Numerical Methods

Toni Karvonen
toni.karvonen@lut.fi
Lappeenranta–Lahti University of Technology LUT

February 27, 2026

Contents

1 Floating-Point Numbers

This section discusses how numbers are represented in a computer. Design choices have to be made: there are infinitely many real numbers yet computational resources are finite.

1.1 Numeral Systems

The familiar *decimal system* system represents numbers in powers of 10. For example, the number 1453.529 expands as

$$1453.529 = 1\cdot 10^3 + 4\cdot 10^2 + 5\cdot 10^1 + 3\cdot 10^0 + 5\cdot 10^{-1} + 2\cdot 10^{-2} + 9\cdot 10^{-3}.$$

In this *positional notation* the contribution of a digit to the value of a number is determined by its value multiplied by a factor that equals a chosen *base* (10 in the decimal system) raised to a power that depends on the position of the digit. The position is counted from the *decimal separator* with non-negative powers on the left and negative on the right. For example, because 5 is the second digit to the left from the separator, it receives power 1 (because counting starts from zero on the left) and thus contributes $5\cdot 10^1 = 50$. Similarly, because 9 is the third digit to the right, it receives power -3 and contributes $9\cdot 10^{-3} = 0.009$.

Definition 1.1 (Positional number system)**.** Let $b \geq 2$ be an integer called *base* (or *radix*). A number in base b is written as

$$(a_n a_{n-1}\cdots a_0.c_1c_2\cdots)_b = a_n\cdot b^n + a_{n-1}\cdot b^{n-1} + \cdots + a_0\cdot b^0 + c_1\cdot b^{-1} + c_2\cdot b^{-2} + \cdots,$$

where the *digits* $a_0,\ldots,a_n$ and $c_1,c_2,\ldots$ are integers between 0 and $b-1$.

The subscript b is typically not used for decimal numbers (i.e., when $b = 10$) except for emphasis.

Example 1.2. To convert the octal (i.e., base $b = 8$) number $(35.07)_8$ to decimal we expand

$$(35.07)_8 = 3\cdot 8^1 + 5\cdot 8^0 + 0\cdot 8^{-1} + 7\cdot 8^{-2} = 3\cdot 8 + 5 + \frac{0}{8} + \frac{7}{8^2} = 29.109375.$$

Example 1.3. In the hexadecimal (i.e., base $b = 16$) the letters from A to F are used to represent the digits from 10 to 15. To convert the hexadecimal number $(\mathrm{AE.1B})_{16}$ to decimal we expand

$$\begin{aligned}(\mathrm{AE.1B})_{16} &= 10\cdot 16^1 + 14\cdot 16^0 + 1\cdot 16^{-1} + 11\cdot 16^{-2}\\ &= 10\cdot 16 + 14 + \frac{1}{16} + \frac{11}{256}\\ &= 174.10546875.\end{aligned}$$

Example 1.4. To convert $(220.1)_3$ to decimal we expand

$$(220.1)_3 = 2\cdot 3^2 + 2\cdot 3^1 + 0\cdot 3^0 + 1\cdot 3^{-1} = 2\cdot 9 + 2\cdot 3 + \frac{1}{3} = 24.333\ldots$$

Observe that a representation that terminates in one base, such as $(220.1)_3$, need not terminate in a different base.

Example 1.5. To convert the decimal number 33.25 to binary (i.e., base $b = 2$) we expand

$$\begin{aligned}(33.25)_{10} = 32 + 1 + \frac{1}{4} &= 2\cdot 2^5 + 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + 1\cdot 2^{-2} \\ &= (100001.01)_2.\end{aligned}$$

Binary system. In the binary system the digits (i.e., 0 and 1) are called binary digits, more commonly referred to as *bits*. For example, the binary number $(110.1)_2 = 6.5$ has four bits. The largest number that can be represented with n bits is

$$(\underbrace{11\cdots 11}_{n \text{ times}})_2 = 1\cdot 2^{n-1} + 1\cdot 2^{n-2} + \cdots + 1\cdot 2^1 + 1\cdot 2^0 = 2^n - 1. \tag{1.1}$$

1.2 Floating-Point Numbers

Although any base can be used to represent any number (possibly in a non-terminating manner), this is not possible in practice due to limited memory. When implementing numbers on a computer one has to choose which numbers to represent. This is done via floating-point numbers.

Definition 1.6 (Floating-point number)**.** A *floating-point number* $a \in \mathbb{R}$ in base b has the form

$$a = s\cdot c\cdot b^e,$$

where $s \in \{-1, 1\}$ is the *sign*, $c \in \mathbb{R}$ the c (or *significand*), and $e \in \mathbb{Z}$ the *exponent*.

The floating-point representation is not unique. For example, 123.45 can be written as

$$12345\cdot 10^{-2} \quad \text{or} \quad 1234.5\cdot 10^{-1},$$

or in infinitely many other ways as a decimal (i.e., base $b = 10$) floating-point number. In order to represent floating-point numbers on a computer one must select ranges for the mantissa and exponent. These choices determine the precision and memory cost of the resulting floating-point format.

1.3 IEEE 754 Standard

The *IEEE Standard for Floating-Point Arithmetic* (IEEE 754) defines a collection of floating-point formats used in virtually all computing. The binary IEEE 754 formats represent floating-point numbers in the form

$$a = s\cdot(1+f)\cdot 2^{e-e_{\min}}, \tag{1.2}$$

where $e_{\min}$, the *exponent bias*, is a positive integer and $f \in (0,1)$ the *fraction*. Both $f \in (0,1)$ and $e \in \mathbb{Z}_{\geq 0}$ are expressed in binary with chosen numbers of bits. That is,

$$f = (0.\bar{b}_1\bar{b}_2\cdots\bar{b}_p)_2 \quad \text{and} \quad e = (\hat{b}_1\hat{b}_2\cdots\hat{b}_q)_2$$

for some positive integeres p and q and bits $\bar{b}_i, \hat{b}_j, \in \{0,1\}$. The sign is represented by a sign b_s in the form $s = (-1)^{b_s}$, so that

$$b_s = 0 \implies s = 1 \qquad \text{and} \qquad b_s = 1 \implies s = -1.$$

To represent a floating-point number in this format the $n = p + q + 1$ bits (p bits for the fraction, q for the exponent and one for the sign) are ordered to give an IEEE 754 binary representation of a floating-point number:

$$(b_s\hat{b}_q\hat{b}_{q-1}\cdots\hat{b}_1\bar{b}_p\bar{b}_{p-1}\cdots\bar{b}_1)_2 = (b_{p+q}b_{p+q-1}\cdots b_0)_2. \tag{1.3}$$

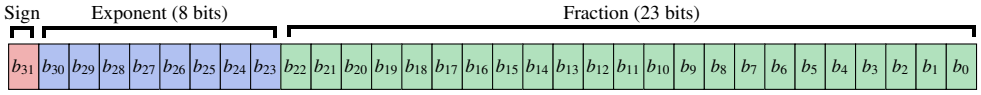

Figure 1.1: Bit layout for 32-bit IEEE 754 floats. See Table 1.1.

