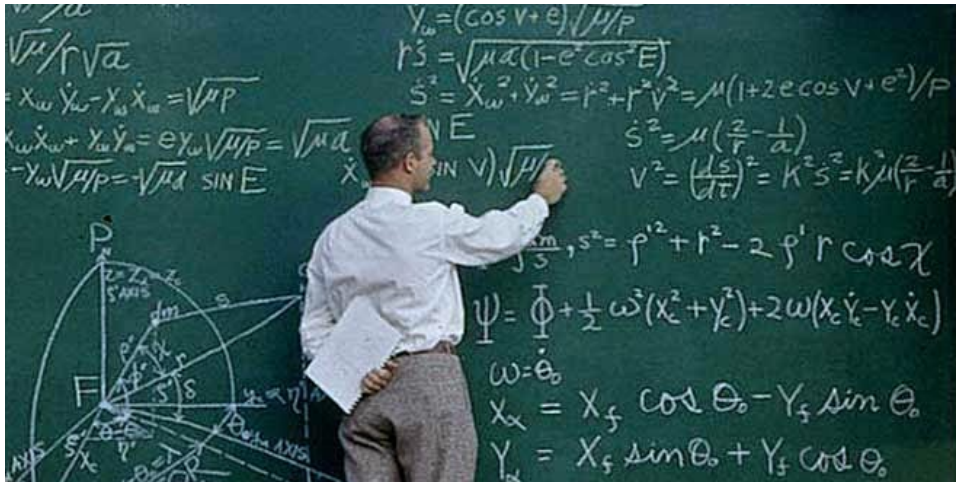
# Modelling and Simulation in Physics

HAP708P, Faculté des Sciences de Montpellier
Felix Brümmer (felix.bruemmer@umontpellier.fr)

# Introduction

# Computational physics

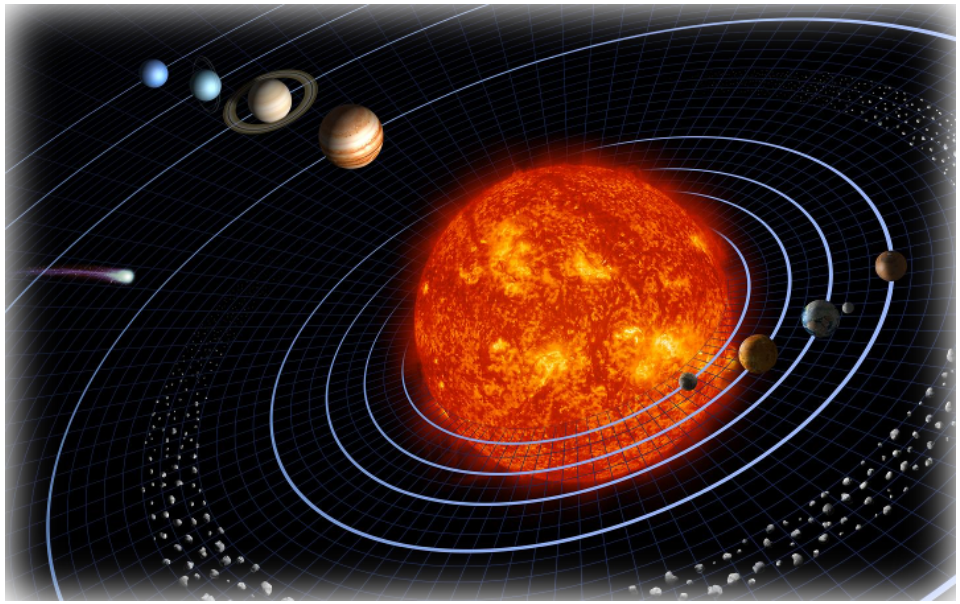
**A common situation** in physics: The equations describing a physical system are known but cannot be solved analytically.



- Exact solutions only exist for a few exceptional problems
  (highly symmetric systems, few degrees of freedom, no dissipation...)
- Controlled approximations sometimes possible for systems sufficiently close to an exactly solvable one
- Generic systems typically require numerical methods!

