

If some x_i is very large, it is possible that x_i^2 will overflow, causing the final result to be **Inf**. We can avoid this as follows.

Let

$$\bar{x} = \max_{i=1:n} |x_i|.$$

Then

$$\|x\|_2 = \bar{x}\sqrt{\left(\frac{x_1}{\bar{x}}\right)^2 + \left(\frac{x_2}{\bar{x}}\right)^2 + \cdots + \left(\frac{x_n}{\bar{x}}\right)^2}.$$

Since $|x_i/\bar{x}| \leqslant 1$ for all i, no overflow will occur. Underflow may occur, but this is harmless.

1.3 Non linear equations in one variable

- **Julia LaTeX strings**: In a Julia REPL or jupyter notebooks, we can include the LatexStrings package. This package allows us to create strings that contain LaTeX and format them. We do this by creating a string like

  L"string contents"

  For example

  L"\int f(x) dx"

```
using LatexStrings
```

- **Intro**: In many applications, one needs the solution to a **nonlinear equation** for which there is no closed formula.

  Suppose you do not have a cube-root function, but only the operations $+$, $-$, $\times$, $\div$

  Polynomials with degree at least five have no general algebraic solution

  Some nonlinear equations may not be solved analytically, for example

  $$10\cosh\left(\frac{x}{4}\right) = x \quad \text{and } 2\cosh\left(\frac{x}{4}\right) = x$$

  Recall the hyperbolic sine, cosine, and tangent functions are defined as

  $$\sinh(t) = \frac{e^t - e^{-t}}{2}$$
  $$\cosh(t) = \frac{e^t + e^{-t}}{2}$$
  $$\tanh(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}}$$

  Also, $\tanh(t) = \frac{\sinh(t)}{\cosh(t)}$, $\frac{d}{dt}\sinh(t) = \cosh(t)$, and $\frac{d}{dt}\cosh(t) = \sinh(t)$

- **Problem statement: roots**: Given $f \in C[a,b]$ (i.e., a *continuous* function $f\colon [a,b] \to \mathbb{R}$) and we want to find $x^* \in [a,b]$ such that

  $$f(x^*) = 0.$$

  The solution x^* is called a **root** or **zero** of the function f. There could be exactly one root, many roots, or no roots at all.

- **The Julia Roots package**: In Julia, the package *Roots* gives us functions like `find_zero` to find or approximate the roots of an equation

```
using Pkg
Pkg.add("Roots")
using Roots
```

Consider the functions

```
f(x) = 10cosh(x/4) - x
g(x) = 2cosh(x/4) - x
```

We can first plot the functions to get an tight interval that contains a root

```
plot(axes_style=:zerolines, xlims=[-2,12], xlabel=L"x",
    ylabel=L"y")
plot!(f, -2, 12, label=L"y = 10\cosh(x/4) - x")
plot!(g, -2, 12, label=L"y = 2\cosh(x/4) - x")
```

Using the function `find_zero`, passing in a function and an interval, we can find approximate or exact roots

```
x1 = find_zero(g, (2,3))
# 2.357551053877402

g(x1) # 0.0
```

- **Iterative methods**: Often there is no closed formula for a root x^* of the function f. Instead of using a formula to compute a root x^*, we will start with an **initial guess** x_0 and generate a **sequence of iterates**

$$x_1, x_2, x_3, \ldots, x_k, \ldots$$

that we hope **converges** to x^*; i.e.,

$$\lim_{k\to\infty} x_k = x^*$$

Note: Different initial guesses x_0 may generate sequences of iterates that converge to different roots. We will see how to deal with this issue.

- **Iterative methods: When to stop**: Since the sequence of iterates is infinite, we must decide when we are close enough to a root x^* . However, we do not know , so how can we decide when we are close enough?

Stop options are to stop when

1. The function value is small:
$$|f(x_k)| < \texttt{ftol}.$$
A problem with this test is that $|f(x_k)|$ may be very small although x_k is still very far from a root.
2. Consecutive iterates are very close to each other:
$$|x_k - x_{k-1}| < \texttt{atol}.$$
A problem with this test is that *atol* must take into account the magnitude of the iterates.

3. Consecutive iterates are **relatively** close to each other:

$$|x_k - x_{k-1}| < \texttt{rtol}\,|x_k|\,.$$

Usually this is more robust than the above absolute test.

Often a combination of the above conditions is used. For example, items 2 and 3 can be combined:

$$|x_k - x_{k-1}| < \texttt{tol}(1 + |x_k|).$$

- **Intermediate value theorem**: If $f \in C[a,b]$ and $f(a) \leqslant s \leqslant f(b)$, then there exists a real number $c \in [a,b]$ such that $f(c) = s$.
- **Bisection method**: Suppose $f \in C[a,b]$ and that $f(a)$ and $f(b)$ have opposite signs; i.e.,

$$f(a) \cdot f(b) < 0.$$

Recall the IVT from calculus

If $f \in C[a,b]$ and $f(a) \leqslant s \leqslant f(b)$, then there exists a real number $c \in [a,b]$ such that $f(c) = s$.

Since f changes sign over $[a,b]$, the Intermediate Value Theorem implies that there is some $x^* \in [a,b]$ such that $f(x^*) = 0$. The **bisection method** searches for a root of f in $[a,b]$ as follows.

1. Let $p = \frac{a+b}{2}$ be the **midpoint** of $[a,b]$.
2. If $f(a) \cdot f(p) < 0$, then there is a root in $[a,p]$.
3. If $f(a) \cdot f(p) = 0$, then p is a root.
4. If $f(a) \cdot f(p) > 0$, then there is a root in $[p,b]$.

Each time we apply the above, we get a subinterval that contains a root that is **half the size** of the interval $[a,b]$.

Consider the Julia code for the bisection method

```
function bisect(f, a, b; maxiters=1000, tol=1e-6)
    fa, fb = f(a), f(b)

    if fa * fb > 0
        error("f(a) and f(b) must have opposite signs")  # Ensure root exists
    end

    for i in 1:maxiters
        p = (a + b) / 2
        fp = f(p)

        if abs(fp) < tol  abs(b - a) < tol  # Stop if function value is small or interval is tiny
            return p
        elseif fa * fp < 0
            b, fb = p, fp
        else
            a, fa = p, fp
        end
    end

    return (a + b) / 2  # Return best approximation if maxiters is reached
end

f(x) = 2cosh(x/4) - x
a, b = 5.0, 10.0

p = bisect(f, a, b, tol=1e-6)
p, f(p)
```

The example

```
f(x) = 2cosh(x/4) - x
a, b = 5.0, 10.0

p = bisect(f, a, b, tol=0.0)
p, f(p)

# (8.507199570713027, 1.7763568394002505e-15)
```

Shows that we get a pretty good approximation

- **Analyzing the bisection method**: Initially, we know a root x^* is somewhere in the interval $[a, b]$. If we let x_k be the midpoint of the kth subinterval, then

$$|x^* - x_0| \leqslant \frac{b-a}{2}.$$

In the next iteration,

$$|x^* - x_1| \leqslant \frac{b-a}{4},$$

and in the following iteration,

$$|x^* - x_2| \leqslant \frac{b-a}{8},$$

and so on, each time reducing our error bound by a factor of 2. In general,

$$|x^* - x_k| \leqslant \frac{b-a}{2} \cdot 2^{-k}, \qquad \text{for } k = 0, 1, 2, \ldots.$$

Suppose we want to compute x_k such that

$$|x^* - x_k| \leqslant \texttt{atol}.$$

Then we just need to find the smallest positive integer k such that

$$\frac{b-a}{2} \cdot 2^{-k} \leqslant \texttt{atol}.$$

That is,

$$\frac{b-a}{2\texttt{atol}} \leqslant 2^k,$$

which gives us

$$\log_2\left(\frac{b-a}{2\texttt{atol}}\right) \leqslant k,$$

so we just need the first integer k that is larger than $\log_2\left(\frac{b-a}{2\texttt{atol}}\right)$. Therefore,

$$k = \left\lceil \log_2\left(\frac{b-a}{2\texttt{atol}}\right) \right\rceil.$$

- **Pros and cons of the bisection method**:
- **Pros:**

  1. **Simple:** The bisection method only requires function values, is easy to understand and implement, and it is easy to analyze.
  2. **Robust:** The bisection method is guaranteed to work, provided that f is continuous and changes sign on the interval $[a, b]$.

  Cons:

  1. **Slow to converge:** The bisection method often requires many function evaluations.
  2. **Does not generalize:** The bisection method only applies to solving equations involving one variable; it does not generalize to solving equations involving multiple variables.

- **Fixed point iteration**: Another simple approach to solving

  $$f(x) = 0$$

  is to re-write it as

  $$x = g(x)$$

  for some continuous function g. We call a point x a **fixed-point** of g if $x = g(x)$.

For example, If we let

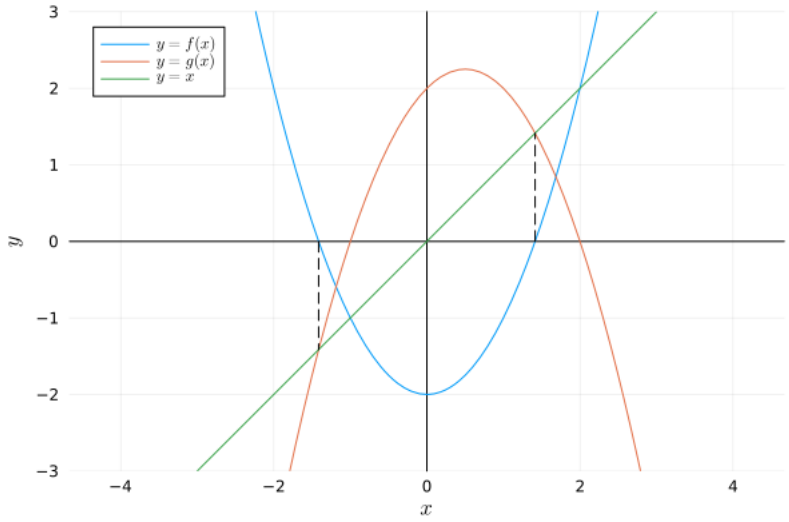
$$g(x) = x - f(x),$$

then

$$x = g(x) \quad \Rightarrow \quad x = x - f(x) \quad \Rightarrow \quad f(x) = 0.$$

Let's plot these functions using $f(x) = x^2 - 2$.

```
f(x) = x^2 - 2
g(x) = x-f(x)
a, b = -3.0, 3.0

plot(axes_style=:zerolines, aspect_ratio=:equal,
 ↪  legend=:topleft, ylims=[-3,3])
plot!(f, a, b, label=L"y = f(x)", c=1)
plot!(g, a, b, label=L"y = g(x)", c=2)
plot!(x -> x, a, b, label=L"y = x", c=3)
plot!([-sqrt(2), -sqrt(2)], [0, -sqrt(2)], linestyles=:dash,
 ↪  color=:black, label=:none)
plot!([sqrt(2), sqrt(2)], [0, sqrt(2)], linestyles=:dash,
 ↪  color=:black, label=:none)
xlabel!(L"x"); ylabel!(L"y")
```



We see that $f(x) = 0$ precisely when $g(x) = x$ (notice when the orange curve intersects the line $y = x$, it traces back up to the root of the blue curve)

- **Fixed point iteration: Choices of g**: There are many possible choices for g:
  - $g(x) = x - f(x)$
  - $g(x) = x + cf(x)$, for some nonzero constant c
  - $g(x) = x - f(x)/f'(x)$

  Some choices for g will be better than others.

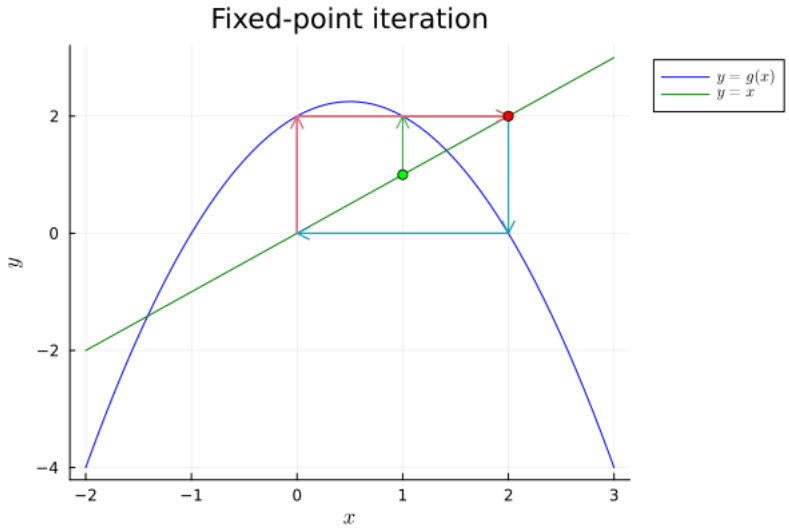
- **Iterations with fixed point iteration**: Given some initial guess x_0, we can use the function g to generate a sequence of iterates as follows:

$$x_{k+1} = g(x_k), \qquad k = 0, 1, 2, \ldots.$$

If the sequence $\{x_k\}$ converges to some point x^*, then we must have $x^* = g(x^*)$, so $f(x^*) = 0$.

Consider the Julia function that runs the above algorithm for $f(x) = x^2 - 2$, and $g(x) = x - f(x)$

```
function fixedPointPlot(g, a, b, x0; num=5, usequiver=true)

    plt = plot(g, a, b, label=L"y = g(x)", color=:blue)
    plot!(x -> x, a, b, label=L"y = x", color=:green)

    x = x0
    for i = 1:num
        if usequiver
            quiver!([x, x], [x, g(x)],
                quiver=([0, g(x)-x], [g(x)-x, 0]))
        else
            plot!([x, x], [x, g(x)], color=i, label=:none)
            plot!([x, g(x)], [g(x), g(x)], color=i,
             ↪  label=:none)
        end
        x = g(x)
    end

    scatter!([x0], [x0], label=:none, color=:lime)
    scatter!([x], [x], label=:none, color=:red)

    xlabel!(L"x")
    ylabel!(L"y")
    plot!(legend=:outertopright)
    title!("Fixed-point iteration")

    return plt
end

g1(x) = x - f(x)
fixedPointPlot(g1, -2, 3, x0, num=5)
```



We can examine k, and x_k

```
using Printf
x0 = 1.0; x = x0
@printf("%4s %12s\n", "k", "xk")
for k = 1:20
    x = g1(x)
    @printf("%4d %12.4e\n", k, x)
end
```

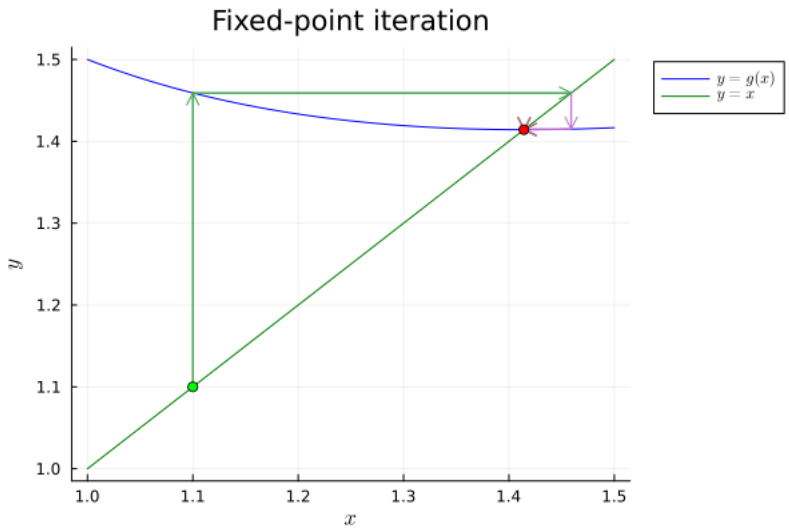
Which gives the table

k	x_k
1	2.0000e+00
2	0.0000e+00
3	2.0000e+00
4	0.0000e+00
5	2.0000e+00
6	0.0000e+00
7	2.0000e+00
8	0.0000e+00
9	2.0000e+00
10	0.0000e+00
11	2.0000e+00
12	0.0000e+00
13	2.0000e+00
14	0.0000e+00
15	2.0000e+00
16	0.0000e+00
17	2.0000e+00
18	0.0000e+00
19	2.0000e+00
20	0.0000e+00

It does not seem to be converging to anything. Now let's try $g(x) = x - f(x)/f'(x)$.

```
fixedPointPlot(x -> x-f(x)/2x, 1.0, 1.5, 1.1, usequiver=true)
```

Gives the plot



We can also examine the absolute errors

```
x = 1.0; xs = sqrt(2)

@printf("%4s %12s\n", "k", "error")
for k = 1:5
    x = (x->x-f(x)/2x)(x)
    @printf("%4d %12.4e\n", k, x - xs)
end
```

k	error
1	8.5786e-02
2	2.4531e-03
3	2.1239e-06
4	1.5947e-12
5	0.0000e+00

It converges very rapidly! We will see later why this is happening.

- **Mean value theorem**: If $f \in C[a,b]$ and f is differentiable on the open interval (a,b), then there exists a number $c \in (a,b)$ such that

$$f'(c) = \frac{f(b) - f(a)}{b - a}$$

Which means there exists some point c in which the tangent line at c is equal to the secant line drawn connecting points a and b

- **Existence and uniqueness of a fixed point**: A fixed point may not exist in $[a, b]$, and if it does, it may not be unique.

  Fixed point theorem: Let $g \in C[a, b]$ such that one of the two following conditions hold:

  1. $g(a) \geqslant a$ and $g(b) \leqslant b$;
  2. $g(a) \leqslant a$ and $g(b) \geqslant b$.

  Then $\exists x^* \in [a, b]$ such that $g(x^*) = x^*$. In addition, if g is differentiable on the open interval (a, b) and

  $$|g'(x)| \leqslant \rho, \quad \forall x \in (a, b),$$

  for some $\rho < 1$, then x^* is the *unique* fixed point in $[a, b]$.

  Proof. Suppose $g(a) \geqslant a$ and $g(b) \leqslant b$. If $g(a) = a$ or $g(b) = b$, then we are done. Otherwise we have $g(a) > a$ and $g(b) < b$. Let

  $$\phi(x) = g(x) - x.$$

  Then $\phi(a) > 0$ and $\phi(b) < 0$. Thus, since ϕ is continuous, the **Intermediate Value Theorem** tells us that there is an $x^* \in [a, b]$ such that $\phi(x^*) = 0$. Thus $x^* = g(x^*)$.

  The other case of $g(a) \leqslant a$ and $g(b) \geqslant b$ can be proven similarly.

  Now suppose g is differentiable and there is a $\rho < 1$ such that $|g'(x)| \leqslant \rho$ for all $x \in (a, b)$. Suppose, **for the sake of contradiciton**, that x^* is not the only fixed point of g in $[a, b]$. Then, there is a $y^* \in [a, b]$ such that $g(y^*) = y^*$ and $y^* \neq x^*$.

  By the **Mean Value Theorem**, there is a ξ strictly between x^* and y^* such that

  $$g'(\xi) = \frac{g(x^*) - g(y^*)}{x^* - y^*} = \frac{x^* - y^*}{x^* - y^*} = 1.$$

  Note that $\xi \in (a, b)$. This contradicts our assumption that $|g'(x)| \leqslant \rho < 1$, for all $x \in (a, b)$. Therefore, the fixed point of g in $[a, b]$ must be unique. ■

- **Convergence**: We have seen that the fixed point iteration does not always converge.

  Theorem: (Convergence of the Fixed Point Iteration): Let $g \in C[a, b]$. If

  - $a \leqslant g(x) \leqslant b$, for all $x \in [a, b]$, and
  - there is a $\rho < 1$ such that $|g'(x)| \leqslant \rho$ for all $x \in (a, b)$,

  then the iteration

  $$x_{k+1} = g(x_k), \qquad k = 0, 1, 2, \ldots$$

  converges to the unique fixed point $x^* \in [a, b]$ starting from any $x_0 \in [a, b]$.

  Proof. First of all, since $g(x) \in [a, b]$ for all $x \in [a, b]$, and since $x_0 \in [a, b]$, we have $x_k \in [a, b]$, for all $k = 0, 1, 2, \ldots$. Moreover, by the **Fixed Point Theorem**, our assumptions imply that there is a unique fixed point $x^* \in [a, b]$.

Let $k \in \{1, 2, \ldots\}$. If $x_{k-1} = x^*$, then we have already converged to the fixed point. Otherwise, suppose that $x_{k-1} \neq x^*$. By the **Taylor Series Theorem** (could also use the **Mean Value Theorem** like above), there exists a ξ strictly between x_{k-1} and x^* such that

$$g(x_{k-1}) = g(x^* + (x_{k-1} - x^*)) = g(x^*) + g'(\xi)(x_{k-1} - x^*).$$

Note that $\xi \in (a, b)$. Thus,

$$|x_k - x^*| = |g(x_{k-1}) - g(x^*)| = |g'(\xi)(x_{k-1} - x^*)| = |g'(\xi)|\,|x_{k-1} - x^*| \leqslant \rho\,|x_{k-1} - x^*|\,.$$

So $|x_k - x^*| \leqslant \rho\,|x_{k-1} - x^*|$ for $k = 1, 2, \ldots$, which implies that

$$0 \leq |x_k - x^*| \leqslant \rho\,|x_{k-1} - x^*| \leqslant \rho^2\,|x_{k-2} - x^*| \leqslant \cdots \leqslant \rho^k\,|x_0 - x^*|\,.$$

Since $\rho < 1$, the right-hand-side converges to 0 as $k \to \infty$. Therefore,

$$\lim_{k\to\infty} |x_k - x^*| = 0,$$

so x_k converges to x^*. ■

- **Contraction factor**: We call ρ the *contraction factor*, the smaller ρ is, the faster the convergence
- **Convergence example**: The first g we considered was

$$g(x) = x - x^2 + 2$$

which has the fixed points $x_1^* = -\sqrt{2}$ and $x_2^* = \sqrt{2}$. Note that

$$g'(x) = 1 - 2x$$

and that

$$g'(x_1^*) = 1 + 2\sqrt{2} = 3.8284271247461903\ldots,$$
$$g'(x_2^*) = 1 - 2\sqrt{2} = -1.8284271247461903\ldots.$$

So, $|g'(x_i^*)| > 1$ for $i = 1, 2$, which explains why the fixed point iteration would not converge to either fixed point.