Figure 1.1 illustrates the bit scheme. In IEEE 754 the integer $e_{\min}$ in (1.2) that determines the *actual exponent* of two is set as $e_{\min} = 2^{q-1} - 1$. This means that the actual exponent ranges [recall (1.1)]

$$\text{from} \quad e_{\min} = 2^{q-1} - 1 \quad \text{to} \quad e_{\max} = 2^q - 1 - e_{\min} = 2(2^{q-1} - 1) = 2^{q-1}. \tag{1.4}$$

Example 1.7. Consider IEEE 754 format floating-point numbers with $p = 3$ bits for the fraction and $q = 2$ bits for the exponent. The total number of bits is $n = p + q + 1 = 6$. The binary number $(101110)_2$ represents a floating-point number of the form (1.2) with $s = (-1)^1 = -1$, $e = (01)_2 = 1$, and $f = (0.110)_2 = 0.75$. Because $e_{\min} = 2^{q-1} - 1 = 1$, $(101110)_2$ corresponds to the decimal number

$$s \cdot (1+f) \cdot 2^{e-e_{\min}} = (-1) \cdot (1+0.75) \cdot 2^{1-1} = -1.75.$$

by (1.2).

Special quantities in IEEE 754. There are exceptions related to the exponents $e = (0\cdots0)_2$ and $e = (1\cdots1)_2$:

- If the exponent is zero, $e = (0\cdots0)_2$:
  - If the *fraction is zero*, $f = (0.0\cdots0)_2$, the number is interpreted as zero. The sign bit determines the whether the zero is "positive" or "negative", which has no effect on most computations. That is,

    $$\text{"positive" zero} = (10\cdots0)_2 \quad \text{and} \quad \text{"negative" zero} = (00\cdots0)_2.$$

  - If the *fraction is not zero*, the floating-point number is called *subnormal* and represents a decimal number of the form

    $$a_{\text{subnormal}} = s \cdot f \cdot 2^{-e_{\min}+1}, \tag{1.5}$$

    which differs from a *normal* number of the form (1.2) by $1 + f \in (1,2)$ being replaced by $f \in (0,1)$. With subnormal numbers smaller numbers can be represented than what would be possible with normal numbers alone.
- If the exponent is maximal, $e = (1\cdots1)_2$:
  - If the *fraction is zero*, $f = (0.0\cdots0)_2$, the number is interpreted as positive (∞) or negative infinite ($-\infty$) depending on the sign bit.
  - If the *fraction is not zero*, the number is interpreted as *not a number* (NaN), meaning that the number is undefined. This can occur when trying to compute $0/0$.

IEEE 754 binary formats. Table 1.1 contains the specifications and key properties of the three most common IEEE 754 binary floating-point formats. These are typically referred to as *half-precision* numbers (16 bit), *floats* or *single-precision numbers* (32 bits), and *doubles* or *double-precision numbers* (64 bits).

Table 1.1: Key properties of IEEE 754 half, single-precision float, and double-precision formats. Note that each format includes one bit for the sign in addition to the exponent and fraction bits. *Machine epsilon* quantifies the precision of a floating-point system. It refers to the difference between 1 and the next largest floating-point number. One can work out that $\varepsilon = 2^{-p}$, where p is the number of bits in the fraction. *The number bits in the fraction determines precision and the number of bits in the exponent determines how small and large numbers can be represented.*

Property	**Half (16-bit)**	**Float (32-bit)**	**Double (64-bit)**
Total bits	16	32	64
Exponent bits (q)	5	8	11
Fraction bits (p)	10	23	52
Exponent bias ($e_{\min}$)	15	127	1023
Smallest positive normal	2^{-14}	2^{-126}	2^{-1022}
Smallest positive subnormal	2^{-24}	2^{-149}	2^{-1074}
Largest finite value ($\approx$)	6.55×10^{4}	3.40×10^{38}	1.80×10^{308}
Machine epsilon (ε)	2^{-10}	2^{-23}	2^{-52}

Example 1.8. Let us find the IEEE 754 float (32 bits) representation of the decimal number $a = -3.75$. Since the sign is $s = (-1)^{b_s}$, the sign bit is one, $b_s = 1$. By (1.4), the exponent bias is $e_{\min} = 2^{q-1} - 1 = 2^7 - 1 = 127$. Because $1 + f \in (1,2)$ and $(1+f)\cdot 2^1 \in (2,4)$, the actual exponent must be 1, meaning that $1 = e - e_{\min} = e - 127$. Therefore $e = 128 = 2^7 = (10\cdots0)_2$, where there are seven zeros. From $3.75 = (1+f)\cdot 2$ we solve $f = \frac{1}{2}\cdot 3.75 - 1 = 0.875 = (0.111)_2$. Therefore the float representation of -3.75 is

$$1\,10000000\,11100000000000000000000,$$

where we use the same colours as in Figure 1.1 to represent the sign, exponent, and fraction, and have appended zeros to the fraction to ensure that the representation has 32 bits.

Example 1.9. Let us proceed to the opposite direction and find out which decimal number a the float represention

$$0\,00010100\,01100000000000000000000$$

corresponds to. The sign bit is zero ($b_s = 0$), so that the sign is positive because $s = (-1)^{b_s} = (-1)^0 = 1$. The exponent is $e = (00010100)_2 = 2^4 + 2^2 = 20$. The fraction is $f = (0.01100000000000000000000)_2 = 2^{-2} + 2^{-3} = 0.375$. Therefore

$$a = s\cdot(1+f)\cdot 2^{e-e_{\min}} = 1\cdot(1+0.375)\cdot 2^{20-127} = 1.375\cdot 2^{-107} \approx 8.47\times 10^{-33}.$$

Example 1.10. Let us verify some properties of floats in Table 1.1. By (1.4), the exponent bias is $e_{\min} = 2^{q-1} - 1 = 2^7 - 1 = 127$. The smallest possible positive normal number is the smallest positive floating-point number of the form (1.2). This means that we want (a) a fraction that is as small as possible and (b) a *positive* exponent (if the exponent were zero

Figure 1.2: Bit layout for 16-bit IEEE 754 half-precision floating point numbers. See Table 1.1.

the number would be subnormal) that is as small as possible. This means that

$$f = (0.\underbrace{0\cdots0}_{23\text{ times}})_2 = 0 \quad \text{and} \quad e = (\underbrace{0\cdots0}_{7\text{ times}}1)_2 = 1.$$

Therefore the smallest positive *normal* float is

$$a = 1\cdot(1+f)\cdot 2^{e-e_{\min}} = 1\cdot(1+0)\cdot 2^{1-127} = 2^{-126} \approx 1.18\times 10^{-38}.$$

To get the smallest positive *subnormal* float we set the exponent to zero, select the smallest positive fraction, $f = (0.0\cdots01)_2 = 2^{-23}$ and use the subnormal form (1.5), which gives

$$a = 1\cdot f\cdot 2^{-127+1} = 2^{-23}\cdot 2^{-126} = 2^{-149} \approx 1.40\times 10^{-45}.$$

The largest finite float has largest possible fraction and an exponent that is one minus the largest possible exponent (because the largest exponent is interpreted as infinity or NaN):

$$f = (0.\underbrace{1\cdots1}_{23\text{ times}})_2 = \sum_{k=1}^{23} 2^{-k} = 1-2^{-23} \quad \text{and} \quad e = (\underbrace{1\cdots1}_{7\text{ times}}0)_2 = \sum_{k=1}^{7} 2^k = 254.$$

These give the largest finite float

$$a = 1\cdot(1+f)\cdot 2^{e-e_{\min}} = (2-2^{-23})\cdot 2^{127} \approx 3.40\times 10^{38}.$$

Floating-point numbers in Python. In Python, one way to recover the 32-bit binary representation of a floating-point number is as follows:

```
import numpy as np
a = -3.75
res = np.binary_repr(np.float32(a).view(np.int32), width=32)
print(res)
```

Running these lines of code should print `11000000011100000000000000000000`, which matches with the representation we obtained in Example 1.8. However, note that in Python the term "float" refers to double-precision (64-bit) floating point numbers.