# Computational physics

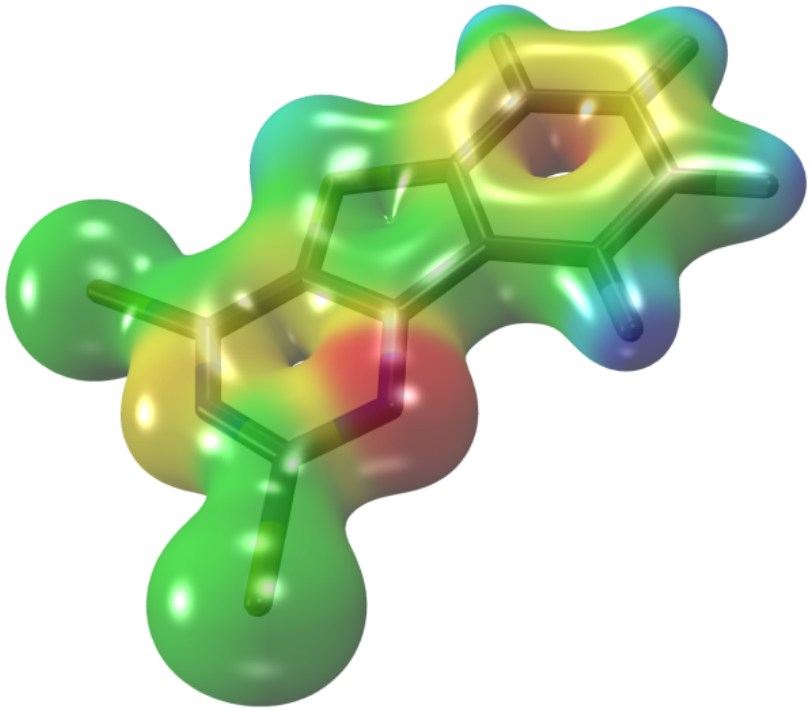
**Example: celestial mechanics**



- Kepler problem: two point masses, potential $V \sim \frac{1}{r}$:
  exactly solvable (trajectories = conic sections).
- Solar system: $n$-body problem ($n > 2$), but gravitational forces between planets small compared to gravitational field of the sun
  $\rightarrow$ can obtain analytic results from perturbation theory
- Generic $n$-body problem ($n > 2$), all masses of the same order
  $\rightarrow$ must solve equations of motion numerically

# Computational physics

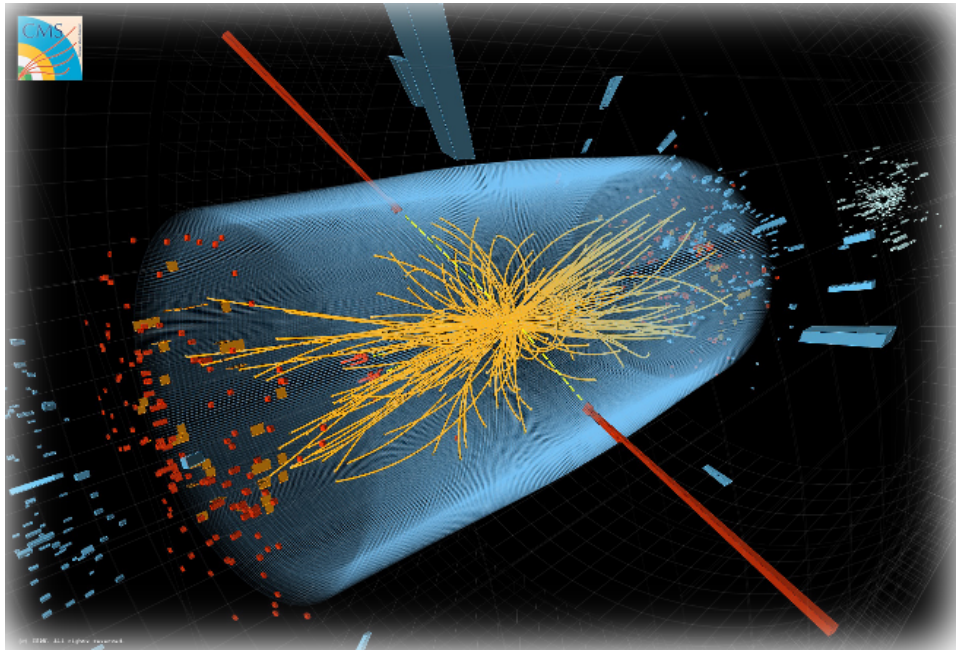
**Example: quantum chemistry**



- Goal: Solve the Schrödinger equation for an entire molecule
- One electron, one nucleus $\rightarrow$ hydrogen-like atom, exact solution in quantum mechanics
- Several electrons $\rightarrow$ numerical methods (Hartree-Fock, post-HF, DFT...)