The second g we considered was

$$g(x) = x - \frac{x^2 - 2}{2x} = \frac{x}{2} + \frac{1}{x},$$

which has the fixed points $x_1^* = -\sqrt{2}$ and $x_2^* = \sqrt{2}$. Now the derivative is

$$g'(x) = \frac{1}{2} - \frac{1}{x^2},$$

and so

$$g'(x_i^*) = 0, \quad i = 1, 2.$$

Thus, for $i = 1, 2$, we have $|g'(x_i^*)| < \rho$, for any $\rho \in (0, 1)$. This explains why the fixed point iteration converged rapidly to x_2^* from $x_0 = 1.1$; we also expect rapid convergence to x_1^* from suitable x_0.

- **Newton's method**: Let
$$f \in C^2[a, b].$$
That is, f is a **twice-continuously differentiable** function over $[a, b]$, which means that the **first** and **second** derivatives of f **exist** and are **continuous** on the open interval (a, b). **Newton's method** is defined as:
$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, 2, \ldots.$$
This is the fixed point iteration using the function $g(x) = x - f(x)/f'(x)$.

- **Formulating Newton's method**: Suppose that $f(x^*) = 0$ and that we are at the iterate x_k. By the **Taylor Series Theorem**, we have
$$f(x^*) = f(x_k) + f'(x_k)(x^* - x_k) + \frac{f''(\xi)}{2}(x^* - x_k)^2,$$
for some point ξ between x^* and x_k. If x_k is already fairly close to x^*, then $(x^* - x_k)^2$ will be very small, so we have
$$0 \approx f(x_k) + f'(x_k)(x^* - x_k).$$
Solving for x^*, we obtain
$$x^* \approx x_k - \frac{f(x_k)}{f'(x_k)}.$$
Therefore, it makes sense to define our next iterate x_{k+1} using this approximation.

- **Another formulation for Newton's method**: Another way to obtain Newton's method is as follows. Consider the **first-order (linear) approximation** of f around the point x_k:
$$f(x) \approx f(x_k) + f'(x_k)(x - x_k), \quad \text{for all } x \approx x_k.$$
Suppose that x_k is close to x^*, and that $f(x^*) = 0$. Then
$$f(x^*) \approx f(x_k) + f'(x_k)(x^* - x_k),$$
which implies that
$$x^* \approx x_k - \frac{f(x_k)}{f'(x_k)}.$$
Therefore, our next iterate should be
$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

- **Newton's method example: The Babylonian method for computing $\sqrt{a}$**: Let $f(x) = x^2 - a$. Newton's method gives us the iteration:
$$x_{k+1} = x_k - \frac{x_k^2 - a}{2x_k} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right).$$

- **Speed of convergence**: If $x_k \to x^*$, we can measure the speed of the convergence as follows.
  - **Linear convergence** means there is a constant $0 < \rho < 1$ such that
  $$|x_{k+1} - x^*| \leqslant \rho\,|x_k - x^*|\,, \quad \text{for all } k \text{ sufficiently large;}$$
  that is,
  $$\lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|} = \rho < 1.$$

– **Superlinear convergence** means there is a sequence $\rho_k \to 0$ such that

$$|x_{k+1} - x^*| \leqslant \rho_k |x_k - x^*|, \quad \text{for all } k \text{ sufficiently large;}$$

that is,

$$\lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|} = 0.$$

– **Quadratic convergence** means there is a constant M such that

$$|x_{k+1} - x^*| \leqslant M |x_k - x^*|^2, \quad \text{for all } k \text{ sufficiently large;}$$

that is,

$$\lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^2} = M < \infty.$$

Note that **quadratic convergence** is an example of **superlinear convergence** with $\rho_k = M |x_k - x^*|$.

- **Quadratic convergence of Newton's method**

Theorem: Let $f \in C^2[a, b]$. If f has a root $x^* \in (a, b)$ such that $f'(x^*) \neq 0$, then there is a $\delta > 0$ such that Newton's method **converges quadratically** to x^* from any $x_0 \in [x^* - \delta, x^* + \delta]$.

Proof. Since

– $f \in C^2[a, b]$
– $x^* \in (a, b)$
– $f'(x^*) \neq 0$

there are positive constants δ_1, ε, and M such that

– $|f'(x)| \geqslant \varepsilon$
– $|f''(x)| \leqslant M$

for all $x \in [x^* - \delta_1, x^* + \delta_1] \subset (a, b)$.

Suppose $x_k \in [x^* - \delta_1, x^* + \delta_1]$. Then, there is a ξ_k between x^* and x_k such that

$$f(x^*) = f(x_k) + f'(x_k)(x^* - x_k) + \frac{f''(\xi_k)}{2}(x^* - x_k)^2.$$

Using the fact that $f(x^*) = 0$, we have

$$0 = f(x_k) + f'(x_k)(x^* - x_k) + \frac{f''(\xi_k)}{2}(x^* - x_k)^2.$$

Also, x_{k+1} satisfies

$$0 = f(x_k) + f'(x_k)(x_{k+1} - x_k).$$

Subtracting these equations, we obtain

$$0 = f'(x_k)(x^* - x_{k+1}) + \frac{f''(\xi_k)}{2}(x^* - x_k)^2.$$

Since $f'(x_k) \neq 0$, we have

$$x^* - x_{k+1} = -\frac{f''(\xi_k)}{2f'(x_k)}(x^* - x_k)^2.$$

Thus,

$$|x^* - x_{k+1}| = \left|\frac{f''(\xi_k)}{2f'(x_k)}\right| |x^* - x_k|^2 \leqslant \frac{M}{2\varepsilon} |x^* - x_k|^2,$$

so if $x_k \to x^*$, then the **convergence will be quadratic**.

We just need to find $\delta > 0$ so that if $x_0 \in [x^* - \delta, x^* + \delta]$, then $x_k \to x^*$. Let

$$\delta = \min\left\{\frac{\varepsilon}{M}, \delta_1\right\}.$$

Suppose that $x_k \in [x^* - \delta, x^* + \delta]$. Then

$$\begin{aligned} |x^* - x_{k+1}| &\leqslant \frac{M}{2\varepsilon} |x^* - x_k|^2 \\ &\leqslant \frac{M}{2\varepsilon}\delta |x^* - x_k| \\ &\leqslant \frac{1}{2} |x^* - x_k| \\ &< \delta, \end{aligned}$$

so $x_{k+1} \in [x^* - \delta, x^* + \delta]$ as well. Thus, if $x_0 \in [x^* - \delta, x^* + \delta]$, we have $x_k \in [x^* - \delta, x^* + \delta]$ for $k = 0, 1, 2, \ldots$.

Moreover,

$$0 \leqslant |x^* - x_k| \leqslant \frac{1}{2} |x^* - x_{k-1}| \leqslant \frac{1}{4} |x^* - x_{k-2}| \leqslant \cdots \leqslant \frac{1}{2^k} |x^* - x_0|.$$

Since $\frac{1}{2^k} |x^* - x_0| \to 0$ as $k \to \infty$, we conclude that $x_k \to x^*$. Thus, if $x_0 \in [x^* - \delta, x^* + \delta]$ then x_k converges to x^* quadratically. ■

- **Pros and cons of Newton's method**:

  Pros:

  - **Fast to converge:** Newton's method enjoys quadratic convergence near the root when $f'(x^*) \neq 0$.
  - **Generalizes to multiple variables:** Let $\mathbf{F}\colon \mathbb{R}^n \to \mathbb{R}^n$. Newton's method for solving

    $$\mathbf{F}(\mathbf{x}) = \mathbf{0}$$

    (i.e., n nonlinear equations with n unknowns) is

    $$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}(\mathbf{x}_k)^{-1}\mathbf{F}(\mathbf{x}_k),$$

    where $\mathbf{J}(\mathbf{x})$ is the $n \times n$ **Jacobian** of $\mathbf{F}$:

    $$\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \cdots & \frac{\partial F_n}{\partial x_n} \end{bmatrix}$$

Cons

- **Requires the derivative:** We must give Newton's method both the function f and its derivative f'. This may not always be possible or easy.
- **Need to start close to x^*:** Newton's method is a **local method**. When x_0 is far from x^*, Newton's method may not converge to x^*, or may require many iterations before quadratic convergence begins.

- **Secant method**: Sometimes it is not possible to evaluate the derivative f':
  - f' is unknown or difficult to obtain
  - evaluating f' takes too much time

  Instead, we can use the **secant approximation** of the derivative. When $x_k \approx x_{k-1}$, we have
  $$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}.$$
  Plugging this approximation into the formula for Newton's method, we get:
  $$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$$

  The secant method is an example of a *Quasi-Newton method* method since we are replacing f' with an approximation of f'.

  When $f'(x^*) \neq 0$, the secant method will converge **superlinearly**, so it may not be as fast as Newton's method.

- **The case of a multiple root**: When $f'(x^*) = 0$, we are no longer guaranteed to obtain superlinear convergence of the secant method, nor quadratic convergence of Newton's method. In this case, both methods will be merely **linearly convergent**.

- **Minimizing a function in one variable**: We can use the root-finding methods described above to find the **minimum** or **maximum** value of a function $\phi \in C^2[a, b]$. Recall that $x^* \in (a, b)$ is a **critical point** of ϕ if
  $$\phi'(x^*) = 0.$$
  We can find x^* by applying Newton's method to this nonlinear equation to obtain:
  $$x_{k+1} = x_k - \frac{\phi'(x_k)}{\phi''(x_k)}.$$

- **Another interpretation**: We can also obtain this by considering the **second-order (quadratic) approximation** of ϕ around the point x_k:
  $$\phi(x) \approx \phi(x_k) + \phi'(x_k)(x - x_k) + \frac{\phi''(x_k)}{2}(x - x_k)^2, \quad \text{for all } x \approx x_k.$$
  If x_k is close to x^*, we expect the minimum/maximum of ϕ to be near the minimum/-maximum of the **quadratic approximation** of ϕ:
  $$q(x) = \phi(x_k) + \phi'(x_k)(x - x_k) + \frac{\phi''(x_k)}{2}(x - x_k)^2.$$
  We should choose x_{k+1} to be the critical point of q, so we want to find x_{k+1} such that $q'(x_{k+1}) = 0$. Note that
  $$q'(x) = \phi'(x_k) + \phi''(x_k)(x - x_k).$$
  Thus $q'(x_{k+1}) = 0$ gives us
  $$x_{k+1} = x_k - \frac{\phi'(x_k)}{\phi''(x_k)}.$$

1.4 Polynomial interpolation

- **Polynomials**: A function p is a **polynomial of degree at most** n if

$$p(x) = c_0 + c_1 x + \cdots + c_n x^n.$$

- **Weierstrass Approximation Theorem** : Polynomials can approximate any continuous function f as close as we want:

  Weierstrass approximation theorem: Let $f \in C[a,b]$. For every $\varepsilon > 0$, there exists a polynomial $p(x)$ such that

$$|f(x) - p(x)| < \varepsilon, \quad \forall x \in [a,b].$$

- **Horner's rule**: Polynomials can be efficiently evaluated using Horner's Rule:

$$p(x) = \Big(\Big(\big(c_n x + c_{n-1}\big)x + c_{n-2}\Big)x + \cdots + c_1\Big)x + c_0$$

  In Julia, we have the implementation

```
function horner(x, c)
    n = length(c) - 1
    p = c[n+1]
    for j = n:-1:1
        p = p*x + c[j]
    end

    return p
end
```

- **Two types of problems**: The function f we would like to approximate by a polynomial may be given to us as:

  1. **A fixed set of data points**: $\{(x_i, y_i)\}_{i=0}^n$, where $y_i = f(x_i)$, but the actual function f is unknown to us.
  2. **An explicit/implicit function**: We are free to choose the x_i and compute $y_i = f(x_i)$, but evaluating f may be expensive.

  In either case, the goal is to find a polynomial p that **interpolates** the data:

$$p(x_i) = y_i, \quad i = 0, 1, \ldots, n.$$

- **Estimating the function**: After constructing an interpolating polynomial p, we can use p to **estimate** the value of f at other values of x. We hope that

$$p(x) \approx f(x), \quad \forall x \in [a,b].$$

  We call the estimation of $f(x)$:

  1. **interpolation** if
$$x_i < x < x_j, \quad \text{for some } i \neq j,$$
  2. **extrapolation** if
$$x < x_i, \ \forall i \quad \text{or} \quad x > x_i, \ \forall i.$$

- **Interpolating polynomial always exists and is unique**:

Theorem: Let $\{(x_i, y_i)\}_{i=0}^n$. If $x_i \neq x_j$ for $i \neq j$, then there exists a unique polynomial $p(x)$ with degree at most n that satisfies

$$p(x_i) = y_i, \quad i = 0, 1, \ldots, n.$$

- **The space of polynomials**: Let $\mathbf{P}_n$ be the set of polynomials with degree at most n.

  $\mathbf{P}_n$ is a **vector space** since it is closed under addition and scalar multiplication:

  1. $p_1(x),\ p_2(x) \in \mathbf{P}_n \implies p_1(x) + p_2(x) \in \mathbf{P}_n$
  2. $c \in \mathbb{R},\ p(x) \in \mathbf{P}_n \implies cp(x) \in \mathbf{P}_n$

  Note that $\dim \mathbf{P}_n = n + 1$.

- **A basis for $\mathbf{P}_n$**: Let $\{\phi_j(x)\}_{j=0}^n$ be a **basis** for $\mathbf{P}_n$; that is:

  1. $\phi_0(x), \ldots, \phi_n(x)$ are **linearly independent**:

  $$c_0\phi_0(x) + \cdots + c_n\phi_n(x) = 0 \quad \implies \quad c_0 = \cdots = c_n = 0$$

  2. $\phi_0(x), \ldots, \phi_n(x)$ **spans $\mathbf{P}_n$**:

  $$\mathbf{P}_n = \text{Span}\{\phi_0(x), \ldots, \phi_n(x)\}$$

  Every $p(x) \in \mathbf{P}_n$ is therefore a unique linear combination of the polynomials in $\{\phi_j(x)\}_{j=0}^n$:

  $$p(x) = \sum_{j=0}^{n} c_j\phi_j(x) = c_0\phi_0(x) + \cdots + c_n\phi_n(x).$$

- **Computing the unique interpolating polynomial**: Given $\{(x_i, y_i)\}_{i=0}^n$, we want to find the unique $p(x) \in \mathbf{P}_n$ that satisfies

  $$p(x_i) = y_i, \quad i = 0, 1, \ldots, n.$$

  Thus, we just need to find scalars $c_0, \ldots, c_n$ such that

  $$\begin{aligned} p(x_0) &= c_0\phi_0(x_0) + \cdots + c_n\phi_n(x_0) = y_0 \\ p(x_1) &= c_0\phi_0(x_1) + \cdots + c_n\phi_n(x_1) = y_1 \\ &\vdots \\ p(x_n) &= c_0\phi_0(x_n) + \cdots + c_n\phi_n(x_n) = y_n \end{aligned}$$

  This is equivalent to the linear system $Ac = y$:

  $$\begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_n) & \phi_1(x_n) & \cdots & \phi_n(x_n) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

- **Using different bases for $\mathbf{P}_n$**: Any basis $\{\phi_j(x)\}_{j=0}^n$ we use will give us the same interpolating polynomial $p(x)$, but different bases will have different computational properties.

We will study **three** different bases:

1. **Monomial basis** $\{1, x, x^2, \ldots, x^n\}$, for which the matrix A is often ill-conditioned;
2. **Lagrange polynomial basis**, for which the matrix A is the identity matrix I;
3. **Newton polynomial basis**:, for which the matrix A is lower triangular.

In each case we will look at how to **construct** $p(x)$ and how to **evaluate** $p(x)$.

- **Monomial interpolation**: When using the **monomial basis** $\{1, x, x^2, \ldots, x^n\}$ to find the polynomial $p(x) \in \mathbf{P}_n$ that interpolates the data points $\{(x_i, y_i)\}_{i=0}^n$, the matrix A is given by

$$A = \begin{bmatrix} 1 & x_0 & \cdots & x_0^n \\ 1 & x_1 & \cdots & x_1^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^n \end{bmatrix}.$$

  This matrix is precisely the **Vandermonde matrix**

- **Determinant of the Vandermonde matrix**: We can compute the determinant of A as follows (using $n = 3$ for simplicity).

  First we do some row-reduction which does not change the determinant.

$$\begin{aligned} \det(A) &= \begin{vmatrix} 1 & x_0 & x_0^2 & x_0^3 \\ 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \end{vmatrix} \\ &= \begin{vmatrix} 1 & x_0 & x_0^2 & x_0^3 \\ 0 & x_1 - x_0 & x_1^2 - x_0^2 & x_1^3 - x_0^3 \\ 0 & x_2 - x_0 & x_2^2 - x_0^2 & x_2^3 - x_0^3 \\ 0 & x_3 - x_0 & x_3^2 - x_0^2 & x_3^3 - x_0^3 \end{vmatrix} \end{aligned}$$

  Then we do some column-reduction steps, which again do not change the determinant.

$$\begin{aligned} \det(A) &= \begin{vmatrix} 1 & x_0 & x_0^2 & 0 \\ 0 & x_1 - x_0 & x_1^2 - x_0^2 & x_1^3 - x_0^3 - x_0(x_1^2 - x_0^2) \\ 0 & x_2 - x_0 & x_2^2 - x_0^2 & x_2^3 - x_0^3 - x_0(x_2^2 - x_0^2) \\ 0 & x_3 - x_0 & x_3^2 - x_0^2 & x_3^3 - x_0^3 - x_0(x_3^2 - x_0^2) \end{vmatrix} \\ &= \begin{vmatrix} 1 & x_0 & x_0^2 & 0 \\ 0 & x_1 - x_0 & x_1^2 - x_0^2 & (x_1 - x_0)x_1^2 \\ 0 & x_2 - x_0 & x_2^2 - x_0^2 & (x_2 - x_0)x_2^2 \\ 0 & x_3 - x_0 & x_3^2 - x_0^2 & (x_3 - x_0)x_3^2 \end{vmatrix} \\ &= \begin{vmatrix} 1 & x_0 & 0 & 0 \\ 0 & x_1 - x_0 & (x_1 - x_0)x_1 & (x_1 - x_0)x_1^2 \\ 0 & x_2 - x_0 & (x_2 - x_0)x_2 & (x_2 - x_0)x_2^2 \\ 0 & x_3 - x_0 & (x_3 - x_0)x_3 & (x_3 - x_0)x_3^2 \end{vmatrix} \\ &= \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & x_1 - x_0 & (x_1 - x_0)x_1 & (x_1 - x_0)x_1^2 \\ 0 & x_2 - x_0 & (x_2 - x_0)x_2 & (x_2 - x_0)x_2^2 \\ 0 & x_3 - x_0 & (x_3 - x_0)x_3 & (x_3 - x_0)x_3^2 \end{vmatrix} \end{aligned}$$

Next, we pull out the common factors in each of the last three rows.

$$\det(A) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & x_1 & x_1^2 \\ 0 & 1 & x_2 & x_2^2 \\ 0 & 1 & x_3 & x_3^2 \end{vmatrix} (x_1 - x_0)(x_2 - x_0)(x_3 - x_0)$$

We repeat the above process on the bottom-right 3×3 submatrix.

$$\det(A) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & x_2 \\ 0 & 0 & 1 & x_3 \end{vmatrix} (x_2 - x_1)(x_3 - x_1) \cdot (x_1 - x_0)(x_2 - x_0)(x_3 - x_0)$$

Repeating the above process, this time on the bottom-right 2×2 submatrix, we obtain.

$$\det(A) = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} (x_3 - x_2) \cdot (x_2 - x_1)(x_3 - x_1) \cdot (x_1 - x_0)(x_2 - x_0)(x_3 - x_0)$$

Therefore,

$$\det(A) = (x_3 - x_2) \cdot (x_2 - x_1)(x_3 - x_1) \cdot (x_1 - x_0)(x_2 - x_0)(x_3 - x_0).$$

In general,

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

Thus, if $x_i \neq x_j$ for $i \neq j$, then $\det(A) \neq 0$, so A is invertible. Therefore, $Ac = y$ has exactly one solution which implies that there always exists a unique interpolating polynomial when the x_i are distinct.