1.4 Rounding and Precision

Floating-point numbers provide a *finite-precision arithmetic*. That is, not all real numbers can be represented exactly and the results of arithmetic operations can differ from *exact arithmetic*. For example, the familiar decimal number 0.1 cannot be represented exactly as a non-terminating binary number:

$$(0.1)_{10} = (0.00011001100110011\ldots)_2.$$

Consequently, such numbers cannot be represented exactly as binary floating-point numbers and are rounded to a representable floating-point number according the rounding rules specified in the IEEE 754 standard. This can be readily observed in computations. For example, in Python

a simple multiplication whose value in exact arithmetic is 653082380.34 becomes something different:

```
>>> print(27869.01 * 23434)
653082380.3399999
```

We can observe the same phenomenon when printing, for example, 50 decimal digits of $a = 0.1$:

```
>>> a = 0.1
>>> print(f"{a:.{50}f}")
0.10000000000000000555111512312578270211815834045410
```

Precision. Floating-point numbers are *not uniform in precision*. This is both an advantage (a huge range of numbers can be represented) and a disadvantage (rounding errors are inevitable when operating with large numbers). For example, with doubles (64 bits) one can represent numbers between 1 and 2 with precision 2^{-52}, meaning that all numbers on the interval $[1,2]$ round to a multiple of 2^{-52}. Precision decreases as numbers become larger, being 2^{-51} for numbers between 2 and 4 and eventually 2^{971} for numbers between 2^{1023} and 2^{1024}. We can easily see this in action in Python (which uses doubles by default). The following computations return positive values as they should:

```
>>> b = 1.0
>>> a = 1.0 + 2.0**(-52)
>>> print(a - b)
2.220446049250313e-16
>>> b = 2.0**1023
>>> a = 2.0**1023 + 2.0**971
>>> print(a - b)
1.99584030953472e+292
```

Note that in the second computation it is essential to write "`2.0`" rather than "2" to ensure that Python interprets the numbers as floating-point numbers rather than integers, which are handled differently. However, if we make a slightly smaller it is rounded to b and the difference is incorrectly (in comparison to exact arithmetic) evaluated as zero:

```
>>> b = 1.0
>>> a = 1.0 + 0.5*2.0**(-52)
>>> print(a - b)
0.0
>>> b = 2.0**1023
>>> a = 2.0**1023 + 0.5*2.0**971
>>> print(a - b)
0.0
```

Underflow and overflow. Similar phenomena occur when one tries to do arithmetic with numbers smaller than the *underflow level* (the row with smallest positive subnormals in Table 1.1) rounding to zero and numbers larger than the *overflow level* (the wor with largest finite values in Table 1.1) rounding to infinity or throwing an error. For 64-bit doubles these levels are 2^{-1074} and approximately 1.80×10^{308}.

1.5 Numerical Stability

The inherent inadequacies of floating-point numbers need to be taken into account when designing numerical algorithms. *Numerical stability* of an algorithm refers to its ability to tolerate numerical

errors. One should be particularly careful when subtracting nearby floating-point numbers and avoid doing this if possible.

Example 1.11. Computing $\sqrt{1+x}-\sqrt{x} > 0$ is prone to numerical cancellations when x is large because in that case both $\sqrt{1+x}$ and $\sqrt{x}$ are large and nearby but far away from nearest representable floating-point numbers due to their sparsity (recall Section 1.4). For example, for $x = 2^{60}$ this expression evaluates to zero in the 64-bit arithmetic used in Python. In contrast, the equivalent form

$$\sqrt{1+x}-\sqrt{x} = \frac{(\sqrt{1+x}-\sqrt{x})(\sqrt{1+x}+\sqrt{x})}{\sqrt{1+x}+\sqrt{x}} = \frac{1}{\sqrt{1+x}+\sqrt{x}},$$

which used the identity $(a-b)(a+b) = a^2-b^2$, evaluates to a positive value even for $x = 2^{1023}$.

Example 1.12. Similar phenomenon can occur when computing a^2-b^2 for large a and b. This computation can be made more stable by using the identity $(a-b)(a+b) = a^2-b^2$. While this identity contains a subtraction $a-b$, the numbers being subtracted are smaller than in a^2-b^2 whenever $a, b > 1$. Therefore floating-point rounding is less extreme.

The following lines of Python illustrate Example 1.12:

```
>>> a = 2.0**50
>>> b = a - 1.0
>>> print(a**2 - b**2)
2251799813685248.0
>>> print((a - b)*(a + b))
2251799813685247.0
```

The correct value of a^2-b^2 for $a = 2^{50}$ and $b = 2^{50}-1$ is 2251799813685247, which the computation based on the numerically more stable form $(a-b)(a+b)$ returned.

2 Error

This section discusses the nature of numerical errors in computation and how errors are measured.

2.1 Rounding and Truncation Error

Numerical errors arise both from the inherent inaccuracy of finite-precision arithmetic and the application mathematical approximations.

Definition 2.1 (Rounding error)**.** *Rounding error* is the error caused by the use of finite-precision arithmetic (e.g., floating-point numbers) as opposed to exact arithmetic.

Example 2.2. The rounding and precision problems that we observed in Sections 1.4 and 1.5 are examples of rounding errors.

Definition 2.3 (Truncation error)**.** *Truncation error* is the error caused caused by a numerical approximation of a mathematical process.

Example 2.4. From basic analysis we recall that the exponential constant $e \approx 2.718$ can be expressed as the infinite series

$$e = \sum_{k=0}^{\infty} \frac{1}{k!},$$

where $n! = 1 \cdots 2 \cdots (n-1) \cdot n$ is the factorial and $0! = 1$. To approximate e one can truncate this series after n terms (here $n \in \{0,1,2,3\}$):

$$\begin{aligned}
e \approx e_0 &= \sum_{k=0}^{0} \frac{1}{k!} = \frac{1}{0!} = \frac{1}{1} = 1, \\
e \approx e_1 &= \sum_{k=0}^{1} \frac{1}{k!} = \frac{1}{0!} + \frac{1}{1!} = \frac{1}{1} + \frac{1}{1} = 2, \\
e \approx e_2 &= \sum_{k=0}^{2} \frac{1}{k!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} = \frac{1}{1} + \frac{1}{1} + \frac{1}{2} = 2.5, \\
e \approx e_3 &= \sum_{k=0}^{3} \frac{1}{k!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} = \frac{1}{1} + \frac{1}{1} + \frac{1}{2} + \frac{1}{2 \cdot 3} = 2.666\ldots
\end{aligned}$$

Any of the differences $e - e_n$ for $n \in \{0,1,2,3\}$ is a truncation error.