# Computational physics

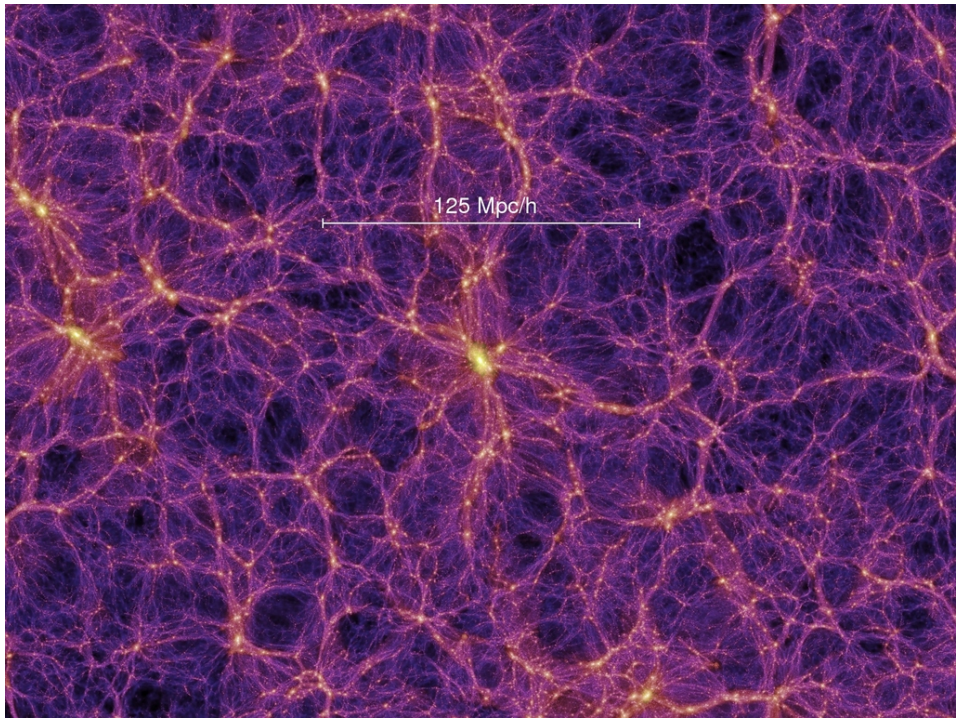
**Example: elementary particle physics**



- Elementary particles (excitations of quantum fields) without interactions: theory exactly solvable
- Particles with weak interactions (quantum electrodynamics...): perturbation theory
- Particles charged under the strong nuclear force at low energies
  $\rightarrow$ numerical methods: lattice field theory

# Computational physics: Some 21st century examples

**Cosmic structure formation** → Springel et al. 2005




Simulation of the dark matter distribution in the universe, starting from primordial density fluctuations: $10^{10}$ "particles" interacting via Newtonian gravity, computing time = 1 month on a supercomputer