- **An implementation of monomial interpolation**:

```
function vandermonde(x)
    n = length(x) - 1
    return [x[i+1]^j for i=0:n, j=0:n]
    # Equivalent to
    A = Array{Float64}(undef, n+1,n+1)
    # Or A = zeros(n+1,n+1)
    for i in 0:n
        for j in 0:n
            A[i+1,j+1] = x[i+1]^j
        end
    end
    return A
end
x = [2, 3, 4, -1.0]
A = vandermonde(x)
```

Yields

$$\begin{matrix} 1.0 & 2.0 & 4.0 & 8.0 \\ 1.0 & 3.0 & 9.0 & 27.0 \\ 1.0 & 4.0 & 16.0 & 64.0 \\ 1.0 & -1.0 & 1.0 & -1.0 \end{matrix}$$

```
function monointerp(x, y)
    A = vandermonde(x)
    # Solve A*c = y for c
    c = A\y # Returns c as it is the last evaluated
    ↪ expression
end
```

- **Monomial interpolation example**: Find the polynomial

$$p_1(x) = c_0 + c_1 x$$

that interpolates the following data points.

$$(x_0, y_0) = (2, 1)$$
$$(x_1, y_1) = (6, 2)$$

What are the values of $p_1(3)$, $p_1(5)$, and $p_1(7)$?

```
x = [2, 6.]
y = [1, 2.]
c1 = monointerp(x, y)
```

The dot syntax that you see above (6. and 2.) gives us float64 arrays, we can also get rational arrays (symbolic) by using the following syntax

```
x = [2, 6//1]
y = [1, 2//1]
c1 = monointerp(x, y)
# Yields 1//2 and 1//4
```

Thus, the polynomial is

$$p_1(x)\frac{1}{2} + \frac{1}{4}x$$

- **Horner's rule for a vector of x values**

```
function horner(x::AbstractVector, c)
    n = length(c) - 1
    p = c[n+1]*ones(length(x))
    for j = n:-1:1
        p .= p.*x .+ c[j]
    end
    p # Return p
end
horner(x::Real, c) = horner([x], c)[1]
```

- **Pros and cons of monomial interpolation**: **Pros:**

  1. **Simple:** just need to form A and solve $Ac = y$
  2. **Evaluating $p(x)$ is fast:** about $2n$ flops to compute $p(x)$ using Horner's Rule

  Cons:

  1. **Constructing $p(x)$ is expensive:** about $\frac{2}{3}n^3$ flops to solve $Ac = y$ using Gaussian elimination
  2. **Adding a new data point is expensive:** must form A and solve $Ac = y$ again
  3. **Solving $Ac = y$ is often inaccurate:** the Vandermonde matrix is often ill-conditioned.

- **Lagrange interpolation**: The motivation for **Lagrange interpolation** is to try to find a basis $\{\phi_j(x)\}_{j=0}^n$ for which the matrix

$$A = \begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_n) & \phi_1(x_n) & \cdots & \phi_n(x_n) \end{bmatrix}$$

  is equal to the identity matrix I. Then solving $Ac = y$ would be trivial: we would just set $c = y$.

  Thus, we would like to find polynomials that satisfy

$$\phi_i(x_i) = 1 \quad \text{for all } i, \quad \text{and} \quad \phi_j(x_i) = 0 \quad \text{for } i \neq j.$$

  Such polynomials are called **Lagrange polynomials**, and we denote them as:

$$L_0(x), L_1(x), \ldots, L_n(x).$$

  Example: Recall the data from above:

$$\begin{aligned} (x_0, y_0) &= (2, 1) \\ (x_1, y_1) &= (6, 2) \\ (x_2, y_2) &= (4, 3) \\ (x_3, y_3) &= (8, 2) \end{aligned}$$

  Find the Lagrange polynomial $L_0(x) \in \mathbf{P}_3$ such that $L_0(x_0) = 1$ and $L_0(x_1) = L_0(x_2) = L_0(x_3) = 0$.

Since $L_0(6) = L_0(4) = L_0(8) = 0$, and since $L_0(x) \in \mathbf{P}_3$, we must have

$$L_0(x) = a(x-6)(x-4)(x-8)$$

for some constant a.

Plugging in $x = 2$, we have that

$$1 = a(2-6)(2-4)(2-8),$$

so $a = -\frac{1}{48}$. Therefore,

$$L_0(x) = -\frac{1}{48}(x-6)(x-4)(x-8).$$

- **Constructing the Lagrange interpolating polynomial**: From the above exercises, we see that

$$L_j(x) = \frac{(x-x_0)\cdots(x-x_{j-1})(x-x_{j+1})\cdots(x-x_n)}{(x_j-x_0)\cdots(x_j-x_{j-1})(x_j-x_{j+1})\cdots(x_j-x_n)} = \prod_{\substack{i=0 \\ i\neq j}}^{n} \frac{(x-x_i)}{(x_j-x_i)}.$$

Thus, to construct each L_j, we just need to compute the **barycentric weights**:

$$w_j = \frac{1}{\rho_j}, \quad \text{where} \quad \rho_j = \prod_{\substack{i=0 \\ i\neq j}}^{n} (x_j - x_i).$$

Then

$$L_j(x) = w_j \prod_{\substack{i=0 \\ i\neq j}}^{n} (x - x_i).$$

Computing these weights requires about n^2 flops.

- **Evaluating the Lagrange interpolating polynomial**: Recall that the Lagrange interpolating polynomial is given by

$$p(x) = \sum_{j=0}^{n} y_j L_j(x).$$

Notice that

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

for $x \neq x_j$. Thus,

$$p(x) = \psi(x) \sum_{j=0}^{n} \frac{y_j w_j}{(x-x_j)}, \quad \text{for} \quad x \notin \{x_0, \ldots, x_n\}.$$

When $y_j = 1$, for all j, we have that $p(x) = 1$, for all x. Therefore,

$$1 = \psi(x) \sum_{j=0}^{n} \frac{w_j}{(x-x_j)},$$

which implies that

$$\psi(x) = \frac{1}{\sum_{j=0}^{n} \frac{w_j}{(x-x_j)}}.$$

Thus, we obtain the **barycentric formula** for $p(x)$:

$$p(x) = \frac{\sum_{j=0}^{n} \frac{y_j w_j}{(x-x_j)}}{\sum_{j=0}^{n} \frac{w_j}{(x-x_j)}}, \quad \text{for} \quad x \notin \{x_0, \ldots, x_n\}.$$

Evaluating $p(x)$ requires about $5n$ flops.

- **An implementation for evaluating the Lagrange interpolating polynomial**: The following code includes a strategy described in the 2004 paper *Barycentric Lagrange Interpolation* by Berrut and Trefethen for handling the case when $x \in \{x_0, \ldots, x_n\}$. Without using this strategy, the evaluation would return 'NaN' when $x = x_k$.

  In this paper they also discuss the numerical cancellation that occurs when $x \approx x_j$, making the calculation of $w_j/(x - x_j)$ inaccurate

```
function lagrangeeval(xspan::AbstractVector, w::Vector, x::AbstractVector, y::Vector)

    n = length(x)

    top = zero(xspan)
    bottom = zero(xspan)
    exact = zeros(Int, length(xspan))

    for j=1:n
        xdiff = xspan .- x[j]
        temp = w[j]./xdiff
        top += temp*y[j]
        bottom += temp

        exact[xdiff .== 0.0] .= j  # exact[i] = j if xspan[i] = x[j]
    end

    p = top./bottom

    iinds = findall(exact .!= 0)  # gives the indices i of xspan that equal some x[j]
    jinds = exact[iinds]          # gives the corresponding j indices
    p[iinds] = y[jinds]           # sets any NaNs in p to the correct values from y

    return p
end

lagrangeeval(xx::Real, w::Vector, x::AbstractVector, y::Vector) = lagrangeeval([xx], w, x, y)[1]
```

- **Pros and cons of Lagrange interpolation**: **Pros:**

1. **Constructing $p(x)$ is fast:** roughly n^2 flops to compute the barycentric weights
2. **Evaluating $p(x)$ is fast**: about $5n$ flops to compute $p(x)$ compared to $2n$ flops using Horner's Rule
3. **Adding a new interpolation point is fast**: barycentric weights can be updated in $\mathcal{O}(n)$ flops
4. **Can easily change the function**: the barycentric weights only depend on x_i, and not on the function f

Cons:

1. **Cannot also interpolate derivative values**:

- **Newton polynomial basis**: The Lagrange polynomial basis can be thought of as

$$\phi_j(x) = \prod_{\substack{i=0 \\ i \neq j}}^{n} (x - x_i), \quad j = 0, 1, \ldots, n.$$

The **Newton polynomial** basis is defined in a very similar way:

$$\phi_j(x) = \prod_{i=0}^{j-1} (x - x_i), \quad j = 0, 1, \ldots, n.$$

Using the Newton basis, the matrix

$$A = \begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_n) & \phi_1(x_n) & \cdots & \phi_n(x_n) \end{bmatrix}$$

is **lower-triangular**. This means that we can solve $Ac = y$ using **forward-substitution** in $\mathcal{O}(n^2)$ flops.

For example, The Newton polynomial basis for the data

$$\begin{aligned} (x_0, y_0) &= (2, 1) \\ (x_1, y_1) &= (6, 2) \\ (x_2, y_2) &= (4, 3) \\ (x_3, y_3) &= (8, 2) \end{aligned}$$

is the following set of polynomials

$$\begin{aligned} \phi_0(x) &= 1 \\ \phi_1(x) &= (x - 2) \\ \phi_2(x) &= (x - 2)(x - 6) \\ \phi_3(x) &= (x - 2)(x - 6)(x - 4) \end{aligned}$$

Using this basis, we find the interpolating polynomial

$$p(x) = c_0\phi_0(x) + c_1\phi_1(x) + c_2\phi_2(x) + c_3\phi_3(x)$$

by solving the linear system $Ac = y$, where

$$A = \begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \phi_2(x_0) & \phi_3(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \phi_2(x_1) & \phi_3(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \phi_2(x_2) & \phi_3(x_2) \\ \phi_0(x_3) & \phi_1(x_3) & \phi_2(x_3) & \phi_3(x_3) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 \\ 1 & 2 & -4 & 0 \\ 1 & 6 & 12 & 48 \end{bmatrix}..$$

Thus, $Ac = y$ is as follows.

$$\begin{array}{lllllllll} c_0 & & & & & & & = & 1 \\ c_0 & + & 4c_1 & & & & & = & 2 \\ c_0 & + & 2c_1 & - & 4c_2 & & & = & 3 \\ c_0 & + & 6c_1 & + & 12c_2 & + & 48c_3 & = & 2 \end{array}$$

We solve this system by forward-substitution:

$$\begin{aligned} c_0 &= 1 \\ c_1 &= \frac{1}{4}(2 - c_0) = \frac{1}{4} \\ c_2 &= -\frac{1}{4}(3 - c_0 - 2c_1) = -\frac{3}{8} \\ c_3 &= \frac{1}{48}(2 - c_0 - 6c_1 - 12c_2) = \frac{1}{12} \end{aligned}$$

Thus, the interpolating polynomial is

$$p(x) = 1 + \frac{1}{4}(x-2) - \frac{3}{8}(x-2)(x-6) + \frac{1}{12}(x-2)(x-6)(x-4)$$

- **Divided differences**: Given the data points $\{(x_i, y_i)\}_{i=0}^n$, where $y_i = f(x_i)$, the Newton form of the interpolating polynomial can also be written in closed-form as:

$$p(x) = \sum_{j=0}^{n} c_j \phi_j(x) = \sum_{j=0}^{n} \left(f[x_0, \ldots, x_j] \prod_{i=0}^{j-1} (x - x_i) \right).$$

That is, the coefficient c_j of $\phi_j(x)$ is the so-called j**th divided difference**:

$$c_j = f[x_0, \ldots, x_j].$$

Divided differences are defined recursively by

$$f[x_i] = f(x_i), \qquad \text{for } 1 \leqslant i \leqslant n,$$

and

$$f[x_i, \ldots, x_j] = \frac{f[x_{i+1}, \ldots, x_j] - f[x_i, \ldots, x_{j-1}]}{x_j - x_i}, \qquad \text{for } 1 \leqslant i < j \leqslant n.$$

Divided differences can be nicely represented in a table as follows.

i	x_i	$f[x_i]$	$f[x_{i-1}, x_i]$	$f[x_{i-2}, x_{i-1}, x_i]$	$\cdots$	$f[x_0, \ldots, x_n]$
0	x_0	$f(x_0)$				
1	x_1	$f(x_1)$	$f[x_0, x_1]$			
2	x_2	$f(x_2)$	$f[x_1, x_2]$	$f[x_0, x_1, x_2]$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	
n	x_n	$f(x_n)$	$f[x_{n-1}, x_n]$	$f[x_{n-2}, x_{n-1}, x_n]$	$\cdots$	$f[x_0, \ldots, x_n]$

Note that the coefficients $c_j = f[x_0, \ldots, x_j]$ of the Newton polynomial appear along the diagonal of the table.

Let's complete the divided difference table for the data

$$(x_0, y_0) = (2, 1)$$
$$(x_1, y_1) = (6, 2)$$
$$(x_2, y_2) = (4, 3)$$
$$(x_3, y_3) = (8, 2)$$

i	x_i	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot]$
0	2	1			
1	6	2	$\frac{2-1}{6-2}$		
2	4	3	$\frac{3-2}{4-6}$		
3	8	2	$\frac{2-3}{8-4}$		

i	x_i	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot]$
0	2	1			
1	6	2	$\frac{1}{4}$		
2	4	3	$-\frac{1}{2}$	$\frac{-\frac{1}{2}-\frac{1}{4}}{4-2}$	
3	8	2	$-\frac{1}{4}$	$\frac{-\frac{1}{4}+\frac{1}{2}}{8-6}$	

i	x_i	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot]$
0	2	1			
1	6	2	$\frac{1}{4}$		
2	4	3	$-\frac{1}{2}$	$-\frac{3}{8}$	
3	8	2	$-\frac{1}{4}$	$\frac{1}{8}$	$\frac{\frac{1}{8}+\frac{3}{8}}{8-2}$

i	x_i	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot]$
0	2	1			
1	6	2	$\frac{1}{4}$		
2	4	3	$-\frac{1}{2}$	$-\frac{3}{8}$	
3	8	2	$-\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{12}$

Thus, the interpolating polynomial is

$$p(x) = 1 + \frac{1}{4}(x-2) - \frac{3}{8}(x-2)(x-6) + \frac{1}{12}(x-2)(x-6)(x-4).$$

Note that the Newton form can be evaluated using a nested approach:

$$p(x) = 1 + (x-2)\left(\frac{1}{4} + (x-6)\left(-\frac{3}{8} + \frac{1}{12}(x-4)\right)\right).$$

- **Divided differences in Julia**:

```
0  ################################
1  function divdif(x::Vector{T}, y::Vector{T}) where
   ↪  T<:Union{AbstractFloat,Rational}
2
3      n = length(x)
4      Table = zeros(T, n, n)
5
6      Table[:, 1] = y
7      for k = 2:n
8          for i=k:n
9              Table[i, k] = Table[i, k-1] - Table[i-1, k-1]
10             Table[i, k] /= x[i] - x[i-k+1]
11         end
12     end
13
14     c = diag(Table)
15
16     return c, Table
17 end
18 ################################
```

- **Julia: Evaluate Newton interpolation**:

```
0  ################################
1  function evalnewt(
2          xspan::AbstractVector,
3          x::Vector,
4          c::Vector)
5      n = length(x)
6
7      p = c[n]*ones(length(xspan))
8      for i = 1:length(p)
9          for j = n-1:-1:1
10             p[i] *= xspan[i] - x[j]
11             p[i] += c[j]
12         end
13     end
14
15     return p
16 end
17
18 evalnewt(xx::Real, x::Vector, c::Vector) = evalnewt([xx], x,
   ↪  c)[1]
19 ################################
```

- **The error in polynomial interpolation**: Suppose we are approximating a function f over the interval $[a, b]$ by using the unique polynomial p_n of degree at most n that interpolates the $n + 1$ points

$$(x_0, f(x_0)), \ldots, (x_n, f(x_n)).$$

We will assume that some $x_i = a$ and some $x_j = b$ (i.e., we are only considering **interpolation error**, not **extrapolation error**).

We want to measure the **approximation error** at some $\bar{x} \in [a, b]$:

$$e_n(\bar{x}) = f(\bar{x}) - p_n(\bar{x}).$$

Let p_{n+1} be the polynomial that interpolates the $n + 2$ points

$$(x_0, f(x_0)), \ldots, (x_n, f(x_n)), (\bar{x}, f(\bar{x})).$$

Then

$$p_{n+1}(\bar{x}) = f(\bar{x}).$$

Also, we know that to obtain $p_{n+1}(x)$ from $p_n(x)$, we just need to add a multiple of the Newton basis polynomial,

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

Specifically, we have

$$p_{n+1}(x) = p_n(x) + f[x_0, \ldots, x_n, \bar{x}]\phi_n(x).$$

Thus,

$$\begin{aligned} e_n(\bar{x}) &= f(\bar{x}) - p_n(\bar{x}) \\ &= p_{n+1}(\bar{x}) - p_n(\bar{x}) \\ &= f[x_0, \ldots, x_n, \bar{x}]\phi_n(\bar{x}). \end{aligned}$$

Therefore, the **approximation error** is

$$f(\bar{x}) - p_n(\bar{x}) = f[x_0, \ldots, x_n, \bar{x}] \prod_{i=0}^{n}(\bar{x} - x_i).$$

Note that we need to know $f(\bar{x})$ to compute the right-hand-side of this formula. We need to estimate the divided difference.

- **Bounding the approximation error**: We will use the following generalization of the Mean Value Theorem to bound the approximation error.

  Theorem (Divided Difference and Derivative): Let:

  - $f \in C^n[a, b]$,
  - $x_0, \ldots, x_n \in [a, b]$ be distinct.

  Then there is a ξ somewhere between $x_0, \ldots, x_n$ such that

  $$f[x_0, \ldots, x_n] = \frac{f^{(n)}(\xi)}{n!}.$$

  Proof. Let $p(x)$ be the unique polynomial of degree at most n that interpolates

  $$(x_0, f(x_0)), \ldots, (x_n, f(x_n)),$$

  and let $e(x) = f(x) - p(x)$. Note that $e(x)$ has $n + 1$ roots at $x_0, \ldots, x_n$.

  $$e(x_0) = \cdots = e(x_n) = 0.$$

  Note that $e \in C^n[a, b]$.

Then by **Rolle's Theorem** (or the **Mean Value Theorem**), there are $\xi_0^{(1)}, \ldots, \xi_{n-1}^{(1)}$ points between the points $x_0, \ldots, x_n$ such that

$$e'(\xi_0^{(1)}) = \cdots = e'(\xi_{n-1}^{(1)}) = 0.$$

Now there are $\xi_0^{(2)}, \ldots, \xi_{n-2}^{(2)}$ points between the points $\xi_0^{(1)}, \ldots, \xi_{n-1}^{(1)}$ such that

$$e''(\xi_0^{(2)}) = \cdots = e''(\xi_{n-2}^{(2)}) = 0.$$

We can keep repeating this process and finally conclude that there is a point $\xi = \xi_0^{(n)}$ somewhere between the points $x_0, \ldots, x_n$ such that

$$e^{(n)}(\xi) = 0.$$

Now, since

$$p(x) = f[x_0, \ldots, x_n]x^n + \ldots,$$

and $e(x) = f(x) - p(x)$, we have

$$e^{(n)}(x) = f^{(n)}(x) - f[x_0, \ldots, x_n]n!.$$

Plugging in ξ, we have

$$0 = f^{(n)}(\xi) - f[x_0, \ldots, x_n]n!,$$

which implies that

$$f[x_0, \ldots, x_n] = \frac{f^{(n)}(\xi)}{n!}. \quad \blacksquare$$

Note: Using the convention that

$$f[\underbrace{x, \ldots, x}_{k \text{ times}}] = \frac{f^{(k)}(x)}{k!}, \qquad \text{for } k = 0, 1, \ldots,$$

we can prove a more general result where the points do not need to be distinct.

Theorem (Divided Difference and Derivative): Let:

- $f \in C^n[a, b]$,
- $x_0, \ldots, x_n \in [a, b]$.

Then there is a ξ somewhere between $x_0, \ldots, x_n$ such that

$$f[x_0, \ldots, x_n] = \frac{f^{(n)}(\xi)}{n!}.$$

Recall from above that the **approximation error** is

$$f(\bar{x}) - p_n(\bar{x}) = f[x_0, \ldots, x_n, \bar{x}] \prod_{i=0}^{n} (\bar{x} - x_i).$$

Assuming that $f \in C^{(n+1)}[a, b]$, there is a $\xi \in (a, b)$ such that

$$f(\bar{x}) - p_n(\bar{x}) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (\bar{x} - x_i).$$

Define the **max-norm** of the function $f^{(n+1)}$ as

$$\left\|f^{(n+1)}\right\| = \max_{x\in[a,b]} \left|f^{(n+1)}(x)\right|.$$

Then we have an upper bound on the absolute approximation error at $\bar{x}$:

$$|f(\bar{x}) - p_n(\bar{x})| \leqslant \frac{\left\|f^{(n+1)}\right\|}{(n+1)!} \left|\prod_{i=0}^{n}(\bar{x} - x_i)\right|.$$

The **maximum approximation error** over the interval $[a, b]$ has the following bound:

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \leqslant \frac{\left\|f^{(n+1)}\right\|}{(n+1)!} \max_{x\in[a,b]} \left|\prod_{i=0}^{n}(x - x_i)\right|.$$

- **Chebyshev interpolation: Improving the approximation**: We know by **Weierstrass' Theorem** that we can approximate any function as close as we like using polynomials. However, using evenly spaced points does not work for the function $f(x) = \frac{1}{1+25x^2}$ over the interval $[-1, 1]$.