However, truncation error does not refer only to explicit truncation as in Example 2.4 but to any type of numerical approximation.

Example 2.5. Recall again from basic analysis that the derivative of a function $f\colon \mathbb{R} \to \mathbb{R}$ at a point $x_0 \in \mathbb{R}$ is defined as the limit

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

of the difference quotient $[f(x_0 + h) - f(x_0)]/h$. One can therefore approximate the derivative by computing the difference quotient for a small positive h. For example, the derivative

of $f(x) = x^2$ at $x_0 = 1$ can be approximated as (with $h = 1/10$)

$$f'(x_0) \approx \frac{f(x_0+h) - f(x_0)}{h} = \frac{(1+1/10)^2 - 1^2}{1/10} = \frac{21}{10}.$$

Since $f'(x) = 2x$, the true derivative is $f'(x_0) = f'(1) = 2$. The error

$$f'(x_0) - \frac{21}{10} = 2 - \frac{21}{10} = \frac{1}{10}$$

is a truncation error even though no summation is being truncated. However, even here one can think in terms of a truncation: one is "truncating the limiting process" $h \to 0$ by picking a small h and using that rather than going all the way to zero.

2.2 Precision of Approximation

Definition 2.6 (Precision of approximation)**.** An approximation $\tilde{x}$ to x is said to *approximate x to d decimal places* if

$$|x - \tilde{x}| < \frac{1}{2}10^{-d}.$$

For example, if we want $\tilde{x}$ to approximate x to the nearest integer (i.e., to zero decimal places), then it must hold that $|x - \tilde{x} < 0.5$. This means that the distance between x and $\tilde{x}$ is always rounded to zero when approximated to the nearest integer. Note that this does not mean that both numbers are rounded to the same value when approximated to the nearest integer. For example, if $x = 0.51$ and $\tilde{x} = 0.49$, then $|0.51 - 0.49| = 0.02 < 0.5$ but $x \approx 1$ and $\tilde{x} \approx 0$. Note also that if $\tilde{x}$ approximates x to d decimal places, it also approximates it to $d-1$ decimal places, and so on. What matters is, of course, the highest decimal precision is.

Example 2.7. The number 3.14 approximates π to two decimal places since

$$|\pi - 3.14| = 0.00159\ldots = 0.159\ldots \cdot 10^{-2} < 0.5 \cdot 10^{-2}.$$

The number 3.14 approximates π also to one decimal place since

$$|\pi - 3.14| = 0.00159\ldots = 0.0159\ldots \cdot 10^{-1} < 0.5 \cdot 10^{-1}.$$

The numbe 3.14 does *not* approximate π to three decimal places since

$$|\pi - 3.14| = 0.00159\ldots = 1.59\ldots \cdot 10^{-3} > 0.5 \cdot 10^{-3}.$$

2.3 Absolute and Relative Error

Error can be measured in either absolute or relative terms.

Definition 2.8 (Absolute and relative error)**.** Let $\tilde{x}$ be an approximation of x. Then

$$\textit{absolute error} = |x - \tilde{x}| \quad \text{and} \quad \textit{relative error} = \frac{|x - \tilde{x}|}{|x|}.$$

The absolute error is simply the magnitude of the difference between the true quantity of interest x and its approximation $\tilde{x}$. The relative error measures the error *in proportion to the magnitude of the true value*. Note that when computing the relative error one divides by the magnitude of the true value, not of the approximation. The relative error is often expressed in percents.

Example 2.9. If the number $x = \pi$ is approximated to two decimal places as in Example 2.7 (i.e., $\tilde{x} = 3.14$), then the absolute error of the approximation is

$$|\pi - 3.14| = 0.00159\ldots$$

and the relative error is

$$\frac{|\pi - 3.14|}{|\pi|} = 0.000507\ldots = 0.0507\ldots\,\%.$$

Relative error is often the more useful of the two errors.

Example 2.10. In Section 1.4 we observed that in Python's 64-bit double-precision arithmetic the number

$$x = 2^{1023} + 0.5 \cdot 2^{971} \quad \text{is rounded to} \quad \tilde{x} = 2^{1023}.$$

Therefore the absolute rounding error is

$$|x - \tilde{x}| = |2^{1023} + 0.5 \cdot 2^{971} - 2^{1023}| = 0.5 \cdot 2^{971} = 9.9792 \cdot 10^{291}.$$

This somewhat large number is very small in proportion to the size of x as the relative rounding error is minuscule:

$$\frac{|x - \tilde{x}|}{|x|} = \frac{|2^{1023} + 0.5 \cdot 2^{971} - 2^{1023}|}{|2^{1023} + 0.5 \cdot 2^{971}|} \approx 1.11 \cdot 10^{-16} = 1.11 \cdot 10^{-14}\,\%.$$

2.4 Big-O Notation

Asymptotic big-O notation provides a convenient and concise way to describe the asymptotical, or limiting, behaviour of approximation.

Definition 2.11 (Big-O notation for functions)**.** Let $x_0 \in (a,b)$ and let $f, g\colon (a,b) \to \mathbb{R}$ be functions. We write

$$f(x) = O(g(x)) \quad \text{as} \quad x \to x_0$$

if there are $C \geq 0$ and $\delta > 0$ such that $|f(x)| \leq Cg(x)$ for all $x \in (x_0 - \delta, x_0 + \delta)$.

Recall that a sequence $(a_n)_{n=1}^{\infty} = (a_1, a_2, \ldots)$ is an ordered collection of real numbers.

Definition 2.12 (Big-O notation for sequences)**.** Let $(a_n)_{n=1}^{\infty}$ and $(b_n)_{n=1}^{\infty}$ be two sequences. We write

$$a_n = O(b_n) \quad \text{as} \quad n \to \infty$$

if there is $C \geq 0$ such that $|a_n| \leq Cb_n$ for all $n \geq 1$.

The notation $O(g(x))$ is read as "of order $g(x)$", "big-oh of $g(x)$", or "oh of $g(x)$". If clear from the context, the limit ($x \to x_0$ or $n \to \infty$) is often omitted. On this course we use big-O notation to describe the behaviour of numerical approximation. However, this notation is very versatile, being also used to describe the time and memory complexity of algorithms.[1] If $g(x) \to 0$ as $x \to x_0$ (or $b_n \to 0$ as $n \to \infty$), the statement $f(x) = O(g(x))$ gives information on how fast $f(x)$ tends to zero as $x \to x_0$. Namely, this means that $f(x)$ tends to zero *at least* as fast as $g(x)$. In such a situation one wants to find g that tends to zero as fast as possible such that $f(x) = O(g(x))$ remains true.

[1] *Data Structures and Algorithms* (BM40A1500).

Example 2.13. Suppose that we want to determine the order of $f(x) = x^2 + x^3$ as $x \to 0$ (i.e., $x_0 = 0$). By Definition 2.11 we need to find a function g and constants $C \geq 0$ ja $\delta > 0$ such that $|f(x)| \leq Cg(x)$ for all $x \in (-\delta, \delta)$. Select $\delta = 1$ (any other value less than one would do as well). Then, for all $x \in (-1, 1)$,

$$|f(x)| = x^2 + |x|^3 \leq x^2 + |x|^2 = 2x^2,$$

where we used the fact that $|a|^n \leq |a|^m$ if $|a| \leq 1$ and $n \geq m$. Therefore we can select $C = 2$ and $g(x) = x^2$, meaning that $f(x)$ is of order x^2 [$f(x) = O(x^2)$] as $x \to 0$. However, note that we also have

$$|f(x)| = x^2 + |x|^3 \leq x^2 + |x|^2 \leq |x| + |x| = 2|x|,$$

so that we could have chosen $g(x) = |x|$.