# Computational physics: Some 21st century examples

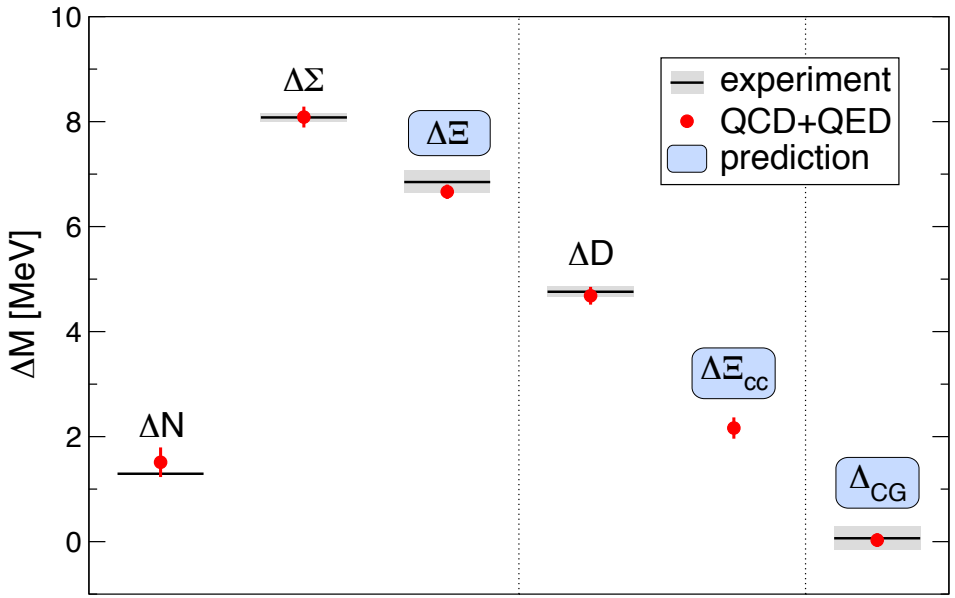
**Computational general relativity** → Ossokine/Buonanno/Dietrich/Haas, SXS project 2017

bh.mp4

Gravity wave emission from two colliding black holes, event GW170104 observed in 2017 by the LIGO experiment

# Computational physics: Some 21st century examples

**Lattice quantum field theory** → Borsanyi et al. 2014




First ab-initio calculation of the proton-neutron mass difference $\Delta N$ (60 TB of simulation data)

# Computational physics: Some 21st century examples

**Heavy ion collisions** → Models and Data Analysis Initiative, https://madai-public.cs.unc.edu/

himovie.mov

Simulation of two Au ions colliding at an energy of 200 GeV at the Relativistic Heavy Ion Collider RHIC

# Overview of this course

**Contents: Algorithms for computational physics**

- Numerical error and algorithmic complexity
- Numerical integration and differentiation
- Ordinary differential equations
- Partial differential equations (finite-difference methods)
- Monte-Carlo methods

**Requirements**:

- Knowledge of physics and mathematics at the Physics Bachelor's level ("Licence de Physique")
- Good programming skills
- Previous experience with Python, even if Python is not your "native programming language" → Hervé Wozniak's lectures and tutorials

Up to you to revise these subjects independently where necessary

# Overview of this course

**Course materials:**

- These slides, available on Moodle
- Other lecture notes, e.g. by A. Palacios@UM (this course until 2015; in French)
- Pedagogical textbook: "Computational physics" by M. Newman, CreateSpace 2013.
- Comprehensive textbook: "Numerical recipes in C++ (3rd ed.)" by W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Cambridge Univ. Pr. 2007

Complementary material (the Python 3 language, root-finding methods, numerical linear algebra and applications...):

- Lecture notes for HAP608P "Programmation pour la physique" (L3 level, in French)

To help you with the exercises, and to encourage you to modify and experiment with the algorithms discussed here:

- All example programs on these slides are also available for download on Moodle

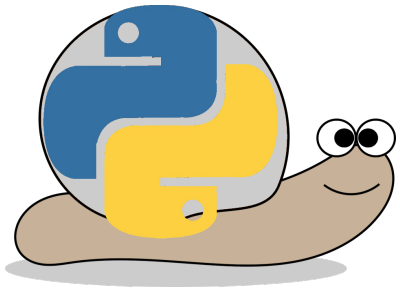
# Computational physics with Python


**The Python 3 programming language**:

- easy to learn, straightforward to read
- widespread, many possible areas of application
- "batteries included": comprehensive and versatile standard library
- (essentially) an interpreted, not a compiled language ⇒ programs are high-level, easily portable
- supports various programming paradigms: procedural programming, object-oriented programming, functional programming...

# Computational physics with Python

**Python's main weakness**: programs are slow, not easy to optimize
$\Rightarrow$ not ideally suited for high-performance computations



For a research project in computational physics with intense demands on computing resources, one would typically prefer a compiled language (C++, FORTRAN...)