  The error bound

  $$\max_{x\in[a,b]} |f(x) - p_n(x)| \leqslant \frac{\left\|f^{(n+1)}\right\|}{(n+1)!} \max_{x\in[a,b]} \left|\prod_{i=0}^{n}(x - x_i)\right|$$

  suggests that we should choose the set of points $x_0, \ldots, x_n$ that **minimizes**

  $$\max_{x\in[a,b]} \left|\prod_{i=0}^{n}(x - x_i)\right|.$$

  Thus, we seek the roots of the **monic polynomial**

  $$\phi_{n+1}(x) = \prod_{i=0}^{n}(x - x_i) = (x - x_0)\cdots(x - x_n)$$

  whose maximum absolute value over $[a, b]$ is minimized.

  For now, we will restrict ourselves to the interval $[a, b] = [-1, 1]$.

  We will see in Chapter 12 that the monic polynomials that achieve the minimum max-absolute-value over $[-1, 1]$ are the **monic Chebyshev polynomials**:

  $$\begin{aligned}
  \tilde{T}_0(x) &= 1 \\
  \tilde{T}_1(x) &= x \\
  \tilde{T}_2(x) &= x^2 - \frac{1}{2} \\
  \tilde{T}_3(x) &= x^3 - \frac{3}{4}x \\
  \tilde{T}_4(x) &= x^4 - x^2 + \frac{1}{8} \\
  &\vdots \\
  \tilde{T}_{n+1}(x) &= x\tilde{T}_n(x) - \frac{1}{2^n}\tilde{T}_{n-1}(x)
  \end{aligned}$$

The **monic Chebyshev polynomials** can also be written as

$$\tilde{T}_{n+1}(x) = \frac{1}{2^n}\cos((n+1)\arccos(x))$$

Note that

$$\max_{x\in[-1,1]} \left|\tilde{T}_{n+1}(x)\right| = \frac{1}{2^n}.$$

The roots of $\tilde{T}_{n+1}$ are called the **Chebyshev points** and are given by

$$x_i = \cos\left(\frac{2i+1}{2(n+1)}\pi\right), \quad i = 0, \ldots, n.$$

We can shift these points to another interval $[a, b]$ using

$$x_i \leftarrow a + \frac{b-a}{2}(x_i + 1), \quad i = 0, \ldots, n.$$

Using the **Chebyshev points**, we obtain the error bound:

$$\max_{x\in[-1,1]} |f(x) - p_n(x)| \leqslant \frac{\left\|f^{(n+1)}\right\|}{2^n(n+1)!}$$

- **Interpolating also derivative values (Hermite cubic interpolation)**: Suppose we want to find a polynomial $p(x)$ that satisfies

$$\begin{aligned} p(x_0) &= f(x_0), \quad p'(x_0) = f'(x_0), \\ p(x_1) &= f(x_1), \quad p'(x_1) = f'(x_1). \end{aligned}$$

We can use these four equations to solve for the four coefficients of a cubic polynomial

$$p(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3.$$

Note that

$$p'(x) = c_1 + 2c_2 x + 3c_3 x^2.$$

Using the monomial basis $\left\{1, x, x^2, x^3\right\}$ we have the linear system:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 \\ 1 & x_1 & x_1^2 & x_1^3 \\ 0 & 1 & 2x_0 & 3x_0^2 \\ 0 & 1 & 2x_1 & 3x_1^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f'(x_0) \\ f'(x_1) \end{bmatrix}.$$

For example: Suppose that

$$\begin{aligned} f(0) &= 0, \quad f'(0) = 1, \\ f(1) &= 1, \quad f'(1) = -1. \end{aligned}$$

Then we need to solve

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ -1 \end{bmatrix}.$$

Therefore, the polynomial is

$$p(x) = x + 2x^2 - 2x^3.$$

- **Using divided differences**: Another approach is to use Newton's form and divided differences.

  We will use the convention that

  $$f[x_i, x_i] = f'(x_i).$$

  Thus, we just need to complete the following table.

x_i	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot]$
x_0	$f(x_0)$			
x_0	$f(x_0)$	$f'(x_0)$		
x_1	$f(x_1)$	$f[x_0,x_1]$	$f[x_0,x_0,x_1]$	
x_1	$f(x_1)$	$f'(x_1)$	$f[x_0,x_1,x_1]$	$f[x_0,x_0,x_1,x_1]$

  Then

  $$p(x) = f(x_0)+f'(x_0)(x-x_0)+f[x_0,x_0,x_1](x-x_0)^2+f[x_0,x_0,x_1,x_1](x-x_0)^2(x-x_1).$$

1.5 Piecewise polynomial interpolation

- **The case for piecewise polynomial interpolation**: In Chapter 10, we studied how to fit a single polynomial $p_n(x)$ to a function f over an interval $[a, b]$ by requiring

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

for some $x_0, \ldots, x_n \in [a, b]$. There are several shortcomings of this **global** approach:

1. We have seen that the error

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i)$$

may be large if $\frac{\left\| f^{(n+1)} \right\|}{(n+1)!}$ is large.

2. High order polynomials "wiggle" too much.
3. Polynomials are infinitely smooth, but the function f may not have this property.
4. Changing even just a single data point may dramatically change the entire interpolating polynomial.

In this chapter, we will take a **local approach. We will break the interval** $[a, b]$ **into** r **subintervals** $[t_{i-1}, t_i]$ **using the break points (also called knots**):

$$a = t_0 < t_1 < \cdots < t_r = b.$$

Over each subinterval $[t_{i-1}, t_i]$, we will approximate f with the following low-order polynomials:

1. **Constant**:
$$p_i(x) = c_i, \quad x \in [t_{i-1}, t_i],$$
2. **Linear**:
$$p_i(x) = a_i + b_i x, \quad x \in [t_{i-1}, t_i],$$
3. **Cubic**:
$$p_i(x) = a_i + b_i x + c_i x^2 + d_i x^3, \quad x \in [t_{i-1}, t_i].$$

We get the **piecewise polynomial** function

$$p(x) = \begin{cases} p_1(x), & \text{if } x \in [t_0, t_1), \\ p_2(x), & \text{if } x \in [t_1, t_2), \\ \vdots & \\ p_r(x), & \text{if } x \in [t_{r-1}, t_r], \end{cases}$$

which gives us a **global** approximation of $f(x)$ over $[a, b]$.

- **Piecewise linear (a.k.a. broken line) interpolation**: Let $p(x)$ be the piecewise linear function that interpolates the points

$$(x_i, f(x_i)), \quad i = 0, \ldots, n.$$

Then p must satisfy

$$p(x_i) = f(x_i), \quad i = 0, \ldots, n.$$

We use the interpolation points x_i as the breakpoints t_i giving us n subintervals

$$[x_{i-1}, x_i], \quad i = 1, \ldots, n.$$

The linear piece $p_i(x)$ is defined over $[x_{i-1}, x_i]$ and satisfies

$$p_i(x_{i-1}) = f(x_{i-1}), \quad p_i(x_i) = f(x_i).$$

Using the Newton form we have:

$$p_i(x) = f(x_{i-1}) + f[x_{i-1}, x_i](x - x_{i-1}), \quad x \in [x_{i-1}, x_i].$$

- **Theorem (Piecewise linear error bound)**: Let
  - $f \in C^2[a, b]$,
  - $\|f''\| = \max_{\xi \in [a,b]} |f''(\xi)|$,
  - $a = x_0 < x_1 < \cdots < x_n = b$,
  - p be the piecewise linear function that interpolates the points $\{(x_i, f(x_i))\}_{i=0}^n$.

  If

$$h = \max_{1 \leqslant i \leqslant n} (x_i - x_{i-1})$$

  is the maximum subinterval length, then

$$|f(x) - p(x)| \leqslant \frac{h^2}{8} \|f''\|, \quad \forall x \in [a, b].$$

  If, in addition, the points $x_0, \ldots, x_n$ are evenly spaced, then $h = (b - a)/n$, so

$$|f(x) - p(x)| = \mathcal{O}\left(\frac{1}{n^2}\right).$$

Proof. Recall that if $f \in C^{(n+1)}[a, b]$ and p is the unique polynomial of degree at most n that interpolates the points $\{(x_i, f(x_i))\}_{i=0}^n$, then there is a $\xi \in (a, b)$ such that the interpolation error is given by

$$f(x) - p(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i).$$

Let $x \in [a, b]$. Then $x \in [x_{i-1}, x_i]$, for some $i = 1, \ldots, n$, so $p(x) = p_i(x)$.

By the above interpolation error formula, there exists $\xi \in (x_{i-1}, x_i)$ such that

$$f(x) - p_i(x) = \frac{f''(\xi)}{2!}(x - x_{i-1})(x - x_i),$$

so we have

$$|f(x) - p_i(x)| = \frac{|f''(\xi)|}{2} |(x - x_{i-1})(x - x_i)|.$$

Define

$$g(x) = |(x - x_{i-1})(x - x_i)|, \quad x \in [x_{i-1}, x_i].$$

Then $g(x) = -(x - x_{i-1})(x - x_i)$, so

$$g'(x) = -(x - x_{i-1}) - (x - x_i) = -2x + x_{i-1} + x_i,$$

and $g''(x) = -2$. Therefore, $g(x)$ is *maximized* when

$$x = \frac{x_{i-1} + x_i}{2},$$

and the maximum value of $g(x)$ is

$$-\left(\frac{x_{i-1}+x_i}{2}-x_{i-1}\right)\left(\frac{x_{i-1}+x_i}{2}-x_i\right)=\left(\frac{x_i-x_{i-1}}{2}\right)^2\leqslant\frac{h^2}{4}.$$

Therefore,

$$\begin{aligned}|f(x)-p(x)| &= |f(x)-p_i(x)|\\ &= \frac{|f''(\xi)|}{2}\,|(x-x_{i-1})(x-x_i)|\\ &\leqslant \frac{|f''(\xi)|}{2}\frac{h^2}{4}\\ &\leqslant \frac{h^2}{8}\,\|f''\|.\end{aligned}$$

Since $x\in[a,b]$ was arbitrarily chosen, this error bound holds for all $x\in[a,b]$. ■

- **Piecewise cubic Hermite interpolation**: Over each subinterval $[x_{i-1},x_i]$, we want

$$p_i(x)=a_i+b_i(x-x_{i-1})+c_i(x-x_{i-1})^2+d_i(x-x_{i-1})^3$$

where

$$\begin{aligned}p_i(x_{i-1}) &= f(x_{i-1})\\ p_i(x_i) &= f(x_i)\\ p_i'(x_{i-1}) &= f'(x_{i-1})\\ p_i'(x_i) &= f'(x_i).\end{aligned}$$

Using these four equations, we can solve for the coefficients a_i, b_i, c_i, and d_i.

Note that

$$p_i'(x)=b_i+2c_i(x-x_{i-1})+3d_i(x-x_{i-1})^2.$$

Let $h_i=x_i-x_{i-1}$. Then

$$\begin{bmatrix}1&0&0&0\\1&h_i&h_i^2&h_i^3\\0&1&0&0\\0&1&2h_i&3h_i^2\end{bmatrix}\begin{bmatrix}a_i\\b_i\\c_i\\d_i\end{bmatrix}=\begin{bmatrix}f(x_{i-1})\\f(x_i)\\f'(x_{i-1})\\f'(x_i)\end{bmatrix}.$$

- **Cubic Hermite error bound**: Using the Newton interpolating form, we can write:

$p_i(x)=f[x_{i-1}]+f[x_{i-1},x_{i-1}](x-x_{i-1})+f[x_{i-1},x_{i-1},x_i](x-x_{i-1})^2+f[x_{i-1},x_{i-1},x_i,x_i](x-x_{i-1})^2(x-x_i)$

In Section 10.7 of Ascher-Greif, it is mentioned that the error in polynomial interpolation of 10.5 extends "seamlessly" to the case of interpolating derivatives. Thus, we have

$$|f(\bar{x})-p_i(\bar{x})|\leqslant\frac{\|f''''\|}{4!}\max_{x\in[x_{i-1},x_i]}(x-x_{i-1})^2(x-x_i)^2,\quad\bar{x}\in[x_{i-1},x_i].$$

Letting $g(x)=(x-x_{i-1})^2(x-x_i)^2$, we want to solve

$$\max_{x\in[x_{i-1},x_i]}g(x).$$

After doing some calculus, we find the maximum value of g is $(x_i - x_{i-1})^4/16$.

Therefore, for $\bar{x} \in [x_{i-1}, x_i]$, we have

$$\begin{aligned} |f(\bar{x}) - p_i(\bar{x})| &\leqslant \frac{\|f''''\|}{4!} \max_{x \in [x_{i-1}, x_i]} (x - x_{i-1})^2 (x - x_i)^2 \\ &= \frac{\|f''''\|}{4!} \frac{1}{16} (x_i - x_{i-1})^4. \end{aligned}$$

If we again let

$$h = \max_{1 \leqslant i \leqslant n} (x_i - x_{i-1}),$$

then we have

$$|f(\bar{x}) - p_i(\bar{x})| \leqslant \frac{h^4}{384} \|f''''\|, \quad \bar{x} \in [x_{i-1}, x_i].$$

This bound is valid for all subintervals $[x_{i-1}, x_i]$, so we have

$$|f(x) - p(x)| \leqslant \frac{h^4}{384} \|f''''\|, \quad x \in [a, b].$$

- **Theorem: (Piecewise cubic Hermite error bound)**: Let
  - $f \in C^4[a, b]$,
  - $\|f''''\| = \max_{\xi \in [a,b]} |f''''(\xi)|$,
  - $a = x_0 < x_1 < \cdots < x_n = b$,
  - p be the piecewise cubic Hermite function described above.

  If

  $$h = \max_{1 \leqslant i \leqslant n} (x_i - x_{i-1})$$

  is the maximum subinterval length, then

  $$|f(x) - p(x)| \leqslant \frac{h^4}{384} \|f''''\|, \quad \forall x \in [a, b].$$

  If, in addition, the points $x_0, \ldots, x_n$ are evenly spaced, then $h = (b - a)/n$, so

  $$|f(x) - p(x)| = \mathcal{O}\left(\frac{1}{n^4}\right).$$

- **Cubic spline interpolation: Introduction**: Suppose now that we do not have access to f'.

  What additional conditions should we use to determine the piecewise cubic polynomial?

  Each cubic piece is determined by four coefficients:

  $$p_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad x \in [x_i, x_{i+1}], \quad i = 0, \ldots, n - 1.$$

  There are n subintervals, so we have a total of $4n$ coefficients to determine.

Thus, we will need a total of $4n$ equations to determine all coefficients.

- **Continuity and differentiability conditions**: Enforcing **continuity** (i.e., $p \in C^0[a,b]$) gives us $2n$ equations:

$$p_i(x_i) = f(x_i), \quad i = 0, \ldots, n-1 \quad \text{(left)}$$
$$p_i(x_{i+1}) = f(x_{i+1}), \quad i = 0, \ldots, n-1 \quad \text{(right)}$$

Enforcing **first derivative continuity** (i.e., $p \in C^1[a,b]$) gives us $n-1$ equations:

$$p_i'(x_{i+1}) = p_{i+1}'(x_{i+1}), \quad i = 0, \ldots, n-2$$

Enforcing **second derivative continuity** (i.e., $p \in C^2[a,b]$) gives us another $n-1$ equations:

$$p_i''(x_{i+1}) = p_{i+1}''(x_{i+1}), \quad i = 0, \ldots, n-2$$

Thus, we have a total of $4n-2$ equations. This is not enough to determine the $4n$ coefficients, so we need to choose two more conditions.

- **Two additional conditions**:

1. **Free boundary**:

$$p''(x_0) = p''(x_n) = 0$$

Gives us the **natural spline**. Often it is not a good choice since it is usually unreasonable to assume that $f''(x_0) = f''(x_n) = 0$.

2. **Clamped boundary**:

$$p'(x_0) = f'(x_0), \quad p'(x_n) = f'(x_n)$$

Gives us the **complete spline**. However, this option is not possible if the values of $f'(x_0)$ and $f'(x_n)$ are unknown.

3. **Not-a-knot**:

$$p_0'''(x_1) = p_1'''(x_1), \quad p_{n-2}'''(x_{n-1}) = p_{n-1}'''(x_{n-1})$$

This means that x_1 and x_{n-1} are no longer knots (i.e., break points) since p_0 and p_1 become a single cubic polynomial, as do p_{n-2} and p_{n-1}.

- **Computing the Cubic Spline**: We start with

$$p_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$
$$p_i'(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2$$
$$p_i''(x) = 2c_i + 6d_i(x - x_i)$$

Solving for a_i: The **left continuity equations** $p_i(x_i) = f(x_i)$ imply that

$$a_i = f(x_i), \quad i = 0, \ldots, n-1.$$

Solving for d_i: The **second derivative continuity equations** $p_i''(x_{i+1}) = p_{i+1}''(x_{i+1})$ become

$$2c_i + 6d_i(x_{i+1} - x_i) = 2c_{i+1} + 6d_{i+1}(x_{i+1} - x_{i+1}), \quad i = 0, \ldots, n-2.$$

We let

$$h_i = x_{i+1} - x_i, \quad i = 0, \ldots, n-1.$$

Therefore,

$$c_i + 3d_i h_i = c_{i+1}, \quad i = 0, \ldots, n-2.$$

We define

$$c_n = c_{n-1} + 3d_{n-1}h_{n-1}.$$

Then

$$d_i = \frac{c_{i+1} - c_i}{3h_i}, \quad i = 0, \ldots, n-1.$$

Solving for b_i: The **right interpolation equations** $p_i(x_{i+1}) = f(x_{i+1})$ imply that

$$a_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 + d_i(x_{i+1} - x_i)^3 = f(x_{i+1}), \quad i = 0, \ldots, n-1.$$

Since $a_i = f(x_i)$, we have

$$f(x_i) + b_i h_i + c_i h_i^2 + d_i h_i^3 = f(x_{i+1}), \quad i = 0, \ldots, n-1.$$

Simplifying and substituting formula for d_i, we obtain

$$b_i + c_i h_i + \frac{c_{i+1} - c_i}{3h_i} h_i^2 = \frac{f(x_{i+1}) - f(x_i)}{h_i}, \quad i = 0, \ldots, n-1.$$

This further simplifies to

$$b_i + \frac{h_i}{3}(c_{i+1} + 2c_i) = f[x_i, x_{i+1}], \quad i = 0, \ldots, n-1.$$

Therefore,

$$b_i = f[x_i, x_{i+1}] - \frac{h_i}{3}(c_{i+1} + 2c_i), \quad i = 0, \ldots, n-1.$$

Solving for c_i: The **first derivative continuity equations** $p_i'(x_{i+1}) = p_{i+1}'(x_{i+1})$ become

$$b_i + 2c_i(x_{i+1} - x_i) + 3d_i(x_{i+1} - x_i)^2 = b_{i+1} + 2c_{i+1}(x_{i+1} - x_{i+1}) + 3d_{i+1}(x_{i+1} - x_{i+1})^2, \quad i = 0, \ldots, n-2.$$

Again substituting the formula for d_i, we get

$$b_i + 2c_i h_i + 3\frac{c_{i+1} - c_i}{3h_i} h_i^2 = b_{i+1}, \quad i = 0, \ldots, n-2.$$

Simplifying,

$$b_i + (c_{i+1} + c_i)h_i = b_{i+1}, \quad i = 0, \ldots, n-2.$$

Shifting the index, we have

$$b_{i-1} + (c_i + c_{i-1})h_{i-1} = b_i, \quad i = 1, \ldots, n-1.$$

Substituting the formula for b_i and b_{i-1}, we obtain

$$f[x_{i-1}, x_i] - \frac{h_{i-1}}{3}(c_i + 2c_{i-1}) + (c_i + c_{i-1})h_{i-1} = f[x_i, x_{i+1}] - \frac{h_i}{3}(c_{i+1} + 2c_i), \quad i = 1, \ldots, n-1.$$

Moving the c terms to the LHS and everything else to the RHS, we obtain

$$h_i(c_{i+1} + 2c_i) - h_{i-1}(c_i + 2c_{i-1}) + 3h_{i-1}(c_i + c_{i-1}) = 3\left(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]\right), \quad i = 1, \ldots, n-1.$$

Combining common terms, we have

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = 3\left(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]\right), \quad i = 1, \ldots, n-1.$$

Summary:

$$a_i = f(x_i), \quad i = 0, \ldots, n-1$$

$$b_i = f[x_i, x_{i+1}] - \frac{h_i}{3}(c_{i+1} + 2c_i), \quad i = 0, \ldots, n-1$$

$$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = 3\left(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]\right), \quad i = 1, \ldots, n-1$$

$$d_i = \frac{c_{i+1} - c_i}{3h_i}, \quad i = 0, \ldots, n-1$$

- **Free boundary** $p_0''(x_0) = p_{n-1}''(x_n) = 0$

  The condition $p_0''(x_0) = 0$ gives us

  $$2c_0 + 6d_0(x_0 - x_0) = 0 \quad \Longrightarrow \quad c_0 = 0.$$

  The condition $p_{n-1}''(x_n) = 0$ gives us

  $$2c_{n-1} + 6d_{n-1}(x_n - x_{n-1}) = 0 \quad \Longrightarrow \quad c_n = c_{n-1} + 3d_{n-1}h_{n-1} = 0.$$

  The linear system

  $$h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = 3\left(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]\right), \quad i = 1, \ldots, n-1$$

  can thus be written as

  $$\begin{bmatrix} 2(h_0+h_1) & h_1 & & & & \\ h_1 & 2(h_1+h_2) & h_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & h_{n-3} & 2(h_{n-3}+h_{n-2}) & h_{n-2} & \\ & & & h_{n-2} & 2(h_{n-2}+h_{n-1}) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-2} \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_{n-2} \\ \psi_{n-1} \end{bmatrix},$$

  where $\psi_i = 3\left(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]\right)$, for $i = 1, \ldots, n-1$.