In Example 2.13 the order $g(x) = x^2$ is better than $g(x) = |x|$ because it describes the behaviour of f close to $x_0 = 0$ more accurately. Namely, as $x \to 0$,

$$f(x) = O(x^2) \quad \text{and} \quad f(x) = O(|x|)$$

but

$$x^2 = O(|x|) \quad \text{and} \quad |x| \neq O(x^2),$$

where $|x| \neq O(x^2)$ means that $|x|$ is *not* of order x^2 as $x \to 0$. When written down these four big-O statements are

$$f(x) \text{ tends to zero at least as fast as } x^2 \quad \text{and} \quad f(x) \text{ tends to zero at least as fast } |x|$$

but

$$x^2 \text{ tends to zero at least as fast as } |x| \quad \text{and} \quad |x| \text{ tends to zero } \textit{slower} \text{ than } x^2.$$

Figure 2.1 illustrates big-O notation for common polynomials around $x_0 = 0$.

Remark 2.14. To write $f(x) = O(g(x))$ as $x \to x_0$ is an abuse of notation. Because "=" commutes elsewhere in mathematics (i.e., $a = b$ is the same as $b = a$), $f(x) = O(g(x))$ suggests that one could write $O(g(x)) = f(x)$, which is difficult to make sense of. What $f(x) = O(g(x))$ as $x \to x_0$ means is that f *is an element of the collection of functions that*

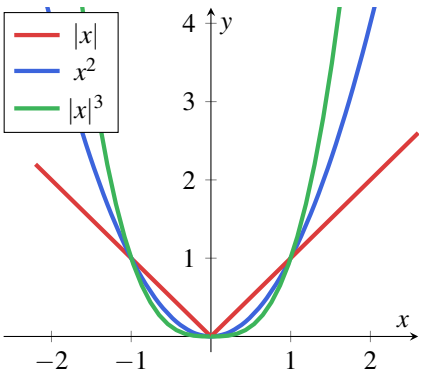

Figure 2.1: The functions $|x|$, x^2 and $|x|^3$ around $x_0 = 0$. Because $|x|$ is above x^2 when $x \in (-1, 1)$ and x^2 is above $|x|^3$, the figure illustrates that $x^2 = O(|x|)$ and $|x|^3 = O(x^2)$ as $x \to 0$. *Observe that the ordering of the graphs changes when* $|x| > 1$. If we considered a point x_0 such that $|x_0| > 1$ (say $x_0 = 2$), the big-O relations would need to be reversed: $|x| = O(x^2)$ and $x^2 = O(|x|^3)$ as $x \to 2$. Note that here we must use $|x|$ and $|x|^3$ rather than x and x^3 because x and x^3 are negative when $x < 0$ but $f(x) = O(g(x))$ enforces an upper bound on $|f(x)| \geq 0$ rather than $f(x)$.

are bounded from above by some constant times g around x_0. The notation must therefore be interpreted as convenient shorthand only.

2.5 Properties of big-O Notation

The following proposition shows that big-O notation is *transitive*. The proposition is nothing but a big-O version of the elementary fact that $a \leq b$ and $b \leq c$ implies $a \leq c$.

Proposition 2.15. *If $f(x) = O(g_1(x))$ and $g_1(x) = O(g_2(x))$ as $x \to x_0$, then*

$$f(x) = O(g_2(x)) \quad \text{as} \quad x \to x_0.$$

Proof. By Definition 2.11, there are $C_1, C_2 \geq 0$ and $\delta_1, \delta_2 > 0$ such that

$$|f(x)| \leq C_1 g_1(x) \quad \text{for all} \quad x \in (x_0 - \delta_1, x_0 + \delta_1)$$

and

$$|g_1(x)| \leq C_2 g_2(x) \quad \text{for all} \quad x \in (x_0 - \delta_2, x_0 + \delta_2).$$

These two inequalities are valid simultaneously when $x \in (x_0 - \delta, x_0 + \delta)$ for $\delta = \min\{\delta_1, \delta_2\}$. Therefore, for $x \in (x_0 - \delta, x_0 + \delta)$,

$$|f(x)| \leq C_1 g_1(x) \leq C_1 |g_1(x)| \leq C_1 \cdot C_2 g_2(x),$$

so that we can set $g = g_2$, $C = C_1 \cdot C_2$ and $\delta = \min\{\delta_1, \delta_2\}$ in Definition 2.11. □

The next proposition shows that the constant C in Definition 2.11 does not affect the order.

Proposition 2.16. *If $f(x) = O(g_1(x))$ as $x \to x_0$ and $g_1(x) = C_{12} g_2(x)$ for $C_{12} \geq 0$, then $f(x) = O(g_2(x))$ as $x \to x_0$.*

Proof. By Definition 2.11 there are $C_1 \geq 0$ and $\delta > 0$ such that $|f(x)| \leq C_1 g_1(x)$ for all $x \in (x_0 - \delta, x_0 + \delta)$. From the assumption $g_1(x) = C_{12} g_2(x)$ it follows that, for all $x \in (x_0 - \delta, x_0 + \delta)$,

$$|f(x)| \leq C_1 g_1(x) = C_1 \cdot C_{12} g_2(x),$$

so that we can select $C = C_1 \cdot C_{12}$ and $g = g_2$ in Definition 2.11. □

Propositions 2.15 and 2.16 have natural variants for sequences (Definition 2.12). The following example shows a rather typical application of big-O notation in numerical approximation where one is usually interested in the behaviour of an error as some *discretisation parameter* h tends to zero.

Example 2.17. In Example 2.5 we approximated the derivative of $f(x) = x^2$ at $x_0 = 1$ with the difference quotient

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

We now compute the order of the absolute error of this approximation as $h \to 0$. Because $f'(x) = 2x$ and $x_0 = 1$, the absolute error $\varepsilon(h)$ is

$$\begin{aligned}
\varepsilon(h) = |f'(x_0) - q(h)| = \left| f'(x_0) - \frac{f(x_0 + h) - f(x_0)}{h} \right| &= \left| 2 - \frac{(1+h)^2 - 1}{h} \right| \\
&= \left| 2 - \frac{2h + h^2}{h} \right| \\
&= |2 - (2 + h)| \\
&= |h|,
\end{aligned}$$

where we used the identity $(a+b)^2 = a^2 + 2ab + b^2$. This shows that $\varepsilon(h) = O(|h|)$, which means that the error of the difference quotient approximation $f'(x_0) \approx q(h)$ tends to zero with *linear order* as $h \to 0$. Note that the absolute error $\rho(h)$ is of the same order because

$$\rho(h) = \frac{|f'(x_0) - q(h)|}{|f'(x_0)|} = \frac{1}{|f'(x_0)|}\varepsilon(h),$$

where $1/|f'(x_0)| = 1/2 > 0$. We can now use Proposition 2.16 to deduce that $\rho(h) = O(|h|)$.