Here we use Python for its pedagogical qualities. The goal of this course is to understand how numerical algorithms work. You should then (hopefully) be able to implement them in any language of your choice if needed.

# Numerical error, stability, algorithmic complexity

# In this chapter:

- Representing numerical data in Python
- Numerical error
- Numerical stability
- Algorithmic complexity

# Python's representation of numerical data

A finite computer cannot possibly provide infinite computing resources:

- Numbers represented with finite precision
  → rounding error
  → numerical instabilities if errors accumulate
- Computing time, memory and bandwith are limited:
  → approximate results, truncation error
  → limits on the maximal size of feasible tasks

# Python's representation of numerical data

Python provides three basic numerical data types:

- integer numbers (`int`)
- real floating-point numbers (`float`)
- complex floating-point numbers (`complex`)

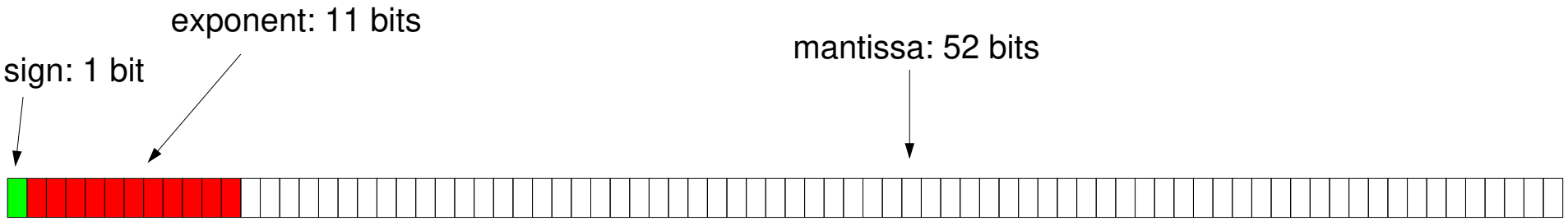
Unlike most other programming languages, there is (theoretically) no limit to the size of an `int` in Python: arbitrary-precision arithmetic.
In practice it is of course limited by the machine's memory.

A `float` is a fixed-precision data type of 8 bytes = 64 bits, as specified in the "double precision" norm IEEE754.

A `complex` corresponds to two `float`, one each for the real and imaginary parts.

# Double-precision floating-point numbers

The meaning of the 64 bits of a `float`:




- The exponent $E$ can represent $2^{11} = 2048$ different numbers, chosen by convention to be between $-1022$ and $1023$. The two remaining values have a special meaning.
- With the $b_0 \dots b_{51}$ bits of the mantissa and the sign bit $s$, the numerical value is

$$(-1)^s \left(1 + \sum_{n=1}^{52} b_{52-n} 2^{-n}\right) \cdot 2^E \,.$$

- Absolute values between $2^{-1022} \approx 10^{-308}$ and $2^{1024} \approx 10^{308}$ (and $0$) with a precision of $53 \log_{10} 2 \approx 16$ decimals.
- When the absolute value of a variable becomes greater than $10^{308}$: overflow, it is set to the special value `inf` (infinity).
- When it becomes smaller than $10^{-308}$: underflow, it is set to zero.

## Exercise

Write two versions of a program which calculates the factorial $x!$ of a given number $x$. In the first version, all numerical data is represented by variables of the type `int`, and in the second version, by variables of the type `float`. What do you obtain when trying to calculate $200!$ with both programs? Explain what you observe.

# Numerical error: Rounding error

Relative precision = 16 digits

**Example**

In Python: $\sqrt{2} = 1.4142135623730951$
In reality: $\sqrt{2} = 1.41421356237309504\,88\ldots$
Rounding error: $0.00000000000000005\,12\ldots$

$3.0$ and $2.999999999999999$ are "the same number" in double precision!

But Python doesn't know that ⇒ don't test equality of two `floats` like this:

```
x = 1.1 + 2.2 # x = 3.3000000000000003
if x == 3.3:  # False!
    do_something_with(x)
```

but rather test if they are equal to within the expected precision:

```
precision = 1.0E-15
if abs(x - 3.3) < precision:    # better
    do_something_with(x)
```

# Numerical error: Information loss

Problem when adding or subtracting numbers of (vastly) different order of magnitude.