  The above matrix is **symmetric tridiagonal**.

  Since the matrix is **strictly diagonally dominant**, the Gershgorin circle theorem implies that all its eigenvalues are positive, so it is a **positive definite** matrix.

  Thus, the matrix is **nonsingular** and the system can be solved in $\mathcal{O}(n)$ flops.

- **Clamped boundary:** $p'(x_0) = f'(x_0)$, $p'(x_n) = f'(x_n)$: The condition $p_0'(x_0) = f'(x_0)$ gives us

  $$b_0 + 2c_0(x_0 - x_0) + 3d_0(x_0 - x_0)^2 = f'(x_0) \quad \Longrightarrow \quad b_0 = f'(x_0)$$

  The condition $p_{n-1}'(x_n) = f'(x_n)$ gives us

  $$b_{n-1} + 2c_{n-1}(x_n - x_{n-1}) + 3d_{n-1}(x_n - x_{n-1})^2 = f'(x_n),$$

  which implies that

  $$b_{n-1} + 2c_{n-1}h_{n-1} + 3d_{n-1}h_{n-1}^2 = f'(x_n).$$

Recall:

$$b_i = f[x_i, x_{i+1}] - \frac{h_i}{3}(c_{i+1} + 2c_i), \quad i = 0, \ldots, n-1$$

$$d_i = \frac{c_{i+1} - c_i}{3h_i}, \quad i = 0, \ldots, n-1$$

Therefore, $b_0 = f'(x_0)$ becomes

$$f[x_0, x_1] - \frac{h_0}{3}(c_1 + 2c_0) = f'(x_0)$$

which gives us

$$2h_0c_0 + h_0c_1 = 3\left(f[x_0, x_1] - f'(x_0)\right)$$

In addition, $b_{n-1} + 2c_{n-1}h_{n-1} + 3d_{n-1}h_{n-1}^2 = f'(x_n)$ becomes

$$f[x_{n-1}, x_n] - \frac{h_{n-1}}{3}(c_n + 2c_{n-1}) + 2c_{n-1}h_{n-1} + 3\frac{c_n - c_{n-1}}{3h_{n-1}}h_{n-1}^2 = f'(x_n).$$

Simplifying, we get

$$-h_{n-1}(c_n + 2c_{n-1}) + 6c_{n-1}h_{n-1} + 3(c_n - c_{n-1})h_{n-1} = 3\left(f'(x_n) - f[x_{n-1}, x_n]\right),$$

which becomes

$$h_{n-1}c_{n-1} + 2h_{n-1}c_n = 3\left(f'(x_n) - f[x_{n-1}, x_n]\right).$$

The linear system

$$\begin{gathered}
2h_0c_0 + h_0c_1 = 3\left(f[x_0, x_1] - f'(x_0)\right) \\
h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_ic_{i+1} = 3\left(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]\right), \quad i = 1, \ldots, n-1 \\
h_{n-1}c_{n-1} + 2h_{n-1}c_n = 3\left(f'(x_n) - f[x_{n-1}, x_n]\right)
\end{gathered}$$

can thus be written as

$$\begin{bmatrix}
2h_0 & h_0 & & & & & \\
h_0 & 2(h_0 + h_1) & h_1 & & & & \\
 & \ddots & \ddots & \ddots & & & \\
 & & & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\
 & & & & h_{n-1} & 2h_{n-1}
\end{bmatrix}
\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} =
\begin{bmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \\ \psi_n \end{bmatrix},$$

where

$$\begin{aligned}
\psi_0 &= 3\left(f[x_0, x_1] - f'(x_0)\right), \\
\psi_i &= 3\left(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]\right), \quad i = 1, \ldots, n-1, \\
\psi_n &= 3\left(f'(x_n) - f[x_{n-1}, x_n]\right).
\end{aligned}$$

Again, the above matrix is **symmetric tridiagonal** and **strictly diagonally dominant**.

Thus, the matrix is **positive definite**, hence **nonsingular**, and the linear system can be solved in $\mathcal{O}(n)$ flops.

- **Error bounds**

  1. **Clamped ends**:

$$\max_{x \in [a,b]} |f(x) - p(x)| \leqslant \frac{5}{384} \|f''''\| h^4$$

2. **Not-a-Knot**:

$$\max_{x\in[a,b]} |f(x) - p(x)| \approx \frac{5}{384} \|f''''\| h^4$$

Impressively, we obtain $\mathcal{O}(h^4)$ methods, the same order as the piecewise cubic Hermite interpolation, but only using half the information about the function f.

- **Parametric curves**: In this last section, we look at interpolating a set of points

$$(x_0, y_0), \ldots, (x_n, y_n)$$

by a **parametric curve**

$$x = x(t), \quad y = y(t), \quad t \in [a, b].$$

A simple approach is to separately interpolate the x-coordinate data

$$(t_0, x_0), \ldots, (t_n, x_n)$$

and the y-coordinate data

$$(t_0, y_0), \ldots, (t_n, y_n)$$

where $t_i = i/n \in [0, 1]$.

- **Parametric piecewise cubic Hermite polynomials**: Piecewise interpolation using cubic Hermite polynomials provides a much better solution.

We will draw a curve between two points (x_0, y_0) and (x_1, y_1), and we will use **two guidepoints** $(x_0 + \alpha_0, y_0 + \beta_0)$ and $(x_1 - \alpha_1, y_1 - \beta_1)$ to specify the slope of the curve at the endpoints.

Specifically, **Bezier polynomials** are cubic Hermite polynomials $p_x(t)$ and $p_y(t)$ that satisfy

$$\begin{array}{llll} p_x(0) = x_0, & p_x(1) = x_1, & p_x'(0) = 3\alpha_0, & p_x'(1) = 3\alpha_1 \\ p_y(0) = y_0, & p_y(1) = y_1, & p_y'(0) = 3\beta_0, & p_y'(1) = 3\beta_1 \end{array}$$

The factor of 3 is there to ensure that the curve lies within the convex hull of the four points.

It can be shown that

$$\begin{aligned} p_x(t) &= \big(2(x_0 - x_1) + 3(\alpha_0 + \alpha_1)\big)t^3 + \big(3(x_1 - x_0) - 3(\alpha_1 + 2\alpha_0)\big)t^2 + 3\alpha_0 t + x_0 \\ p_y(t) &= \big(2(y_0 - y_1) + 3(\beta_0 + \beta_1)\big)t^3 + \big(3(y_1 - y_0) - 3(\beta_1 + 2\beta_0)\big)t^2 + 3\beta_0 t + y_0 \end{aligned}$$

1.6 Continuous least squares approximation

- **Recall:** In the previous two chapters we approximated a function $f(x)$ over an interval $[a,b]$ by interpolating points $\{(x_i, f(x_i))\}_{i=0}^n$ (and possibly values of f') by either a polynomial $p_n(x)$ having degree at most n, or by a piecewise polynomial (e.g., a cubic spline) $p(x)$.

  The **error of the approximation** was measured using the **infinity-norm**:

  $$\|f-p\|_\infty := \max_{x\in[a,b]} |f(x)-p(x)|\,.$$

  We found valid upper bounds on this error, such as

  $$\|f-p_n\|_\infty \leqslant \frac{\left\|f^{(n+1)}\right\|_\infty}{(n+1)!} \max_{x\in[a,b]} \prod_{i=0}^{n} |x-x_i|\,,$$

  for the unique polynomial $p_n(x)$ having degree at most n interpolating $(x_0, f(x_0)), \ldots, (x_n, f(x_n))$.

  We even looked an minimizing this upper bound, which led us to studying **Chebyshev points**.

- **Best approximation:** We now will look at minimizing the **approximation error** directly. Given a set of **linearly independent** functions $\{\phi_j(x)\}_{j=0}^n$, we want to find a function

  $$p(x) = \sum_{j=0}^{n} c_j \phi_j(x)$$

  that minimizes the error $\|f-p\|$, where $\|\cdot\|$ is a **function norm**.

- **Function norms**: Let $\|\cdot\|$ be a **norm** for $f \in C[a,b]$.

  Then $\|\cdot\|$ is a function that takes an input f and returns a real number.

  A norm $\|\cdot\|$ for functions must satisfy:

  1. $\|f\| \geqslant 0$, for all functions f, and $\|f\| = 0$ if and only if $f(x) = 0$, for all $x \in [a,b]$;
  2. $\|\alpha f\| = |\alpha|\,\|f\|$, for all $\alpha \in \mathbb{R}$;
  3. $\|f+g\| \leqslant \|f\| + \|g\|$ for all $f, g \in C[a,b]$.

- **Examples of function norms**: The L_1 norm of a function $f \in C[a,b]$ is defined as

  $$\|f\|_1 := \int_a^b |f(x)|\,dx.$$

  The L_∞ norm of a function $f \in C[a,b]$ is defined as

  $$\|f\|_\infty := \max_{x\in[a,b]} |f(x)|\,.$$

  We will define the L_2 norm in terms of the inner-product.

- L_2 **norm**: The **inner-product** of two functions $f, g \in C[a,b]$ is defined as

  $$\langle f, g\rangle := \int_a^b f(x)g(x)\,dx.$$

  Note that we can use the inner-product to talk about the **angle** between functions. For example, functions f and g are said to be **orthogonal** if $\langle f, g\rangle = 0$.

The L_2 norm of a function $f \in C[a,b]$ is defined as

$$\|f\|_2 := \sqrt{\langle f, f\rangle} = \left(\int_a^b [f(x)]^2\, dx\right)^{\frac{1}{2}}.$$

In this chapter, we will consider methods for minimizing $\|f-p\|_2$, also known as the **continuous least-squares problem**.

- **Solving the continuous least-squares problem**: We want to find the coefficients $c_0, \ldots, c_n$ such that

$$p(x) = \sum_{j=0}^{n} c_j \phi_j(x)$$

minimizes $\|f-p\|_2$ over all functions $p \in \text{Span}\{\phi_0, \ldots, \phi_n\}$.

Equivalently, we can minimize

$$\begin{aligned}
\|f-p\|_2^2 &= \langle f-p, f-p\rangle \\
&= \langle f, f\rangle - 2\langle f, p\rangle + \langle p, p\rangle \\
&= \langle f, f\rangle - 2\left\langle f, \sum_{j=0}^{n} c_j\phi_j \right\rangle + \left\langle \sum_{j=0}^{n} c_j\phi_j, \sum_{j=0}^{n} c_j\phi_j \right\rangle \\
&= \langle f, f\rangle - 2\sum_{j=0}^{n} c_j \langle f, \phi_j\rangle + \sum_{j=0}^{n}\sum_{k=0}^{n} c_j c_k \langle \phi_j, \phi_k\rangle \\
&= \langle f, f\rangle - 2b^T c + c^T B c,
\end{aligned}$$

where $B \in \mathbb{R}^{(n+1)\times(n+1)}$ and $b, c \in \mathbb{R}^{n+1}$ are defined as

$$B := \begin{bmatrix} \langle\phi_0,\phi_0\rangle & \cdots & \langle\phi_0,\phi_n\rangle \\ \vdots & \ddots & \vdots \\ \langle\phi_n,\phi_0\rangle & \cdots & \langle\phi_n,\phi_n\rangle \end{bmatrix}, \qquad b := \begin{bmatrix} \langle f,\phi_0\rangle \\ \vdots \\ \langle f,\phi_n\rangle \end{bmatrix}, \qquad c := \begin{bmatrix} c_0 \\ \vdots \\ c_n \end{bmatrix}.$$

- **The normal equations**: Let $g\colon \mathbb{R}^{n+1} \to \mathbb{R}$ be defined as

$$g(c) = c^T B c - 2b^T c + \langle f, f\rangle.$$

So we want to find $c^* \in \mathbb{R}^{n+1}$ such that

$$g(c^*) \leqslant g(c), \quad \forall c \in \mathbb{R}^{n+1}.$$

From multivariable calculus, we know that any **local minimizer** c^* much satisfy $\nabla g(c^*) = 0$.

It is not too difficult to show that

$$\nabla g(c) = 2Bc - 2b.$$

Therefore, we just need to solve the following linear system:

$$Bc^* = b.$$

This linear system is known as the **normal equations** for the **continuous least-squares problem**.

- **Unique solution to the normal equations**: In addition, the matrix B is **symmetric** and **positive definite**.

  It is clear that B is symmetric since $\langle \phi_j, \phi_k \rangle = \langle \phi_k, \phi_j \rangle$.

  To show B is positive definite, we let $c \in \mathbb{R}^{n+1}$ be nonzero, and note that

  $$c^T B c = \langle p, p \rangle = \|p\|_2^2 ,$$

  where $p = \sum_{j=0}^{n} c_j \phi_j$. Since $c \neq 0$, we have that $p \neq 0$ due to the fact the functions $\phi_0, \ldots, \phi_n$ are **linearly independent**. Thus, $c^T B c = \|p\|_2^2 > 0$.

  Positive definite matrices are nonsingular, so B is nonsingular.

  The proof is easy. Suppose that B is singular. Then there is a nonzero vector c such that $Bc = 0$. Then we have

  $$c^T B c = 0,$$

  which contradicts the fact that B is positive definite. Thus B must be nonsingular. Thus, there is a **unique solution** c^* to the normal equations.

- **Unique global minimizer**: Let $\Delta c \in \mathbb{R}^{n+1}$ be nonzero. Then,

  $$\begin{aligned} g(c^* + \Delta c) &= (c^* + \Delta c)^T B (c^* + \Delta c) - 2b^T (c^* + \Delta c) + \langle f, f \rangle \\ &= (c^*)^T B c^* + 2 (\Delta c)^T B c^* + (\Delta c)^T B \Delta c - 2b^T c^* - 2b^T \Delta c + \langle f, f \rangle \\ &= g(c^*) + 2 (\Delta c)^T (B c^* - b) + (\Delta c)^T B \Delta c \\ &= g(c^*) + (\Delta c)^T B \Delta c \\ &> g(c^*). \end{aligned}$$

  Therefore, $g(c^*) < g(c^* + \Delta c)$, for all $\Delta c \neq 0$, which implies that c^* is the **unique global minimizer**.

- **Pros and cons of using the monomial basis for continuous least squares**:
  Pros:

  1. Simple. Over the interval $[a, b]$, we have:

     $$B_{ij} = \frac{(b-a)^{i+j+1}}{i+j+1}, \quad i, j = 0, \ldots, n.$$

     Thus, B is easy to evaluate. When $[a, b] = [0, 1]$, we have the famous Hilbert matrix

     $$B = H_{n+1} = \begin{bmatrix} 1 & \frac{1}{2} & \cdots & \frac{1}{n+1} \\ \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n+2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2n+1} \end{bmatrix}.$$

  Cons:

  1. Solving $Bc = b$ can be expensive for large n.
  2. The matrix B is highly **ill-conditioned** which means that the computed solution c is highly sensitive to errors in the right-hand-side vector b and to the round-off errors that occur during the computation.

- **Orthogonal basis functions**: It would be great if we could find a basis $\phi_0, \ldots, \phi_n$ for which solving $Bc = b$, where

$$B := \begin{bmatrix} \langle \phi_0, \phi_0 \rangle & \cdots & \langle \phi_0, \phi_n \rangle \\ \vdots & \ddots & \vdots \\ \langle \phi_n, \phi_0 \rangle & \cdots & \langle \phi_n, \phi_n \rangle \end{bmatrix}, \qquad b := \begin{bmatrix} \langle f, \phi_0 \rangle \\ \vdots \\ \langle f, \phi_n \rangle \end{bmatrix}.$$

  is trivial.

  The best case is when $B = I$, but having B **diagonal** would also be great.

  We want a basis that satisfies:

$$\langle \phi_i, \phi_j \rangle = 0, \quad i \neq j.$$

  That is, we want $\phi_0, \ldots, \phi_n$ to be **pairwise orthognal**.

  An **orthogonal basis** is a basis that is pairwise orthogonal.

  If $\{\phi_0, \ldots, \phi_n\}$ is an orthogonal basis and

$$\langle \phi_i, \phi_i \rangle = 1, \quad i = 0, \ldots, n,$$

  we say that $\{\phi_0, \ldots, \phi_n\}$ is an **orthonormal basis**.

  Note that

$$B_{ii} = \langle \phi_i, \phi_i \rangle = \|\phi_i\|_2^2.$$

  Thus, to solve $Bc = b$ for an orthogonal basis $\{\phi_0, \ldots, \phi_n\}$ we simply set

$$c_i = \frac{b_i}{\|\phi_i\|_2^2}, \quad i = 0, \ldots, n.$$

  If the basis is orthonormal, then $B = I$, so we have $c = b$.

- **Gram-Schmidt orthogonalization**: Given a basis $\{\psi_0, \ldots, \psi_n\}$, we can create an **orthogonal basis** for

$$\text{Span}\{\psi_0, \ldots, \psi_n\}$$

  using the **Gram-Schmidt process**.

  Suppose we have just two functions $\{\psi_0, \psi_1\}$. First we set

$$\phi_0 := \psi_0.$$

  Now let p be the orthogonal projection of ψ_1 onto $\text{Span}\{\phi_0\}$.

  That is, $p = c_0\phi_0$, where c_0 is found by solving $Bc = b$.

  Since $B = \big[\langle \phi_0, \phi_0 \rangle\big]$ and $b = \big[\langle \psi_1, \phi_0 \rangle\big]$, we have $c_0 = \langle \psi_1, \phi_0 \rangle \,/\, \langle \phi_0, \phi_0 \rangle$, so

$$p = \frac{\langle \psi_1, \phi_0 \rangle}{\langle \phi_0, \phi_0 \rangle} \phi_0.$$

Then we let $\phi_1 = \psi_1 - p$, which is the residual of the projection of ψ_1 onto $\text{Span}\{\phi_0\}$.

That is,

$$\phi_1 := \psi_1 - \frac{\langle \psi_1, \phi_0 \rangle}{\langle \phi_0, \phi_0 \rangle} \phi_0.$$

Recall that the residual of an orthogonal projection is orthogonal to every basis vector.

Thus, $\langle \phi_1, \phi_0 \rangle = 0$, so we have an orthogonal basis.

Additionally, it can be shown that

$$\text{Span}\{\phi_0, \phi_1\} = \text{Span}\{\psi_0, \psi_1\}.$$

Suppose after k steps we have computed an orthogonal basis $\{\phi_0, \ldots, \phi_k\}$ that satisfies

$$\text{Span}\{\phi_0, \ldots, \phi_k\} = \text{Span}\{\psi_0, \ldots, \psi_k\}.$$

As before, we let $p = \sum_{j=0}^{k} c_j \phi_j$ be the projection of ψ_{k+1} onto $\text{Span}\{\phi_0, \ldots, \phi_k\}$.

To do this, we need to solve $Bc = b$:

$$\begin{bmatrix} \langle \phi_0, \phi_0 \rangle & & & \\ & \langle \phi_1, \phi_1 \rangle & & \\ & & \ddots & \\ & & & \langle \phi_k, \phi_k \rangle \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_k \end{bmatrix} = \begin{bmatrix} \langle \psi_{k+1}, \phi_0 \rangle \\ \langle \psi_{k+1}, \phi_1 \rangle \\ \vdots \\ \langle \psi_{k+1}, \phi_k \rangle \end{bmatrix}.$$

Therefore, we have

$$p = \sum_{j=0}^{k} c_j \phi_j = \sum_{j=0}^{k} \frac{\langle \psi_{k+1}, \phi_j \rangle}{\langle \phi_j, \phi_j \rangle} \phi_j.$$

Letting ϕ_{k+1} be the residual of the projection of ψ_{k+1} onto $\text{Span}\{\phi_0, \ldots, \phi_k\}$ (i.e., $\phi_{k+1} = \psi_{k+1} - p$), we obtain

$$\phi_{k+1} := \psi_{k+1} - \sum_{j=0}^{k} \frac{\langle \psi_{k+1}, \phi_j \rangle}{\langle \phi_j, \phi_j \rangle} \phi_j.$$

Since the residual of an orthogonal projection is orthogonal to every basis vector, we have that $\{\phi_0, \ldots, \phi_{k+1}\}$ is orthogonal.

Additionally, we can show that

$$\text{Span}\{\phi_0, \ldots, \phi_{k+1}\} = \text{Span}\{\psi_0, \ldots, \psi_{k+1}\}.$$

- **Summary**: Given a basis $\{\psi_0, \ldots, \psi_n\}$, we can create an orthogonal basis $\{\phi_0, \ldots, \phi_n\}$ that satisfies

  $$\text{Span}\{\phi_0, \ldots, \phi_n\} = \text{Span}\{\psi_0, \ldots, \psi_n\}$$

  using the **Gram-Schmidt process**:

  $$\phi_k := \psi_k - \sum_{j=0}^{k-1} \frac{\langle \psi_k, \phi_j \rangle}{\langle \phi_j, \phi_j \rangle} \phi_j, \quad k = 0, \ldots, n.$$

- **An orthonormal basis**: To obtain an orthonormal basis, we just need to **normalize** each vector by dividing it by its norm:

$$\phi_k \leftarrow \frac{1}{\|\phi_k\|_2}\phi_k, \quad k = 0, \ldots, n.$$

- **Legendre polynomial basis**: The **Legendre polynomials**, named after Adrien-Marie Legendre, form an othogonal basis for the space of polynomials having degree at most n. These polynomials are defined on the interval $[-1, 1]$ and are obtained by performing the **Gram-Schmidt process** on the monomial basis $\left\{1, x, x^2, \ldots, x^n\right\}$.