We shall mostly work with errors $\varepsilon(h)$ that have *polynomial orders* as $h \to 0$. If a function is a polynomial, the term with the smallest degree determines the rate.

Proposition 2.18. *If* $\varepsilon(h) = a_n h^n + a_{n+1}h^{n+1} + \cdots + a_{n+m}h^{n+m}$ *for* $n, m \in \mathbb{N} \cup \{0\}$ *and* $a_n, \ldots, a_{n+m} \in \mathbb{R}$*, then*

$$\varepsilon(h) = O(|h|^n) \quad \text{as} \quad h \to 0.$$

Proof. Select $\delta = 1$ and note that $x_0 = 0$. For $h \in (x_0 - \delta, x_0 + \delta) = (-1, 1)$,

$$|\varepsilon(h)| = |a_n h^n + a_{n+1}h^{n+1} + \cdots + a_{n+m}h^{n+m}| \leq |a_n||h|^n + \cdots + |a_{n+m}||h|^{n+m},$$

where we used the triangle inequality $|a+b| \leq |a| + |b|$. Because $|h|^{n+m} \leq |h|^n$ for $h \in (-1, 1)$, we obtain

$$|\varepsilon(h)| \leq (|a_n| + \cdots + |a_{n+m}|)|h|^n,$$

where the constant $C = |a_n| + \cdots + |a_{n+m}|$ is non-negative (i.e., ≥ 0). It follows from Definition 2.11 that $\varepsilon(h) = O(|h|^n)$. □

We are yet to discuss Definition 2.12, which defines big-O notation for sequences. On this course we use this notation to describe the error of numerical methods as n, the number of *discretisation points* or *time steps*, grows.

Example 2.19. Recall Example 2.4 where we approximated the exponential constant $e \approx 2.718$ by truncating the infinite series expansion $e = \sum_{k=0}^{\infty} 1/k!$ after n terms:

$$e \approx e_n = \sum_{k=0}^{n} \frac{1}{k!}.$$

Let us determine the order of this approximation (i.e., the order of the error $\varepsilon_n = |e - e_n|$) as $n \to \infty$. We have

$$\varepsilon_n = |e - e_n| = \left| \sum_{k=0}^{\infty} \frac{1}{k!} - \sum_{k=0}^{n} \frac{1}{k!} \right| = \left| \sum_{k=n+1}^{\infty} \frac{1}{k!} \right| = \sum_{k=n+1}^{\infty} \frac{1}{k!}. \tag{2.1}$$

For any $m \geq 1$ we can write

$$(n+m)! = 1 \cdot 2 \cdots (n+1) \cdot (n+2) \cdots (n+m) = (n+1)! \cdot (n+2) \cdots (n+m), \tag{2.2}$$

where the product $(n+2)\cdots(n+m)$ is considered equal to one if $m = 1$. By plugging (2.2) in (2.1) we obtain

$$\varepsilon_n = \sum_{k=n+1}^{\infty} \frac{1}{k!} = \sum_{m=1}^{\infty} \frac{1}{(n+m)!} = \frac{1}{(n+1)!}\left(1 + \sum_{m=2}^{\infty} \frac{1}{(n+2)\cdots(n+m)}\right).$$

The final series converges (we skip the proof), which means that we can set

$$a_n = \varepsilon_n, \quad b_n = \frac{1}{(n+1)!} \quad \text{and} \quad C = 1 + \sum_{m=2}^{\infty} \frac{1}{(n+2)\cdots(n+m)}$$

in Definition 2.12. Therefore

$$\varepsilon_n = |e - e_n| = O\left(\frac{1}{(n+1)!}\right).$$

3 Building Blocks of Numerical Methods

This section reviews basic notions and tools that recur when numerical methods are constructed and implemented.

3.1 Taylor Polynomials

The Taylor expansion of a smooth (i.e., sufficiently many times differentiable) function forms the basis for of many numerical methods.

Definition 3.1 (Taylor polynomials)**.** Let $f\colon (a,b) \to \mathbb{R}$ be an n times differentiable function at a point $x_0 \in (a,b)$. The nth *Taylor polynomial* of f at the point x_0 is

$$P_n(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2!}(x-x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n. \tag{3.1}$$

The nth Taylor polynomial is also called the Taylor polynomial *of degree* n.

Recall that $f^{(n)}(x)$ stands for the nth derivative of f at a point x. That is,

$$f^{(n)}(x) = \frac{\mathrm{d}^n}{\mathrm{d}x^n} f(x).$$

Moreover, the zeroth derivative is simply the function itself: $f^{(0)}(x) = f(x)$. Therefore the summation notation can be used to write the Taylor polynomial as

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

Example 3.2. Let us form the Taylor polynomials of degrees $n \in \{1,3,5\}$ for the function $f(x) = \sin(x)$ at the point $x_0 = 0$. To form these Taylor polynomials we need the values of derivatives of f up to order 5 at $x_0 = 0$. By recalling that the derivative of $\sin(x)$ is $\cos(x)$

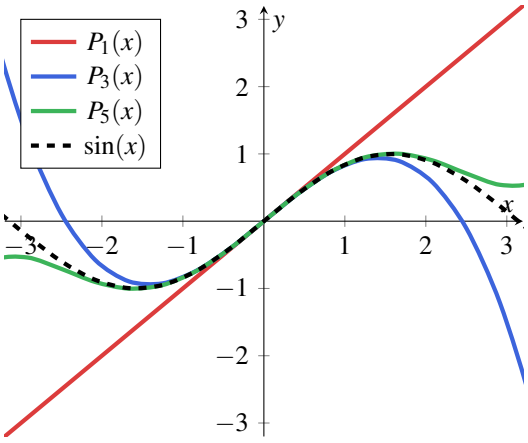

Figure 3.1: The function $f(x) = \sin(x)$ and its Taylor polynomials P_1, P_3 and P_5 of degrees $n \in \{1,3,5\}$. Taylor polynomials are defined in (3.1) and computed for the sinusoid in Example 3.2. Observe that increasing the degree of the Taylor polynomial improves the accuracy of the approximation $f(x) \approx P_n(x)$. The polynomial P_1 is a decent approximation when $|x| < 1$ whereas P_5 works well for $|x| < 2.5$.

and the derivative of $\cos(x)$ is $-\sin(x)$ we get

$$\begin{aligned}
f^{(0)}(x) = f(x) = \sin(x) &\quad\Longrightarrow\quad f^{(0)}(0) = 0,\\
f'(x) = \cos(x) &\quad\Longrightarrow\quad f'(0) = 1,\\
f''(x) = -\sin(x) &\quad\Longrightarrow\quad f''(0) = 0,\\
f^{(3)}(x) = -\cos(x) &\quad\Longrightarrow\quad f^{(3)}(0) = -1,\\
f^{(4)}(x) = \sin(x) &\quad\Longrightarrow\quad f^{(4)}(0) = 0,\\
f^{(5)}(x) = \cos(x) &\quad\Longrightarrow\quad f^{(5)}(0) = 1.
\end{aligned}$$

By plugging these values in (3.1) we obtain the Taylor polynomials

$$\begin{aligned}
P_1(x) &= f(x_0) + f'(x_0)(x - x_0) = f(0) + f'(0)x = x,\\
P_3(x) &= f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f^{(3)}(0)}{3!}x^3 = x - \frac{1}{6}x^3,\\
P_5(x) &= f^{(0)}(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f^{(3)}(0)}{3!}x^3 + + \frac{f^{(4)}(0)}{4!}x^4 + \frac{f^{(5}(0)}{5!}x^5\\
&= x - \frac{1}{6}x^3 + \frac{1}{120}x^5.
\end{aligned}$$

These polynomials are depicted in Figure 3.1.