**Example**: $x = 1$, $y = 1 + 10^{-14}\sqrt{2}$, therefore $10^{14}(y - x) = \sqrt{2}$.

In Python: $\sqrt{2} = 1.414213562373095\cdots$

$$x = 1.000000000000000\cdots$$
$$y = 1.000000000000014\cdots$$
$$y - x = 1.4\cdots\cdots\cdots\cdots\cdots\cdots 10^{-14}$$

Explicitly: the program

```
x = 1.0
root2 = 2**0.5
y = 1.0 + 1.0E-14 * root2
print(root2)
print(1.0E14 * (x - y))
```

will produce the output

```
1.4142135623730951
1.4210854715202004
```

$\Rightarrow$ rounding error already in the 3rd decimal!

## Exercise

- Write a program which calculates the solutions of the second-order equation $ax^2 + bx + c = 0$ by the standard formula,

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}, \qquad \Delta = b^2 - 4ac\,.$$

  What do you obtain for $a = c = 0.001$ and $b = 1000$?
- Show that the two solutions can also be written

$$x = \frac{2c}{-b \mp \sqrt{\Delta}}\,.$$

  Modify your program to calculate the solutions also with the second formula, and run it with $a = c = 0.001$ and $b = 1000$. What do you obtain? Explain your results.

# Numerical error: Truncation error

Any quantity defined by a limit may not be represented exactly on the computer.

**Example:**

$$e = \lim_{N \to \infty} \sum_{n=0}^{N} \frac{1}{n!}$$

Impossible to sum infinitely many terms in practice, need to stop at some $N$

$\Rightarrow$ truncation error

In reality: $e = 2.71828182845904\ldots$
With $N = 10$: $e \approx 2.71828180114638$
Truncation error: $0.00000002731266\ldots$

# Numerical error: Absolute and relative error

For any numerical approximation $\tilde{x}$ of some quantity $x$, we define the absolute error $\epsilon(x, \tilde{x})$,

$$\epsilon(x, \tilde{x}) = |x - \tilde{x}|$$

and the relative error $\epsilon_r(x, \tilde{x})$

$$\epsilon_r(x, \tilde{x}) = \frac{|x - \tilde{x}|}{|x|} = \epsilon\left(1, \frac{\tilde{x}}{x}\right) .$$

The exact values of $\epsilon$, $\epsilon_r$ are generally unknown (or else there would be no need for numerical approximations). In practice, one supposes that they are random variables following a normal (Gaussian) probability distribution.

Denote by $\sigma$ the standard deviation of $\epsilon$ and by $C$ the standard deviation of $\epsilon_r$,

$$\sigma = C|x| .$$

E.g. for the rounding error due to the limits of double-precision floating point arithmetic, $C \approx 10^{-16}$.

Error analysis aims to estimate $C$ (or $\sigma$) in order to estimate the typical size of $\epsilon_r$ (or $\epsilon$).

# Numerical error: Error propagation

From standard probability theory (just as for experimental uncertainties):

- For the sum $y = x_1 + x_2$ of two quantities $x_1$ and $x_2$ with uncorrelated uncertainties $\sigma_1$ and $\sigma_2$, one has $\sigma_y^2 = \sigma_1^2 + \sigma_2^2$ and therefore

$$\sigma_y = \sqrt{\sigma_1^2 + \sigma_2^2}\,.$$

- For a product $y = x_1 x_2$, the squared relative uncertainties must be added, hence

$$C_y = \sqrt{C_1^2 + C_2^2}$$

- General case: Let $y = y(x_1, \ldots x_n)$, then the uncertainty for $y$ is

$$\sigma_y = \sqrt{\sum_{i=1}^{n} \left( \frac{\partial y}{\partial x_i} \sigma_i \right)^2}$$

# Numerical stability

An algorithm is called

- unstable: small variations in the input data can produce large variations in the output data
  $\Rightarrow$ the numerical error is amplified
- stable: small variations in the input data will not lead to large variations in the output data
  $\Rightarrow$ the numerical error remains of the same order or is even diminished

The precise definition of stability depends on the algorithm under study.