- **A three-term recurrence relation for Legendre polynomials**: The Legendre polynomials (normalized so that $\phi_k(1) = 1$, for all k) can be described using the following **three-term recurrence relation**:

$$\begin{aligned}
\phi_0(x) &= 1, \\
\phi_1(x) &= x, \\
\phi_{k+1}(x) &= \frac{2k+1}{k+1}x\phi_k(x) - \frac{k}{k+1}\phi_{k-1}(x), \quad k = 1, \ldots, n-1.
\end{aligned}$$

  This remarkable fact means that we only need the previous two polynomials to determine the next polynomial.

- **Best approximation on** $[a, b]$: We go between $x \in [-1, 1]$ and $t \in [a, b]$ using

$$t = \frac{1}{2}\left[(b-a)x + (a+b)\right] \quad \text{and} \quad x = \frac{2t-a-b}{b-a}.$$

  Define

$$\hat{\phi}_i(t) = \phi_i\left(\frac{2t-a-b}{b-a}\right).$$

  Then it can be shown that

$$\left\langle \hat{\phi}_i, \hat{\phi}_j \right\rangle = \begin{cases} 0, & i \neq j, \\ \dfrac{b-a}{2i+1}, & i = j. \end{cases}$$

1.7 Numerical differentiation

- **A first-order method**: In Chapter 1, we experimented with the approximation

$$f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h}$$

and saw that the **discretization error** (a.k.a. **truncation error**) satisfies

$$\left| f'(x_0) - \frac{f(x_0+h)-f(x_0)}{h} \right| \approx \frac{h}{2} |f''(x_0)| = \mathcal{O}(h)$$

when h is small enough and $f''(x_0) \neq 0$. We say that this approximation is **first-order accurate**.

- **A second-order method**: We also saw in Exercise 1.2 from HW1 that

$$f'(x_0) \approx \frac{f(x_0+h)-f(x_0-h)}{2h}$$

and that the truncation error satisfies

$$\left| f'(x_0) - \frac{f(x_0+h)-f(x_0-h)}{2h} \right| \approx \frac{h^2}{6} |f'''(x_0)| = \mathcal{O}(h^2)$$

when h is small enough and $f'''(x_0) \neq 0$. Thus, this approximation is **second-order accurate**.

- **Overview**: In this section, we will use the **Taylor series** of $f(x)$ derive these and other formulas to approximate the first and second derivatives of a function $f(x)$.

  These approximations will use the value of $f(x)$ at evenly-spaced points:

$$\ldots,\ x_0-2h, \quad x_0-h, \quad x_0, \quad x_0+h, \quad x_0+2h,\ \ldots$$

- **Two-point formulas: Forward difference approximation for** $f'(x_0)$ Let f be twice-differentiable on an interval containing x_0 and x_0+h. From the Taylor series, we have

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(\xi), \quad \text{for some } \xi \in (x_0, x_0+h).$$

  Solving for $f'(x_0)$, we obtain

$$f'(x_0) = \frac{f(x_0+h)-f(x_0)}{h} - \frac{h}{2} f''(\xi).$$

  Thus, the **forward** difference formula

$$\frac{f(x_0+h)-f(x_0)}{h}$$

  gives a first order (i.e., $\mathcal{O}(h)$) approximation of $f'(x_0)$:

$$\left| f'(x_0) - \frac{f(x_0+h)-f(x_0)}{h} \right| = \frac{h}{2} |f''(\xi)|, \quad \text{for some } \xi \in (x_0, x_0+h).$$

- **Two point formulas: Backward difference approximation for** $f'(x_0)$: Let f be twice-differentiable on an interval containing x_0 and x_0-h. From the Taylor series, we have

$$f(x_0-h) = f(x_0) - hf'(x_0) + \frac{h^2}{2} f''(\xi), \quad \text{for some } \xi \in (x_0-h, x_0).$$

  Use this Taylor series to obtain the **backward** difference approximation of $f'(x_0)$. Show that this approximation is also first order.

Solving for $f'(x_0)$, we obtain

$$f'(x_0) = \frac{f(x_0) - f(x_0 - h)}{h} + \frac{h}{2} f''(\xi).$$

Thus, the **backward** difference formula

$$\boxed{\frac{f(x_0) - f(x_0 - h)}{h}}$$

gives a first order (i.e., $\mathcal{O}(h)$) approximation of $f'(x_0)$:

$$\left| f'(x_0) - \frac{f(x_0) - f(x_0 - h)}{h} \right| = \frac{h}{2} \left| f''(\xi) \right|, \quad \text{for some } \xi \in (x_0 - h, x_0).$$

- **Three-point formulas: Centered difference approximation for** $f'(x_0)$: Let f be thrice-continuously-differentiable on an interval containing $x_0 - h$ and $x_0 + h$. From the Taylor series, we have

$$f(x_0 + h) = f(x_0) + h f'(x_0) + \frac{h^2}{2} f''(x_0) + \frac{h^3}{6} f'''(\xi_1), \quad \text{for some } \xi_1 \in (x_0, x_0 + h),$$
$$f(x_0 - h) = f(x_0) - h f'(x_0) + \frac{h^2}{2} f''(x_0) - \frac{h^3}{6} f'''(\xi_2), \quad \text{for some } \xi_2 \in (x_0 - h, x_0).$$

Subtracting the second from the first, we get

$$f(x_0 + h) - f(x_0 - h) = 2h f'(x_0) + \frac{h^3}{6} \left(f'''(\xi_1) + f'''(\xi_2) \right).$$

Since f''' is continous, the Intermediate Value Theorem tells us that there exists ξ strictly between ξ_1 and ξ_2 such that

$$f'''(\xi) = \frac{1}{2} \left(f'''(\xi_1) + f'''(\xi_2) \right).$$

Therefore, we have

$$f(x_0 + h) - f(x_0 - h) = 2h f'(x_0) + \frac{h^3}{3} f'''(\xi),$$

and solving for $f'(x_0)$ gives us

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} - \frac{h^2}{6} f'''(\xi).$$

Thus, the **centered** difference formula

$$\frac{f(x_0 + h) - f(x_0 - h)}{2h}$$

gives a second order (i.e., $\mathcal{O}(h^2)$) approximation of $f'(x_0)$:

$$\left| f'(x_0) - \frac{f(x_0 + h) - f(x_0 - h)}{2h} \right| = \frac{h^2}{6} \left| f'''(\xi) \right|, \quad \text{for some } \xi \in (x_0 - h, x_0 + h).$$

- **One-sided three-point approximation for** $f'(x_0)$: Let f be four-times differentiable on an interval containing x_0 and $x_0 + 2h$. From the Taylor series, we have

$$f(x_0 + h) = f(x_0) + h f'(x_0) + \frac{h^2}{2} f''(x_0) + \frac{h^3}{6} f'''(x_0) + \mathcal{O}(h^4),$$
$$f(x_0 + 2h) = f(x_0) + 2h f'(x_0) + 2h^2 f''(x_0) + \frac{4h^3}{3} f'''(x_0) + \mathcal{O}(h^4).$$

Let's take y_1 times the first equation *plus* y_2 times the second equation:

$$y_1 f(x_0+h)+y_2 f(x_0+2h) = (y_1+y_2)f(x_0)+(y_1+2y_2)hf'(x_0)+\left(\frac{1}{2}y_1+2y_2\right)h^2 f''(x_0)+\left(\frac{1}{6}y_1+\frac{4}{3}y_2\right)h^3 f'''(x_0)+$$

To keep the $f'(x_0)$ term and delete the $f''(x_0)$ term, we choose y_1 and y_2 such that

$$\begin{aligned} y_1 + 2y_2 &= 1, \\ \frac{1}{2}y_1 + 2y_2 &= 0. \end{aligned}$$

Thus, we just need to solve the linear system

$$\begin{bmatrix} 1 & 2 \\ \frac{1}{2} & 2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Therefore, $y_1 = 2$ and $y_2 = -\frac{1}{2}$. Thus,

$$2f(x_0+h) - \frac{1}{2}f(x_0+2h) = \frac{3}{2}f(x_0) + hf'(x_0) - \frac{h^3}{3}f'''(x_0) + \mathcal{O}(h^4).$$

Then multiplying both sides by 2, we get

$$4f(x_0+h) - f(x_0+2h) = 3f(x_0) + 2hf'(x_0) - \frac{2h^3}{3}f'''(x_0) + \mathcal{O}(h^4).$$

Solving for $f'(x_0)$, we obtain

$$f'(x_0) = \frac{-3f(x_0)+4f(x_0+h)-f(x_0+2h)}{2h} + \frac{h^2}{3}f'''(x_0) + \mathcal{O}(h^3).$$

Therefore, $y_1 = 2$ and $y_2 = -\frac{1}{2}$. Thus,

$$2f(x_0+h) - \frac{1}{2}f(x_0+2h) = \frac{3}{2}f(x_0) + hf'(x_0) - \frac{h^3}{3}f'''(x_0) + \mathcal{O}(h^4).$$

Then multiplying both sides by 2, we get

$$4f(x_0+h) - f(x_0+2h) = 3f(x_0) + 2hf'(x_0) - \frac{2h^3}{3}f'''(x_0) + \mathcal{O}(h^4).$$

Solving for $f'(x_0)$, we obtain

$$f'(x_0) = \frac{-3f(x_0)+4f(x_0+h)-f(x_0+2h)}{2h} + \frac{h^2}{3}f'''(x_0) + \mathcal{O}(h^3).$$

Thus, the **one-sided three-point** difference formula

$$\frac{-3f(x_0)+4f(x_0+h)-f(x_0+2h)}{2h}$$

gives a second order (i.e., $\mathcal{O}(h^2)$) approximation of $f'(x_0)$:

$$\left| f'(x_0) - \frac{-3f(x_0)+4f(x_0+h)-f(x_0+2h)}{2h} \right| \leqslant \frac{h^2}{3}\left|f'''(x_0)\right| + \mathcal{O}(h^3).$$

Note: It can be shown using a more complicated argument that if f is thrice-continuously-differentiable on an interval containing x_0 and $x_0 + 2h$, then

$$\left| f'(x_0) - \frac{-3f(x_0) + 4f(x_0 + h) - f(x_0 + 2h)}{2h} \right| = \frac{h^2}{3} \left| f'''(\xi) \right|, \quad \text{for some } \xi \in (x_0, x_0 + 2h).$$

- **Centered three-point approximation for $f''(x_0)$**: Let f be four-times continuously-differentiable on an interval containing $x_0 - h$ and $x_0 + h$. From the Taylor series, we have

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0) + \frac{h^3}{6} f'''(x_0) + \frac{h^4}{24} f^{(iv)}(\xi_1), \quad \text{for some } \xi_1 \in (x_0, x_0 + h),$$
$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{h^2}{2} f''(x_0) - \frac{h^3}{6} f'''(x_0) + \frac{h^4}{24} f^{(iv)}(\xi_2), \quad \text{for some } \xi_2 \in (x_0 - h, x_0).$$

This time we add the Taylor expansions to obtain

$$f(x_0 + h) + f(x_0 - h) = 2f(x_0) + h^2 f''(x_0) + \frac{h^4}{24} \left(f^{(iv)}(\xi_1) + f^{(iv)}(\xi_2) \right)$$

Since $f^{(iv)}$ is continous, the Intermediate Value Theorem tells us that there exists ξ strictly between ξ_1 and ξ_2 such that

$$f^{(iv)}(\xi) = \frac{1}{2} \left(f^{(iv)}(\xi_1) + f^{(iv)}(\xi_2) \right).$$

Therefore, we have

$$f(x_0 + h) + f(x_0 - h) = 2f(x_0) + h^2 f''(x_0) + \frac{h^4}{12} f^{(iv)}(\xi),$$

and solving for $f''(x_0)$ gives us

$$f''(x_0) = \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2} - \frac{h^2}{12} f^{(iv)}(\xi).$$

Thus, the **centered** difference formula

$$\frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2}$$

gives a second order (i.e., $\mathcal{O}(h^2)$) approximation of $f''(x_0)$:

$$\left| f''(x_0) - \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2} \right| = \frac{h^2}{12} \left| f^{(iv)}(\xi) \right|, \quad \text{for some } \xi \in (x_0 - h, x_0 + h).$$

- **Five-point formulas: Centered five-point approximation for $f'(x_0)$**: Let f be seven-times differentiable on an interval containing $x_0 - 2h$ and $x_0 + 2h$. From the Taylor series, we have

$$\begin{aligned}
f(x_0 + h) &= k_0 + hk_1 + h^2 k_2 + h^3 k_3 + h^4 k_4 + h^5 k_5 + h^6 k_6 + \mathcal{O}(h^7), \\
f(x_0 - h) &= k_0 - hk_1 + h^2 k_2 - h^3 k_3 + h^4 k_4 - h^5 k_5 + h^6 k_6 + \mathcal{O}(h^7), \\
f(x_0 + 2h) &= k_0 + 2hk_1 + 4h^2 k_2 + 8h^3 k_3 + 16h^4 k_4 + 32h^5 k_5 + 64h^6 k_6 + \mathcal{O}(h^7), \\
f(x_0 - 2h) &= k_0 - 2hk_1 + 4h^2 k_2 - 8h^3 k_3 + 16h^4 k_4 - 32h^5 k_5 + 64h^6 k_6 + \mathcal{O}(h^7),
\end{aligned}$$

where $k_n = \frac{f^{(n)}(x_0)}{n!}$, for $n = 0, \ldots, 6$.

We multiple the above equations respectively by y_1, y_2, y_3, y_4 and add them to obtain

$$y_1 f(x_0+h)+y_2 f(x_0-h)+y_3 f(x_0+2h)+y_4 f(x_0-2h) = (y_1+y_2+y_3+y_4)k_0+(y_1-y_2+2y_3-2y_4)hk_1+(y_1+y_2+4y_3+$$

We can specify four conditions to solve for y_1, y_2, y_3, y_4. We want to keep the $f'(x_0)$ term, and get the highest accuracy possible. Thus, we want

$$\begin{aligned} y_1 - y_2 + 2y_3 - 2y_4 &= 1, \\ y_1 + y_2 + 4y_3 + 4y_4 &= 0, \\ y_1 - y_2 + 8y_3 - 8y_4 &= 0, \\ y_1 + y_2 + 16y_3 + 16y_4 &= 0. \end{aligned}$$

Thus, we have

$$8f(x_0+h)-8f(x_0-h)-f(x_0+2h)+f(x_0-2h) = 12hf'(x_0)-48h^5\frac{f^{(v)}(x_0)}{120}+\mathcal{O}(h^7).$$

Solving for $f'(x_0)$, we obtain:

$$f'(x_0) = \frac{f(x_0-2h)-8f(x_0-h)+8f(x_0+h)-f(x_0+2h)}{12h}+\frac{h^4}{30}f^{(v)}(x_0)+\mathcal{O}(h^6)$$

Thus, the **centered five-point** difference formula

$$\frac{f(x_0-2h)-8f(x_0-h)+8f(x_0+h)-f(x_0+2h)}{12h}$$

gives a fourth order (i.e., $\mathcal{O}(h^4)$) approximation of $f'(x_0)$:

$$\left|f'(x_0)-\frac{f(x_0-2h)-8f(x_0-h)+8f(x_0+h)-f(x_0+2h)}{12h}\right| \leqslant \frac{h^4}{30}\left|f^{(v)}(x_0)\right|+\mathcal{O}(h^6).$$

Note: It can be shown using a more complicated argument that if f is five-times-continuously-differentiable on an interval containing x_0-2h and x_0+2h, then

$$\left|f'(x_0)-\frac{f(x_0-2h)-8f(x_0-h)+8f(x_0+h)-f(x_0+2h)}{12h}\right| = \frac{h^4}{30}\left|f^{(v)}(\xi)\right|,$$

for some $\xi \in (x_0-2h, x_0+2h)$.

- **Richardson extrapolation**: **Richardson extrapolation** is a simple technique for generating higher order numerical methods from lower order methods.

For example, let's look at our **first-order accurate** method for approximating $f'(x_0)$. Letting

$$S(h) = \frac{f(x_0+h)-f(x_0)}{h}, \quad C = -\frac{f''(x_0)}{2}, \quad \text{and} \quad D = -\frac{f'''(x_0)}{6},$$

we have

$$f'(x_0) - S(h) = Ch + Dh^2 + \mathcal{O}(h^3).$$

Let

$$e(h) = f'(x_0) - S(h)$$

be the **approximation error**.

If we could compute the exact value of $e(h)$, then we could compute the exact value of

$$f'(x_0) = S(h) + e(h).$$

Instead of getting the exact value of $e(h)$, we can use the fact that

$$e(h) = Ch + Dh^2 + \mathcal{O}(h^3)$$

to get an **approximation** of $e(h)$.

We can approximate $e(h)$ by computing $S(h)$ and $S(h/2)$. Notice that

$$f'(x_0) - S(h) = Ch + Dh^2 + \mathcal{O}(h^3), \tag{1}$$

$$f'(x_0) - S(h/2) = \frac{C}{2}h + \frac{D}{4}h^2 + \mathcal{O}(h^3). \tag{2}$$

$$\tag{3}$$

Subtracting these equations the unknown $f'(x_0)$ terms cancel, and we get

$$S(h/2) - S(h) = \frac{C}{2}h + \frac{3}{4}Dh^2 + \mathcal{O}(h^3),$$

which implies that

$$2(S(h/2) - S(h)) = Ch + \frac{3}{2}Dh^2 + \mathcal{O}(h^3).$$

Since $e(h) = Ch + Dh^2 + \mathcal{O}(h^3)$, we have that

$$2(S(h/2) - S(h)) - e(h) = \frac{D}{2}h^2 + \mathcal{O}(h^3)$$

which implies that

$$e(h) = 2(S(h/2) - S(h)) - \frac{D}{2}h^2 + \mathcal{O}(h^3).$$

Therefore,

$$e(h) \approx 2(S(h/2) - S(h)).$$

We can now use our error approximation to improve the approximation of $f'(x_0)$.

Since $f'(x_0) = S(h) + e(h)$, we have that

$$f'(x_0) = \boxed{2S(h/2) - S(h)} - \frac{D}{2}h^2 + \mathcal{O}(h^3),$$

which gives us a **second-order accurate** method from a **first-order accurate** method!

- **Simplifying the approximation**: Computing $S(h)$ and $S(h/2)$ will evaluate $f(x_0)$ twice. We can avoid this by simplifying the approximation formula:

$$2S(h/2) - S(h) = 2\frac{f(x_0 + h/2) - f(x_0)}{h/2} - \frac{f(x_0 + h) - f(x_0)}{h} \tag{4}$$

$$= \frac{4f(x_0 + h/2) - 4f(x_0) - f(x_0 + h) + f(x_0)}{h} \tag{5}$$

$$= \frac{-3f(x_0) + 4f(x_0 + h/2) - f(x_0 + h)}{h}. \tag{6}$$

$$\tag{7}$$

Replacing $h/2$ with h, we see that this is the **one-sided three-point** difference formula,

$$\frac{-3f(x_0) + 4f(x_0 + h) - f(x_0 + 2h)}{2h}.$$

- **From second-order to third-order**: Suppose we have a **second-order** numerical method, $S(h)$, for approximating $\bar{x}$:

  $$\bar{x} = S(h) + Ch^2 + \mathcal{O}(h^3).$$

  Then,

  $$\bar{x} = S(h/2) + \frac{C}{4}h^2 + \mathcal{O}(h^3).$$

  We can cancel the h^2 terms by multiplying the second equation by **four** and subtracting the first equation, which gives us

  $$3\bar{x} = 4S(h/2) - S(h) + \mathcal{O}(h^3).$$

  Thus,

  $$\bar{x} = \boxed{\frac{4S(h/2) - S(h)}{3}} + \mathcal{O}(h^3),$$

  giving us a **third-order** numerical method for approximating $\bar{x}$.

- **The general technique**: In general, **Richardson extrapolation** can be used to obtain an $(n+1)^{\text{st}}$-order method from an n^{th}-order method.

  Suppose that our n^{th}-order method for approximating $\bar{x}$ is given by

  $$\bar{x} = S(h) + Ch^n + \mathcal{O}(h^{n+1}).$$

  Substituting h with $h/2$, we have

  $$\bar{x} = S(h/2) + \frac{C}{2^n}h^n + \mathcal{O}(h^{n+1}).$$

  We then multiply the second equation by 2^n and subtract the first equation to cancel the h^n terms:

  $$(2^n - 1)\bar{x} = 2^n S(h/2) - S(h) + \mathcal{O}(h^{n+1}).$$

  This gives us

  $$\bar{x} = \boxed{\frac{2^n S(h/2) - S(h)}{2^n - 1}} + \mathcal{O}(h^{n+1}),$$

  which is an $(n+1)^{\text{st}}$-order method for approximating $\bar{x}$.

- **Deriving formulas using Lagrange polynomial interpolation**: we will approximate the derivative of a function f at a point x_0 using the following simple recipe:

  1. Compute the value of the function at a few nearby points.
  2. Interpolate the points with a polynomial.
  3. Use the derivative(s) of the polynomial at x_0 to approximate the derivative(s) of f at x_0.

Remark. *Theorem: (Polynomial Interpolation Error)*: Suppose $x_0, x_1, \ldots, x_n \in [a, b]$ are distinct and $f \in C^{n+1}[a, b]$. Then, for each $x \in [a, b]$, there is a $\xi(x) \in (a, b)$ such that

$$f(x) = p_n(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!}\prod_{i=0}^{n}(x - x_i),$$

where p_n is the polynomial interpolating the points $(x_0, f(x_0)), \ldots, (x_n, f(x_n))$.