From Example 3.2 and Figure 3.1 we observe that increasing the degree of the Taylor polynomial appears to make it a better approximation to f. Taylor's theorem familiar from basic analysis puts this observation on rigorous footing.

Theorem 3.3 (Taylor's theorem)**.** *Suppose that $f\colon [a,b] \to \mathbb{R}$ is continuous on $[a,b]$ and $n+1$ times differentiable on (a,b). Let $x_0 \in (a,b)$. Then for each $x \in [a,b]$ we have*

$$f(x) - P_n(x) = R_{n+1}(x),$$

where the remainder term *R_{n+1} is given by*

$$R_{n+1}(x) = \frac{f^{(n+1)}(c_x)}{(n+1)!}(x - x_0)^{n+1}$$

for some c_x between x_0 and x.

The number c_x in Taylor's theorem depends on x and n. We can use big-O notation from Section 2.4 to rewrite Taylor's theorem if the $(n+1)$th derivative of f is uniformly bounded.

Corollary 3.4. *Suppose that $f\colon [a,b] \to \mathbb{R}$ is continuous on $[a,b]$ and $n+1$ times differentiable on (a,b). Let $x_0 \in (a,b)$. If there is a constant $L \geq 0$ such that $|f^{(n+1)}(x)| \leq L$ for all $x \in (a,b)$, then*

$$R_{n+1}(x) = O(|x - x_0|^{n+1}) \quad \text{as} \quad x \to x_0. \tag{3.2}$$

Proof. Taylor's theorem and the assumption $|f^{(n+1)}(x)| \leq L$ for all $x \in (a,b)$ yield

$$|R_{n+1}(x)| = \left|\frac{f^{(n+1)}(c_x)}{(n+1)!}(x - x_0)^{n+1}\right| = \frac{|f^{(n+1)}(c_x)|}{(n+1)!}|x - x_0|^{n+1} \leq \frac{L}{(n+1)!}|x - x_0|^{n+1}$$

for all $x \in (a,b)$. Because $x_0 \in (a,b)$, there is $\delta > 0$ such that $(x_0 - \delta, x + \delta) \subset (a,b)$, so that the above inequality holds for all $x \in (x_0 - \delta, x + \delta)$. Therefore we can select $f(x) = R_{n+1}(x)$, $g(x) = |x - x_0|^{n+1}$ and $C = L/(n+1)!$ in Definition 2.11. This proves (3.2). □

By selecting $x = x_0 + h$ (so that $x - x_0 = h$) we can write (3.2) in the alternative form

$$R_{n+1}(x_0 + h) = O(|h|^{n+1}) \quad \text{as} \quad h \to 0$$

that shows more clearly how the remainder term tends to zero as one considers points closer and closer (i.e., $h \to 0$) to the expansion point x_0.

Example 3.5. It is important to understand that remainder term R_{n+1} can be expected to be small only in the vicinity of x_0. For example, consider the Taylor polynomial of degree $n = 2$ at the point $x_0 = 1$ for the function $f(x) = x^4 + 2x^2$. Because f is defined on the whole of $\mathbb{R}$, we can pick any interval that contains x_0 and use that in Corollary 3.4. Let us pick $[a, b] = [0, 2]$. Then (as $n + 1 = 3$)

$$|f^{(3)}(x)| = \left| \frac{\mathrm{d}^2}{\mathrm{d}x^2}(4x^3 + 4x) \right| = \left| \frac{\mathrm{d}}{\mathrm{d}x}(12x^2 + 4) \right| = |24x| \leq 48 \quad \text{for all} \quad x \in [0, 2].$$

Therefore we can apply Corollary 3.4 with $n = 2$, $x_0 = 1$ and $L = 48$ to conclude that

$$f(x) - P_2(x) = R_3(x) = O(|x - 1|^3) \quad \text{as} \quad x \to 1,$$

which means that the error of the degree two Taylor polynomial P_2 is *cubic* as $x \to x_0 = 1$. However, something completely different happens as one moves away from $x_0 = 1$. Because $n = 2$, we see from the definition of the Taylor polynomial in (3.1) that P_2 is a polynomial of degree two, meaning that $P_2(x) = \alpha x^2 + \beta x + \gamma$ for certain $\alpha, \beta, \gamma \in \mathbb{R}$. Since $f(x) = x^4 + 2x^2$, this means that

$$f(x) - P_2(x) = x^4 + (2 - \alpha)x^2 - \beta x - \gamma,$$

which is a polynomial of degree 4 with a positive leading coefficient (i.e., the degree 4 term has coefficient 1). This means that

$$f(x) - P_2(x) \to \infty \quad \text{as} \quad x \to \infty \text{ or } x \to -\infty.$$

Consequently, the error can be made arbitrarily large by looking at x that is sufficiently far from $x_0 = 1$.

3.2 Linearisation

The case $n = 1$ in Taylor expansion (Definition 3.1) is one of the main tools to construct numerical methods.

Definition 3.6 (Linear approximation)**.** Let $f\colon (a, b) \to \mathbb{R}$ be a once differentiable function at a point $x_0 \in (a, b)$. The *linear approximation* to f at x_0 is the linear polynomial

$$L(x) = P_1(x) = f(x_0) + f'(x_0)(x - x_0). \tag{3.3}$$

The process of computing L and using it the approximate f is called *linearisation*.

Linear approximation is preferred over higher degree Taylor polynomials because it is so easy to compute: only $f(x_0)$ and $f'(x_0)$ are required. In applications it is often cumbersome or even impossible to obtain derivatives of higher orders than this. Moreover, linearisation easily generalises for multivariate functions $f\colon \mathbb{R}^d \to \mathbb{R}$ via the *gradient* and results in numerical methods that are straightforward to implement and understand.

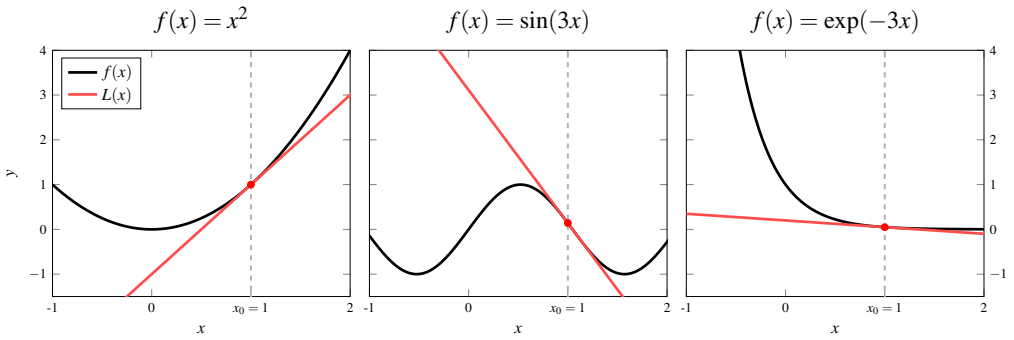

Figure 3.2: Linear approximations (red) to the three functions $f(x) = x^2$, $f(x) = \sin(3x)$ and $f(x) = \exp(-3x)$ at the point $x_0 = 1$. Observe that the linear approximations coincide with the *tangent lines* at $x_0 = 0$. That is, the linear approximation is the line whose slope coincides with the derivative of f at x_0.