Given in Lagrange form, we have

$$p_n(x) = f(x_0)L_0(x) + \cdots + f(x_n)L_n(x),$$

where

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

- **Deriving the one-sided three-point approximation for** $f'(x_0)$: We will use the following points

$$x_0, \quad x_1 = x_0 + h, \quad x_2 = x_0 + 2h.$$

By the **Polynomial Interpolation Error Theorem** we have that

$$f(x) = p_2(x) + \frac{f^{(3)}(\xi(x))}{3!}(x - x_0)(x - x_1)(x - x_2),$$

which implies that

$$\begin{aligned} f'(x) = p_2'(x) &+ \frac{d}{dx}\frac{f^{(3)}(\xi(x))}{3!}(x - x_0)(x - x_1)(x - x_2) \\ &+ \frac{f^{(3)}(\xi(x))}{3!}\big((x - x_1)(x - x_2) + (x - x_0)(x - x_2) + (x - x_0)(x - x_1)\big).. \end{aligned}$$

Now we just substitute $x = x_0$ to obtain

$$f'(x_0) = p_2'(x_0) + \frac{f^{(3)}(\xi)}{3!}(x_0 - x_1)(x_0 - x_2),$$

for some $\xi \in (x_0, x_0 + 2h)$.

Note that

$$p_2(x) = f(x_0)\frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + f(x_1)\frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + f(x_2)\frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}.$$

Thus,

$$p_2'(x) = f(x_0)\frac{(x - x_1) + (x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + f(x_1)\frac{(x - x_0) + (x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + f(x_2)\frac{(x - x_0) + (x - x_1)}{(x_2 - x_0)(x_2 - x_1)},$$

and

$$p_2'(x_0) = f(x_0)\frac{(x_0 - x_1) + (x_0 - x_2)}{(x_0 - x_1)(x_0 - x_2)} + f(x_1)\frac{(x_0 - x_2)}{(x_1 - x_0)(x_1 - x_2)} + f(x_2)\frac{(x_0 - x_1)}{(x_2 - x_0)(x_2 - x_1)}.$$

Using the fact that $x_1 = x_0 + h$ and $x_2 = x_0 + 2h$, we can simplify this as

$$p_2'(x_0) = f(x_0)\frac{-h - 2h}{(-h)(-2h)} + f(x_0 + h)\frac{-2h}{(h)(-h)} + f(x_0 + 2h)\frac{-h}{(2h)(h)}.$$

Therefore,

$$p_2'(x_0) = -f(x_0)\frac{3}{2h} + f(x_0 + h)\frac{2}{h} - f(x_0 + 2h)\frac{1}{2h},$$

which gives us the familiar formula

$$p_2'(x_0) = \frac{-3f(x_0) + 4f(x_0 + h) - f(x_0 + 2h)}{2h}.$$

Thus,

$$\begin{aligned} f'(x_0) &= p_2'(x_0) + \frac{f^{(3)}(\xi)}{3!}(x_0 - x_1)(x_0 - x_2) \\ &= \frac{-3f(x_0) + 4f(x_0 + h) - f(x_0 + 2h)}{2h} + \frac{f^{(3)}(\xi)}{3!}(-h)(-2h) \\ &= \frac{-3f(x_0) + 4f(x_0 + h) - f(x_0 + 2h)}{2h} + \frac{h^2}{3}f^{(3)}(\xi). \end{aligned}$$

.

Summarizing,

$$\boxed{f'(x_0) = \frac{-3f(x_0) + 4f(x_0 + h) - f(x_0 + 2h)}{2h} + \frac{h^2}{3}f^{(3)}(\xi), \quad \text{for some } \xi \in (x_0, x_0 + 2h).}$$

This is the same formula we obtained in Section 14.1, but this time we have a rigorous proof of the existence of ξ.

- **Nonuniformly spaced points**: Suppose we would like to approximate $f'(x_0)$ using the values of f at the three points:

$$x_{-1} = x_0 - h_0, \qquad x_0, \qquad x_1 = x_0 + h_1,$$

where $h_0 \neq h_1$. You may guess that we should use the formula

$$\frac{f(x_0 + h_1) - f(x_0 - h_0)}{h_0 + h_1},$$

but this turns out to be **incorrect**!

Using polynomial interpolation, we can obtain a better approximation of $f'(x_0)$.

Let $p_2(x)$ be the polynomial that interpolates $(x_{-1}, f(x_{-1}))$, $(x_0, f(x_0))$, and $(x_1, f(x_1))$. Then

$$p_2(x) = f(x_{-1})\frac{(x - x_0)(x - x_1)}{(x_{-1} - x_0)(x_{-1} - x_1)} + f(x_0)\frac{(x - x_{-1})(x - x_1)}{(x_0 - x_{-1})(x_0 - x_1)} + f(x_1)\frac{(x - x_{-1})(x - x_0)}{(x_1 - x_{-1})(x_1 - x_0)},$$

so we have

$$p_2'(x) = f(x_{-1})\frac{(x - x_0) + (x - x_1)}{(x_{-1} - x_0)(x_{-1} - x_1)} + f(x_0)\frac{(x - x_{-1}) + (x - x_1)}{(x_0 - x_{-1})(x_0 - x_1)} + f(x_1)\frac{(x - x_{-1}) + (x - x_0)}{(x_1 - x_{-1})(x_1 - x_0)}.$$

Thus,

$$p_2'(x_0) = f(x_{-1})\frac{x_0 - x_1}{(x_{-1} - x_0)(x_{-1} - x_1)} + f(x_0)\frac{(x_0 - x_{-1}) + (x_0 - x_1)}{(x_0 - x_{-1})(x_0 - x_1)} + f(x_1)\frac{x_0 - x_{-1}}{(x_1 - x_{-1})(x_1 - x_0)},$$

and so

$$p_2'(x_0) = f(x_{-1})\frac{-h_1}{(-h_0)(-h_0 - h_1)} + f(x_0)\frac{h_0 - h_1}{(h_0)(-h_1)} + f(x_1)\frac{h_0}{(h_0 + h_1)(h_1)},$$

which simplifies to

$$p_2'(x_0) = \frac{h_1 - h_0}{h_0 h_1}f(x_0) + \frac{1}{h_0 + h_1}\left(\frac{h_0}{h_1}f(x_1) - \frac{h_1}{h_0}f(x_{-1})\right).$$

The **Polynomial Interpolation Error Theorem** tells us that

$$f(x) = p_2(x) + \frac{f^{(3)}(\xi(x))}{3!}(x - x_{-1})(x - x_0)(x - x_1),$$

and so

$$f'(x_0) = p_2'(x_0) + \frac{f^{(3)}(\xi)}{3!}(x_0 - x_{-1})(x_0 - x_1),$$

for some $\xi \in (x_0 - h_0, x_0 + h_1)$.

Therefore,

$$\boxed{f'(x_0) = \frac{h_1 - h_0}{h_0 h_1} f(x_0) + \frac{1}{h_0 + h_1}\left(\frac{h_0}{h_1} f(x_1) - \frac{h_1}{h_0} f(x_{-1})\right) - \frac{f^{(3)}(\xi)}{6} h_0 h_1,}$$

for some $\xi \in (x_0 - h_0, x_0 + h_1)$.

- **Roundoff error in numerical differentiation**: In the examples above we have seen that the actual error degrades when h becomes too small due to roundoff error.

  This is happening since, in each of the formulas we have looked at, we are subtracting values that are very nearly equal, leading to **catastrophic cancellation** and a significant loss of precision.

  The error for the **centered divided difference** satisfies:

  $$\left| f'(x_0) - \frac{f(x_0 + h) - f(x_0 - h)}{2h} \right| = \frac{h^2}{6} \left| f'''(\xi) \right|, \quad \text{for some } \xi \in (x_0 - h, x_0 + h).$$

  Thus, when $f'''(x_0) \neq 0$, we have that

  $$\left| f'(x_0) - \frac{f(x_0 + h) - f(x_0 - h)}{2h} \right| \approx \frac{h^2}{6} \left| f'''(x_0) \right|, \quad \text{for } h > 0 \text{ small.}$$

- **Analyzing the roundoff error**: Let $\bar{f}(x) \equiv \text{fl}(f(x))$, and define

  $$D_h = \frac{f(x_0 + h) - f(x_0 - h)}{2h},$$
  $$\bar{D}_h = \frac{\bar{f}(x_0 + h) - \bar{f}(x_0 - h)}{2h}..$$

  Define the **roundoff error in the calculation of** f as

  $$e_r(x) = \bar{f}(x) - f(x)$$

  and suppose that $|e_r(x)| \leqslant \varepsilon$, where ε is some small multiple of the **unit-roundoff** η (recall that $\eta \approx 1.1 \times 10^{-16}$ for 'Float64').

  Also, suppose that $|f'''(\xi)| \leqslant M$, for all $\xi \in (x_0 - h, x_0 + h)$. Thus,

  $$|f'(x_0) - D_h| = \frac{h^2}{6} |f'''(\xi)| \leqslant \frac{h^2 M}{6}.$$

  Then

  $$\begin{aligned} \left| f'(x_0) - \bar{D}_h \right| &= \left| f'(x_0) - D_h + D_h - \bar{D}_h \right| \\ &\leqslant |f'(x_0) - D_h| + \left| D_h - \bar{D}_h \right| \\ &\leqslant \frac{h^2 M}{6} + \left| D_h - \bar{D}_h \right|. \end{aligned}$$

  .

Now let's bound $\left|D_h - \bar{D}_h\right|$. We have

$$\begin{aligned}
\left|D_h - \bar{D}_h\right| &= \left|\frac{f(x_0+h) - f(x_0-h)}{2h} - \frac{\bar{f}(x_0+h) - \bar{f}(x_0-h)}{2h}\right| \\
&= \frac{1}{2h}\left|f(x_0+h) - f(x_0-h) - \bar{f}(x_0+h) + \bar{f}(x_0-h)\right| \\
&= \frac{1}{2h}\left|e_r(x_0-h) - e_r(x_0+h)\right| \\
&\leqslant \frac{1}{2h}\left(|e_r(x_0-h)| + |e_r(x_0+h)|\right) \\
&\leqslant \frac{1}{2h} 2\varepsilon \\
&\leqslant \frac{\varepsilon}{h}.
\end{aligned}$$

.

Therefore,

$$\left|f'(x_0) - \bar{D}_h\right| \leqslant \frac{h^2 M}{6} + \frac{\varepsilon}{h}.$$

Initially, when h is rather large, the roundoff error term $\frac{\varepsilon}{h}$ will be much smaller than the approximation error (a.k.a. **truncation error**) term $\frac{h^2 M}{6}$.

However, as h becomes small, $\frac{\varepsilon}{h}$ eventually becomes larger than $\frac{h^2 M}{6}$.

- **The "optimal" h for the centered divided difference formula**: Define

$$E(h) = \frac{h^2 M}{6} + \frac{\varepsilon}{h}.$$

We want to find the minimum value of $E(h)$. First we solve $E'(h) = 0$ to find the critical points:

$$E'(h) = \frac{hM}{3} - \frac{\varepsilon}{h^2} = 0 \quad \Longrightarrow \quad h^3 = \frac{3\varepsilon}{M} \quad \Longrightarrow \quad h = \sqrt[3]{\frac{3\varepsilon}{M}}.$$

Next, we check if this critical point is a local minimizer or maximizer:

$$E''(h) = \frac{M}{3} + \frac{2\varepsilon}{h^3} \quad \Longrightarrow \quad E''\left(\sqrt[3]{\frac{3\varepsilon}{M}}\right) = \frac{M}{3} + \frac{2\varepsilon}{\frac{3\varepsilon}{M}} = M > 0.$$

Thus, the critical point $h = \sqrt[3]{\frac{3\varepsilon}{M}}$ is a local minimizer, and the local minimum is

$$E\left(\sqrt[3]{\frac{3\varepsilon}{M}}\right) = \frac{1}{2}\sqrt[3]{M}(3\varepsilon)^{\frac{2}{3}}.$$

- **Advice on choosing h**: Since the values of ε and M are typically unknown, it is usually not possible to know the optimal value of h.

Thus, we should choose a value of h that is well above the point where roundoff error begins to corrupt our calculation.

That is, we should choose h so that the truncation error is much larger than the roundoff error.

In the above example, using h between 10^{-5} and 10^{-4} would be best.

In general, if the order of accuracy is q, then choose

$$h > \eta^{1/(q+1)}.$$

- **Numerical differentiation of noisy data**: Numerical differentiation is very sensitive to small changes in the input since it is an **ill-conditioned** problem.

  In the following numerical test, we will see that just a small amount of noise in the function values can create large errors in the approximation of its derivative.

1.8 Numerical integration

- **Basic quadrature algorithms**: The goal of this chapter is to determine quadrature formulas for approximating

$$I_f = \int_a^b f(x)\,dx \approx \sum_{j=0}^{n} a_j f(x_j).$$

  We just need to determine the **quadrature weights** a_j and **points** x_j.

- **A simple approach**:

  1. Compute the value of the function f at a few points $x_0, \ldots, x_n \in [a, b]$.
  2. Interpolate $(x_0, f(x_0)), \ldots, (x_n, f(x_n))$ with a polynomial p_n written in Lagrange form:
$$p_n(x) = \sum_{j=0}^{n} f(x_j) L_j(x).$$
  3. Then the integral of the polynomial p_n,
$$\int_a^b p_n(x)\,dx = \sum_{j=0}^{n} a_j f(x_j), \quad a_j = \int_a^b L_j(x)\,dx,$$
  approximates the integral of f.

- **Quadrature error**: Recall that

$$f(x) - p_n(x) = f[x_0, \ldots, x_n, x] \prod_{j=0}^{n} (x - x_j).$$

  Therefore, the **quadrature error** is

$$\begin{aligned} E(f) &= \int_a^b f(x)\,dx - \sum_{j=0}^{n} a_j f(x_j) \\ &= \int_a^b f(x)\,dx - \int_a^b p_n(x)\,dx \\ &= \int_a^b \big(f(x) - p_n(x)\big)\,dx \\ &= \int_a^b f[x_0, \ldots, x_n, x] \prod_{j=0}^{n} (x - x_j)\,dx. \end{aligned}$$

- **Trapezoidal rule**: Let $x_0 = a$ and $x_1 = b$. Then

$$L_0(x) = \frac{x-b}{a-b}, \quad L_1(x) = \frac{x-a}{b-a}.$$

  Thus,

$$a_0 = \int_a^b L_0(x)\,dx = \frac{b-a}{2}, \quad a_1 = \int_a^b L_1(x)\,dx = \frac{b-a}{2}.$$

  Therefore, the **trapezoidal rule** is

$$\begin{aligned} I_f \approx \sum_{j=0}^{1} a_j f(x_j) &= \frac{b-a}{2} f(a) + \frac{b-a}{2} f(b) \\ &= \frac{b-a}{2} \big[f(a) + f(b)\big] =: I_{\text{trap}}. \end{aligned}$$

- **Theorem: (Mean Value Theorem for Integrals)**: If $g \in C[a,b]$ and ψ is an integrable function such that $\psi(x) \geqslant 0$ for all $x \in [a,b]$ or $\psi(x) \leqslant 0$ for all $x \in [a,b]$. Then there exists $\xi \in [a,b]$ such that

$$\int_a^b g(x)\psi(x)\,dx = g(\xi)\int_a^b \psi(x)\,dx.$$

Proof. First observe that there exist $\alpha, \beta \in [a,b]$ such that $g(\alpha) = \min_{x\in[a,b]} g(x)$ and $g(\beta) = \max_{x\in[a,b]} g(x)$. Then

$$g(\alpha) \leqslant g(x) \leqslant g(\beta), \quad \forall x \in [a,b].$$

Suppose that $\psi(x) \geqslant 0$ for all $x \in [a,b]$. Then

$$g(\alpha)\psi(x) \leqslant g(x)\psi(x) \leqslant g(\beta)\psi(x), \quad \forall x \in [a,b].$$

Thus,

$$\int_a^b g(\alpha)\psi(x)\,dx \leqslant \int_a^b g(x)\psi(x)\,dx \leqslant \int_a^b g(\beta)\psi(x)\,dx.$$

Letting $s = \int_a^b g(x)\psi(x)\,dx$, we have

$$g(\alpha)\int_a^b \psi(x)\,dx \leqslant s \leqslant g(\beta)\int_a^b \psi(x)\,dx.$$

Let $f(x) = g(x)\int_a^b \psi(x)\,dx$. Then $f \in C[a,b]$ and $f(\alpha) \leqslant s \leqslant f(\beta)$, so by the Intermediate Value Theorem, there is a $\xi \in [a,b]$ such that $f(\xi) = s$; that is,

$$g(\xi)\int_a^b \psi(x)\,dx = \int_a^b g(x)\psi(x)\,dx.$$

The other case of $\psi(x) \leqslant 0$ for all $x \in [a,b]$ can be proven in a similar way. ■

- **Trapezoidal rule error**: Recall that the error is

$$E(f) = \int_a^b f[x_0,\ldots,x_n,x]\prod_{j=0}^{n}(x-x_j)\,dx.$$

For the Trapezoidal rule, we have

$$E(f) = \int_a^b f[a,b,x](x-a)(x-b)\,dx.$$

Let $\psi(x) = (x-a)(x-b)$. Then $\psi(x) \leqslant 0$ for all $x \in [a,b]$. Therefore, by the **Mean Value Theorem for Integrals** we have

$$E(f) = f[a,b,\xi_1]\int_a^b (x-a)(x-b)\,dx.$$

By the **Divided Difference and Derivative Theorem** from Section 10.5, there is a $\xi \in (a,b)$ such that $f[a,b,\xi_1] = \frac{f''(\xi)}{2}$. Also,

$$\int_a^b (x-a)(x-b)\,dx = -\frac{(b-a)^3}{6}.$$

Therefore,

$$E(f) = I_f - I_{\text{trap}} = -\frac{f''(\xi)}{12}(b-a)^3,$$

for some $\xi \in (a,b)$.

- **Midpoint rule**: Let $x_0 = \frac{a+b}{2}$. Then

$$L_0(x) = 1.$$

Thus,

$$a_0 = \int_a^b L_0(x)\,dx = b - a,$$

Therefore, the **midpoint rule** is

$$I_f \approx \sum_{j=0}^{0} a_j f(x_j) = (b-a) f\left(\frac{a+b}{2}\right) =: I_{\text{mid}}.$$

The error is given by

$$E(f) = I_f - I_{\text{mid}} = \frac{f''(\xi)}{24}(b-a)^3,$$

for some $\xi \in (a, b)$.

- **Simpson rule**: Let $x_0 = a$, $x_1 = x_m$, and $x_2 = b$, where $x_m = \frac{a+b}{2}$. Then

$$L_0(x) = \frac{(x-x_m)(x-b)}{(a-x_m)(a-b)}, \quad L_1(x) = \frac{(x-a)(x-b)}{(x_m-a)(x_m-b)}, \quad L_2(x) = \frac{(x-a)(x-x_m)}{(b-a)(b-x_m)}.$$

Thus,

$$a_0 = \int_a^b L_0(x)\,dx = \frac{1}{6}(b-a), \quad a_1 = \int_a^b L_1(x)\,dx = \frac{2}{3}(b-a), \quad a_2 = \int_a^b L_2(x)\,dx = \frac{1}{6}(b-a).$$

Therefore, the **Simpson rule** is

$$\begin{aligned} I_f \approx \sum_{j=0}^{2} a_j f(x_j) &= \frac{1}{6}(b-a)f(a) + \frac{2}{3}(b-a)f(x_m) + \frac{1}{6}(b-a)f(b) \\ &= \frac{b-a}{6}\left[f(a) + 4f\left(\frac{b+a}{2}\right) + f(b)\right] =: I_{\text{Simp}}. \end{aligned}$$

The error is given by

$$E(f) = I_f - I_{\text{Simp}} = -\frac{f''''(\xi)}{90}\left(\frac{b-a}{2}\right)^5,$$

for some $\xi \in (a, b)$.

- **Quadrature rules**:

$$\int_a^b f(x)\,dx = (b-a)\, f\left(\frac{a+b}{2}\right) + \frac{f''(\xi)}{24}(b-a)^3 \qquad \text{(Midpoint)}$$

$$\int_a^b f(x)\,dx = \frac{b-a}{2}\left[f(a) + f(b)\right] - \frac{f''(\xi)}{12}(b-a)^3 \qquad \text{(Trapezoidal)}$$

$$\int_a^b f(x)\,dx = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] - \frac{f''''(\xi)}{90}\left(\frac{b-a}{2}\right)^5 \qquad \text{(Simpson)}$$

In each case, ξ is some unknown point in the interval (a, b).

- **Precision**: If f is a polynomial of degree at most **one**, then the **midpoint and trapezoidal rules are exact**, so we say that these rules have **precision 1**.

  If f is a polynomial of degree at most **three**, then the **Simpson rule is exact**, so we say that the Simpson rule has **precision 3**.

- **Composite quadrature methods**: We can obtain improved approximations of $\int_a^b f(x)\,dx$ by dividing the interval $[a,b]$ into r equal subintervals:

$$[t_0,t_1], \quad [t_1,t_2], \quad \ldots, \quad [t_{r-1},t_r],$$

  where $t_0 = a$ and $t_r = b$. Then

$$\int_a^b f(x)\,dx = \sum_{i=1}^{r} \int_{t_{i-1}}^{t_i} f(x)\,dx.$$

  The length of each subinterval is

$$h = \frac{b-a}{r},$$

  and $t_i = a + ih$, for $i = 0, 1, \ldots, r$.