Example 3.7. Let us form linear approximations to the functions

$$f_1(x) = -x^2, \quad f_2(x) = \sin(\tfrac{\pi}{2}x) \quad \text{and} \quad f_3(x) = e^x - x^2$$

at the point $x_0 = 1$. For f_1 we get

$$f_1(1) = -1 \text{ and } f_1'(1) = -2 \quad \Longrightarrow \quad L(x) = f(1) + f'(1)(x-1) = -2x + 1.$$

For f_2 we get [recall from trigonometry that $\sin(\frac{\pi}{2}) = 1$]

$$f_2(1) = 1 \text{ and } f_2'(1) = 0 \quad \Longrightarrow \quad L(x) = f(1) + f'(1)(x-1) = 1.$$

For f_3 we get

$$f_3(1) = e - 1 \text{ and } f_3'(1) = e - 2 \quad \Longrightarrow \quad L(x) = f(1) + f'(1)(x-1) = (e-2)x + 1.$$

Figure 3.2 plots linearisations at $x_0 = 1$ three similar functions.

Example 3.7 and Figure 3.2 show that linearisation corresponds to drawing a line that (a) passes through $(x_0, f(x_0))$ and (b) whose slope equals the derivative $f'(x_0)$.

Proposition 3.8. *The linear approximation is the unique line that passes through* $(x_0, f(x_0))$ *and whose slope equals* $f'(x_0)$.

Proof. Every line is given by the equation $y = \alpha x + \beta$ for some $\alpha, \beta \in \mathbb{R}$, where α is the slope of the line. That a line passes through $(x_0, f(x_0))$ means that $f(x_0) = \alpha x_0 + \beta$. That the slope of the line equals $f'(x_0)$ means that $\alpha = f'(x_0)$. Therefore

$$f(x_0) = \alpha x_0 + \beta = f'(x_0)x_0 + \beta \quad \Longrightarrow \quad \beta = f(x_0) - f'(x_0)x_0.$$

Therefore any line specified in the proposition has $\alpha = f'(x_0)$ and $\beta = f(x_0) - f'(x_0)x_0$. That is, there is only one line that passes through $(x_0, f(x_0))$ and whose slope equals $f'(x_0)$. The equation for this line is

$$y = \alpha x + \beta = f'(x_0)x + f(x_0) - f'(x_0)x_0 = f(x_0) + f'(x_0)(x - x_0),$$

which is exactly the linear approximation in (3.3). □

From Figure 3.2 it is apparent that linearisation gives an approximation that is good for x close to x_0 but that can degrade quickly in quality when one moves away from x_0. The following proposition states that the error of linear approximation has *quadratic* order as $x \to x_0$ if f is twice differentiable.

Proposition 3.9. *Suppose that $f\colon [a,b] \to \mathbb{R}$ is continuous on $[a,b]$ and twice differentiable on (a,b). Let $x_0 \in (a,b)$. If there is a constant $L \geq 0$ such that $|f''(x)| \leq L$ for all $x \in (a,b)$, then*

$$f(x) - L(x) = O(|x - x_0|^2) \quad \text{as} \quad x \to x_0.$$

Proof. Because $L(x) = P_1(x)$, the claim follows by setting $n = 1$ in Corollary 3.4. □

3.3 Array programming

Array programming, or *vectorisation*, refers a style of programming where mathematical operations are applied to an array of values at once, rather than to each value individually (i.e., via a loop). Array programming tends to be significantly more *concise* and *easier to read* than an equivalent construction based on loops. In many high-level programming languages, such as Python and MATLAB, array programming is also *significantly faster* than loops and should therefore be used whenever possible. Suppose that we want to square (i.e., compute x^2) for ten million x between 0 and 1 with Python. For array programming we use the Python library `numpy`.

```
>>> import numpy as np
>>> import time
>>> # an array of ten million numbers between 0 and 1
>>> a = np.linspace(start=0, stop=1, num=int(1e7))
>>> # use loop to square each number
>>> start1 = time.time()
>>> b1 = [x**2 for x in a]
>>> end1 = time.time()
>>> # square each number using array programming
>>> start2 = time.time()
>>> b2 = a**2
>>> end2 = time.time()
>>> # print elapsed times
>>> print(end1 - start1)
0.5299568176269531
>>> print(end2 - start2)
0.007016181945800781
```

The elapsed times depend on hardware and vary slightly from one run to the next. But what remains constant is that the second approach based on array programming is much faster (in this case by approximately two orders of magnitude).

Example 3.10. Monte Carlo integration is a straightforward technique to approximate the integral of a function. Suppose that we want approximate the integral $\int_0^1 f(x)\,\mathrm{d}x$ for some integrand function $f\colon [0,1] \to \mathbb{R}$. In Monte Carlo integration we draw n numbers $x_1, \ldots, x_n$ randomly on $[0,1]$ and approximate the integral as the average of integrand evaluations at these points:

$$\int_0^1 f(x)\,\mathrm{d}x \approx \frac{1}{n}\sum_{i=1}^{n} f(x_i). \tag{3.4}$$

In programming languages that support array programming, no loops are needed to compute the Monte Carlo approximation in (3.4).

In Python the Monte Carlo approximation ($n = 200$) to the integral of $f(x) = x^{3.4} + \sin(x)$ could be computed as follows:

```
>>> import numpy as np
>>> rng = np.random.default_rng()
>>> # draw n = 200 random numbers on [0, 1]
>>> n = 200
>>> samples = rng.random(size=20)
>>> # evaluate f at these point and compute average
>>> evals = np.power(samples, 3.4) + np.sin(samples)
>>> mc_approx = np.sum(evals) / n
>>> print(mc_approx)
0.692222065735476
```

The final result will vary from one run to the next because the 200 samples are random. In MATLAB this could be done as follows:

```
>>> # draw n = 200 random numbers on [0, 1]
>>> n = 200;
>>> samples = rand(n, 1);
>>> # evaluate f at these point and compute average
>>> evals = samples.^(3.4) + sin(samples);
>>> mc_approx = sum(evals) / n
mc_approx =

    0.6686
```

Here we used MATLAB's elementwise power signified by dot to compute $x^{3.4}$ for every sample x.

4 Numerical Integration

4.1 Midpoint Rule

4.2 Trapezoidal Rule

4.3 Simpson's Rule

4.4 Error Estimates

5 Solutions to Systems of Equations

5.1 Newton's Method

5.2 Fixed-Point Method

5.3 Direct and Iterative Methods for Linear Systems

5.4 Newton's Method for Linear Systems

5.5 Jacobi Method for Linear Systems

6 Differential Equations

6.1 Euler's Method

6.2 Runge–Kutta Methods

7 Optimisation

7.1 Basic Concepts in Optimisation

7.2 Gradient Descent

7.3 Automatic Differentiation