- **Composite midpoint rule**: From the midpoint rule for the i^{th} interval $[t_{i-1}, t_i]$, we have

$$\begin{aligned}\int_{t_{i-1}}^{t_i} f(x)\,dx &= (t_i - t_{i-1})\, f\left(\frac{t_{i-1}+t_i}{2}\right) + \frac{f''(\xi_i)}{24}(t_i - t_{i-1})^3 \\ &= hf(a + (i-1/2)h) + \frac{f''(\xi_i)}{24}h^3,\end{aligned}$$

  .

  for some $\xi_i \in [t_{i-1}, t_i]$. Thus, we have:

$$\begin{aligned}\int_a^b f(x)\,dx &= \sum_{i=1}^{r} \int_{t_{i-1}}^{t_i} f(x)\,dx \\ &= \sum_{i=1}^{r} \left[hf(a + (i-1/2)h) + \frac{f''(\xi_i)}{24}h^3 \right] \\ &= h\sum_{i=1}^{r} f(a + (i-1/2)h) + \left(\frac{1}{r}\sum_{i=1}^{r} f''(\xi_i)\right)\frac{rh^3}{24}.\end{aligned}$$

  .

  By the **Intermediate Value Theorem**, there is a $\xi \in [a,b]$ such that

$$f''(\xi) = \frac{1}{r}\sum_{i=1}^{r} f''(\xi_i).$$

  Therefore, we obtain the **composite midpoint rule**:

$$\boxed{\int_a^b f(x)\,dx = h\sum_{i=1}^{r} f(a + (i-1/2)h) + \frac{f''(\xi)}{24}(b-a)h^2, \quad \xi \in [a,b],}$$

  which has **order two accuracy**:

$$\boxed{\left| \int_a^b f(x)\,dx - h\sum_{i=1}^{r} f(a + (i-1/2)h) \right| \leqslant \frac{\|f''\|}{24}(b-a)h^2 = \mathcal{O}(h^2).}$$

- **Composite trapezoidal rule**: From the trapezoidal rule for the i^{th} interval $[t_{i-1}, t_i]$, we have

$$\begin{aligned}\int_{t_{i-1}}^{t_i} f(x)\,dx &= \frac{t_i - t_{i-1}}{2}\left[f(t_{i-1}) + f(t_i)\right] - \frac{f''(\xi_i)}{12}(t_i - t_{i-1})^3 \\ &= \frac{h}{2}\left[f(t_{i-1}) + f(t_i)\right] - \frac{f''(\xi_i)}{12}h^3\end{aligned}$$

.

for some $\xi_i \in [t_{i-1}, t_i]$.

Thus, we have:

$$\begin{aligned}\int_a^b f(x)\,dx &= \sum_{i=1}^{r}\int_{t_{i-1}}^{t_i} f(x)\,dx \\ &= \sum_{i=1}^{r}\left[\frac{h}{2}\left[f(t_{i-1}) + f(t_i)\right] - \frac{f''(\xi_i)}{12}h^3\right] \\ &= \frac{h}{2}\left[f(t_0) + 2f(t_1) + \cdots + 2f(t_{r-1}) + f(t_r)\right] \\ &\quad - \frac{1}{r}\left(\sum_{i=1}^{r} f''(\xi_i)\right)\frac{rh^3}{12}.\end{aligned}$$

.

Again, by the **Intermediate Value Theorem**, there is a $\xi \in [a, b]$ such that

$$f''(\xi) = \frac{1}{r}\sum_{i=1}^{r} f''(\xi_i).$$

Therefore, we obtain the **composite trapezoidal rule**:

$$\boxed{\int_a^b f(x)\,dx = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{r-1} f(a+ih) + f(b)\right] - \frac{f''(\xi)}{12}(b-a)h^2, \quad \xi \in [a, b],}$$

which has **order two accuracy**:

$$\boxed{\left|\int_a^b f(x)\,dx - \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{r-1} f(a+ih) + f(b)\right]\right| \leqslant \frac{\|f''\|}{12}(b-a)h^2 = \mathcal{O}(h^2).}$$

- **Composite Simpson rule**: The Simpson rule for the k^{th} *pair* of intervals $[t_{2k-2}, t_{2k}]$ gives us

$$\begin{aligned}\int_{t_{2k-2}}^{t_{2k}} f(x)\,dx &= \frac{t_{2k} - t_{2k-2}}{6}\left[f(t_{2k-2}) + 4f\left(\frac{t_{2k-2} + t_{2k}}{2}\right) + f(t_{2k})\right] - \frac{f''''(\xi_k)}{90}\left(\frac{t_{2k} - t_{2k-2}}{2}\right)^5 \\ &= \frac{h}{3}\left[f(t_{2k-2}) + 4f(t_{2k-1}) + f(t_{2k})\right] - \frac{f''''(\xi_k)}{90}h^5\end{aligned}$$

.

for some $\xi_k \in [t_{2k-2}, t_{2k}]$.

Thus, we have (r must be even):

$$\begin{aligned}\int_a^b f(x)\,dx &= \sum_{k=1}^{r/2} \int_{t_{2k-2}}^{t_{2k}} f(x)\,dx \\ &= \sum_{k=1}^{r/2} \left[\frac{h}{3}\left[f(t_{2k-2}) + 4f(t_{2k-1}) + f(t_{2k})\right] - \frac{f''''(\xi_k)}{90}h^5\right] \\ &= \frac{h}{3}\left[f(t_0) + 4f(t_1) + 2f(t_2) + \cdots + 2f(t_{r-2}) + 4f(t_{r-1}) + f(t_r)\right] \\ &\quad - \frac{1}{r/2}\left(\sum_{k=1}^{r/2} f''''(\xi_k)\right)\frac{h^5}{90}\frac{r}{2}.\end{aligned}$$

.

Once more, by the **Intermediate Value Theorem**, there is a $\xi \in [a, b]$ such that

$$f''''(\xi) = \frac{1}{r/2}\sum_{i=1}^{r/2} f''''(\xi_i).$$

Therefore, we obtain the **composite Simpson rule**:

$$\boxed{\int_a^b f(x)\,dx = \frac{h}{3}\left[f(a) + 4\sum_{k=1}^{r/2} f(t_{2k-1}) + 2\sum_{k=1}^{r/2-1} f(t_{2k}) + f(b)\right] - \frac{f''''(\xi)}{180}(b-a)h^4}$$

for some $\xi \in [a, b]$, which has **order four accuracy**:

$$\boxed{\left|\int_a^b f(x)\,dx - \frac{h}{3}\left[f(a) + 4\sum_{k=1}^{r/2} f(t_{2k-1}) + 2\sum_{k=1}^{r/2-1} f(t_{2k}) + f(b)\right]\right| \leqslant \frac{\|f''''\|}{180}(b-a)h^4 = \mathcal{O}(h^4).}$$

- **Composite quadrature rules summary**:

$$\int_a^b f(x)\,dx = h\sum_{i=1}^{r} f(a + (i - 1/2)h) + \frac{f''(\xi)}{24}(b-a)h^2 \qquad \text{(Midpoint)}$$

$$\int_a^b f(x)\,dx = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{r-1} f(a + ih) + f(b)\right] - \frac{f''(\xi)}{12}(b-a)h^2 \qquad \text{(Trapezoidal)}$$

$$\int_a^b f(x)\,dx = \frac{h}{3}\left[f(a) + 4\sum_{k=1}^{r/2} f(t_{2k-1}) + 2\sum_{k=1}^{r/2-1} f(t_{2k}) + f(b)\right] - \frac{f''''(\xi)}{180}(b-a)h^4 \qquad \text{(Simpson)}$$

In each case, ξ is some unknown point in the interval $[a, b]$.

- **Computational cost**: We measure the computational cost of a quadrature rule by counting the number of function evaluations required.

As h gets smaller, the number of subintervals r gets larger.

We can see from the above quadrature rules that we need:

- r function evaluations for the Midpoint rule;
- $r+1$ function evaluations for the Trapezoidal rule;
- $r+1$ function evaluations for the Simpson rule.

Thus, the Simpson rule is the most efficient in terms of number of function evaluations versus the order of accuracy.

- **Noisy data**: Numerical integration is not sensitive to random noise in the function values.
- **Gaussian quadrature**: The composite quadrature rules in the previous section all used points that were **equally spaced**. Such rules are called **Newton-Cotes formulas**.

In this section, we consider numerical methods for integration that use any choice of points $\{x_0, \ldots, x_n\}$.

We will see that by choosing the points $x_0, \ldots, x_n$ wisely, we will be able to obtain higher precision methods.

The goal is to have a formula that is **exact** for higher degree polynomials.

Gaussian quadrature uses the roots of the Legendre polynomials.

The Legendre polynomials (normalized so that $\phi_k(1) = 1$, for all k) can be described as follows:

$$\begin{aligned}
\phi_0(x) &= 1, \\
\phi_1(x) &= x, \\
\phi_{k+1}(x) &= \left(\frac{2k+1}{k+1}\right) x\phi_k(x) - \left(\frac{k}{k+1}\right) \phi_{k-1}(x), \quad k = 1, 2, \ldots.
\end{aligned}$$

The Legendre polynomial ϕ_k is of degree k and has the property that if $p(x)$ is a polynomial of degree at most $k-1$, then

$$\int_{-1}^{1} \phi_k(x) p(x)\, dx = 0.$$

- **Gauss points**: Let $x_0, \ldots, x_n$ be the roots of the Legendre polynomial ϕ_{n+1}; these points are known as the **Gauss points**.

Let p_n be the polynomial that interpolates the points

$$(x_0, f(x_0)), \ldots, (x_n, f(x_n)).$$

We can write p_n in **Lagrange form** as

$$p_n(x) = \sum_{i=0}^{n} f(x_i) L_i(x),$$

where the Lagrange polynomials are given by

$$L_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}, \quad i = 0, \ldots, n.$$

- **Gaussian quadrature**: We approximate the integral as

$$\int_{-1}^{1} f(x)\, dx \approx \int_{-1}^{1} p_n(x)\, dx = \sum_{i=0}^{n} c_i f(x_i),$$

where

$$c_i = \int_{-1}^{1} L_i(x)\, dx, \quad i = 0, \ldots, n.$$

- **Theorem: (Precision of Gaussian quadrature is $2n+1$)**:

$$\int_{-1}^{1} f(x)\, dx = \int_{-1}^{1} p_n(x)\, dx,$$

where p_n is the polynomial that interpolates f at the roots of the Legendre polynomial ϕ_{n+1} (i.e., at the **Gauss points** $x_0, \ldots, x_n$).

Proof. By the Polynomial Interpolation Error Theorem we have that for each $x \in [-1, 1]$, there is a $\xi(x) \in (-1, 1)$ such that

$$f(x) = p_n(x) + \frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - x_i).$$

First we will suppose that $f(x)$ is a polynomial having degree at most n. Then $f^{(n+1)} \equiv 0$, so we have that $f(x) = p_n(x)$, for all $x \in [-1, 1]$. Therefore,

$$\int_{-1}^{1} f(x)\, dx = \int_{-1}^{1} p_n(x)\, dx = \sum_{i=0}^{n} c_i f(x_i),$$

where

$$c_i = \int_{-1}^{1} \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} dx, \quad i = 0, \ldots, n.$$

Now we suppose that $n + 1 \leqslant \deg(f) \leqslant 2n + 1$. Dividing $f(x)$ by $\phi_{n+1}(x)$, we will obtain a polynomial **quotient** of $q(x)$ and a polynomial **remainder** of $r(x)$ such that

$$f(x) = q(x)\phi_{n+1}(x) + r(x)$$

and $\deg(r) < \deg(\phi_{n+1}) = n + 1$; thus, $\deg(r) \leqslant n$.

If $f(x)$ is any polynomial having degree at most $2n+1$, then

In addition, we have

$$\deg(f) = \deg(q) + \deg(\phi_{n+1}),$$

so

$$n+1 \leqslant \deg(q) + \deg(\phi_{n+1}) \leqslant 2n+1,$$

which implies that

$$0 \leqslant \deg(q) \leqslant n.$$

Thus, we have that

$$\begin{aligned}\int_{-1}^{1} f(x)\,dx &= \int_{-1}^{1} q(x)\phi_{n+1}(x)\,dx + \int_{-1}^{1} r(x)\,dx \\ &= 0 + \int_{-1}^{1} r(x)\,dx \\ &= \int_{-1}^{1} r(x)\,dx.\end{aligned}$$

.

Since $\deg(r) \leq n$ and

$$\begin{aligned}f(x_i) &= q(x_i)\phi_{n+1}(x_i) + r(x_i) \\ &= q(x_i)\cdot 0 + r(x_i) \\ &= r(x_i),.\end{aligned}$$

for $i = 0, \ldots, n$, the first part of the proof then implies that

$$\int_{-1}^{1} r(x)\,dx = \sum_{i=0}^{n} c_i r(x_i) = \sum_{i=0}^{n} c_i f(x_i) = \int_{-1}^{1} p_n(x)\,dx.$$

Thus, we have that

$$\int_{-1}^{1} f(x)\,dx = \int_{-1}^{1} r(x)\,dx = \int_{-1}^{1} p_n(x)\,dx,$$

which completes the proof. ■

- **Gauss quadrature**: For $n = 0$, the only root of $\phi_1(x) = x$ is:

$$x_0 = 0$$

Then

$$\int_{-1}^{1} f(x)\,dx \approx 2f(0).$$

Thus, we obtain the **midpoint rule** which has **precision 1**.

For $n = 1$, the roots of $\phi_2(x) = \frac{1}{2}(3x^2 - 1)$ are:

$$x_0, x_1 = -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}$$

Then

$$\int_{-1}^{1} f(x)\,dx \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right).$$

This is a rule that has **precision 3**.

For $n = 2$, the roots of $\phi_3(x) = \frac{1}{2}(5x^3 - 3x)$ are:

$$x_0, x_1, x_2 = -\sqrt{\frac{3}{5}}, 0, \sqrt{\frac{3}{5}}$$

In this case, we have the **precision 5** rule:

$$\int_{-1}^{1} f(x)\,dx \approx \frac{1}{9}\left(5f\left(-\sqrt{\frac{3}{5}}\right) + 8f(0) + 5f\left(\sqrt{\frac{3}{5}}\right)\right).$$

In general, we have the formula

$$\int_{-1}^{1} f(x)\,dx = \sum_{i=0}^{n} c_i f(x_i) + \frac{2^{2n+3}[(n+1)!]^4}{(2n+3)![(2n+2)!]^2} f^{(2n+2)}(\xi),$$

for some $\xi \in [-1, 1]$, where

$$c_i = \frac{2(1 - x_i^2)}{(n+1)\phi_n(x_i)^2}, \quad i = 0, \ldots, n.$$

- **Composite Gaussian quadrature**: We will now approximate $\int_a^b f(x)\,dx$ by dividing the interval $[a, b]$ into r subintervals:

$$[t_0, t_1], \quad [t_1, t_2], \quad \ldots, \quad [t_{r-1}, t_r],$$

where $a = t_0 < t_1 < \cdots < t_r = b$.

In order to use Gaussian quadrature on interval $[t_{k-1}, t_k]$, we need to make a change of variables:

$$t = \frac{t_k - t_{k-1}}{2}(x + 1) + t_{k-1}, \quad -1 \leqslant x \leqslant 1.$$

Then,

$$\begin{aligned}\int_{t_{k-1}}^{t_k} f(t)\,dt &= \int_{-1}^{1} f\left(\frac{t_k - t_{k-1}}{2}(x+1) + t_{k-1}\right)\frac{t_k - t_{k-1}}{2}\,dx \\ &\approx \sum_{i=0}^{n} c_i f\left(\frac{t_k - t_{k-1}}{2}(x_i+1) + t_{k-1}\right)\frac{t_k - t_{k-1}}{2},\end{aligned}$$

.

where $x_0, \ldots, x_n$ are the **Gauss points** and

$$c_i = \int_{-1}^{1} \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} dx = \frac{2(1 - x_i^2)}{(n+1)\phi_n(x_i)^2}, \quad i = 0, \ldots, n.$$

We can simplify this formula by defining

$$t_{k,i} = \frac{t_k - t_{k-1}}{2}(x_i + 1) + t_{k-1},$$

$$b_{k,i} = c_i \frac{t_k - t_{k-1}}{2},$$

for $i = 0, \ldots, n$.

Then we have

$$\int_a^b f(x)\,dx \approx \sum_{k=1}^{r}\sum_{i=0}^{n} b_{k,i} f(t_{k,i}).$$

- **Uniform mesh**: If $[a,b]$ is divided into r equal subintervals with $h = (b-a)/r$ and $t_k = a + kh$, for $k = 0, \ldots, r$, then we define

$$t_{k,i} = \frac{h}{2}(x_i + 1) + t_{k-1}, \qquad b_i = c_i \frac{h}{2},$$

and it can be shown that

$$\int_a^b f(x)\,dx = \sum_{k=1}^{r}\sum_{i=0}^{n} b_i f(t_{k,i}) + \frac{(b-a)((n+1)!)^4}{(2n+3)!((2n+2)!)^2} f^{(2n+2)}(\xi) h^{2n+2},$$

for some $\xi \in [a,b]$. Thus, we have an accuracy of $\mathcal{O}(h^{2n+2})$ and the number of function evaluations is $r(n+1)$.

- **Adaptive quadrature**: Let

$$f(x) = e^{-3x} \sin 4x$$

and suppose we need to calculate $\int_0^4 f(x)\,dx$.

$f(x)$ varies dramatically for $x < 1$ and then has very small variation for $x > 1$. It would be better to divide the interval $[0,4]$ into many smaller subintervals in the region $[0,1]$ and fewer subintervals in the region $[1,4]$.

We will do this subdivision **adaptively**, only when it helps improve the accuracy of numerical integration.

To do this, we need to obtain a good estimate of the error.

- **Computing an error estimate via Richardson extrapolation**: Let $I_f = \int_a^b f(x)\,dx$ and consider the **Simpson rule**

$$\left[\begin{array}{c} S(a,b) = \frac{h}{3} \\ f(a) + 4f(a+h) + f(b), \end{array}\right]$$

where $h = (b-a)/2$.

Let $S_1 = S(a,b)$.

Since the composite Simpson rule is a **fourth-order accurate** method, we have

$$I_f = S_1 + Kh^4 + \mathcal{O}(h^5).$$

Now consider using the **composite Simpson rule** with a step-size of $h/2$ on the subintervals $[a, a+h]$ and $[a+h, b]$:

$$S(a, a+h) = \frac{h}{6}\left[f(a) + 4f(a+h/2) + f(a+h)\right],$$

$$S(a+h, b) = \frac{h}{6}\left[f(a+h) + 4f(a+3h/2) + f(b)\right].$$

Let $S_2 = S(a, a+h) + S(a+h, b)$.

Then we have

$$I_f = S_2 + K\left(\frac{h}{2}\right)^4 + \mathcal{O}(h^5).$$

Now consider using the **composite Simpson rule** with a step-size of $h/2$ on the subintervals $[a, a+h]$ and $[a+h, b]$:

$$S(a, a+h) = \frac{h}{6}\left[f(a) + 4f(a+h/2) + f(a+h)\right],$$

$$S(a+h, b) = \frac{h}{6}\left[f(a+h) + 4f(a+3h/2) + f(b)\right].$$

Let $S_2 = S(a, a+h) + S(a+h, b)$.

Then we have

$$I_f = S_2 + K\left(\frac{h}{2}\right)^4 + \mathcal{O}(h^5).$$

Therefore,

$$S_1 + Kh^4 + \mathcal{O}(h^5) = S_2 + K\left(\frac{h}{2}\right)^4 + \mathcal{O}(h^5).$$

Then we solve for the error term Kh^4:

$$\begin{bmatrix} Kh^4 = \frac{16}{15} \\ S_2 - S_1 + \mathcal{O}(h^5). \end{bmatrix}$$

From this we conclude that

$$I_f - S_1 = \frac{16}{15}\left[S_2 - S_1\right] + \mathcal{O}(h^5),$$

$$I_f - S_2 = \frac{1}{15}\left[S_2 - S_1\right] + \mathcal{O}(h^5).$$

- **Divide-and-conquer**: We now describe a **divide-and-conquer** approach to obtain a quadrature approximation Q_f of $I_f = \int_a^b f(x)\,dx$ such that

$$|Q_f - I_f| < \texttt{tol},$$

where `tol` is some user-specified tolerance.

The idea is to do an **adaptive local refinement** of the grid of points on which we perform the composite Simpson rule (or any other quadrature rule):

$$a = t_0 < t_1 < \cdots < t_r = b.$$

Over each subinterval $[t_{i-1}, t_i]$, we compute $Q_i \approx I_i = \int_{t_{i-1}}^{t_i} f(x)\,dx$ such that

$$|Q_i - I_i| < \frac{h_i}{b-a}\texttt{tol},$$

where $h_i = t_i - t_{i-1}$, and then let

$$Q_f = \sum_{i=1}^{r} Q_i.$$

Then,

$$\begin{aligned}
|Q_f - I_f| &= \left| \sum_{i=1}^{r} Q_i - \sum_{i=1}^{r} I_i \right| \\
&= \left| \sum_{i=1}^{r} (Q_i - I_i) \right| \\
&\leqslant \sum_{i=1}^{r} |Q_i - I_i| \qquad \text{(by the Triangle Inequality)} \\
&< \sum_{i=1}^{r} \frac{h_i}{b-a} \texttt{tol} \\
&= \frac{\texttt{tol}}{b-a} \sum_{i=1}^{r} h_i \\
&= \texttt{tol}..
\end{aligned}$$

Thus, we just need to check that our error estimate for the current subinterval is small enough.

If the error estimate is not small enough, we divide the current subinterval into two equal pieces and repeat (recursively).