It is obviously best to use stable methods when possible. But often they come at a price: they may be more difficult to implement and/or computationally more expensive.

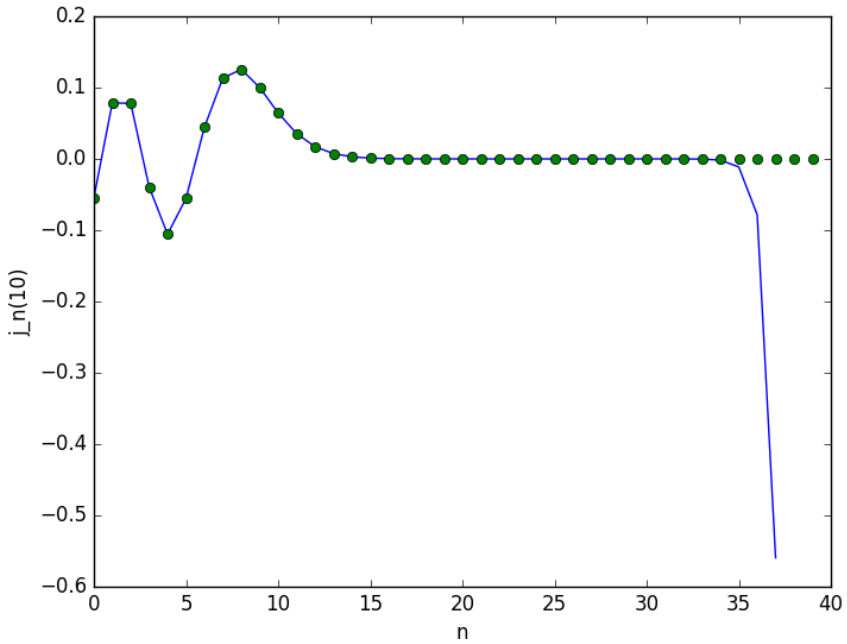
# Numerical stability

Most important for algorithms using a feedback loop: if the error is amplified at each iteration, it may eventually dominate the result.

**Example:** Numerical evaluation of spherical Bessel functions of the first kind (solutions of the radial free Schrödinger equation in spherical coordinates)

$$j_0(x) = \frac{\sin x}{x}\,, \quad j_1(x) = \frac{\sin x}{x^2} - \frac{\cos x}{x}\,, \qquad j_n(x) = \frac{2n-1}{x} j_{n-1}(x) - j_{n-2}(x)$$

Plotting $j_n(10)$ as a function of $n$ (numerical value found recursively, exact value):



# Numerical stability

**Explanations:**

- The recurrence relation has a second solution (the spherical Bessel functions of the second kind $k_n(x)$) which grows monotonically as a function of $n$ for $n > x$
- Numerical error $\Rightarrow$ instead of just $j_n(x)$, the computer really calculates some linear superposition of $j_n(x)$ and $k_n(x)$
- The $k_n(x)$ component is initially small (due to truncation/rounding errors when computing $j_0$ and $j_1$). But it grows at each iteration.
- Finally, for large $n$, the numerical solution is dominated by the growing $k_n(x)$ component.

**Possible solution:**

Use the recurrence relation backwards (for decreasing $n$); normalize the result by $j_0$. Stable.

# Analysis of algorithms

**Some typical computational problems**:

- Evaluate a function with $n$-digit precision
- Find the solution of an equation with a precision of $1/n$
- Solve a system of $n$ equations at fixed precision
- Diagonalize an $n \times n$ matrix
- Sort a list of $n$ elements
- Find some given element within a list of $n$ elements
- ...

**Time complexity** as a measure of an algorithm's efficiency:
How does the run-time $T(n)$ depend on the "characteristic problem size" $n$?

(Other measures could be: consumption of memory $M(n)$ or network bandwith $B(n)\ldots$)

In particular: study the asymptotic behaviour of $T(n)$ for large $n$.

# Analysis of algorithms: Asymptotic growth, Landau symbols

Let $f : \mathbb{R}_+ \to \mathbb{R}_+$ be a monotonically increasing reference function.

We say of some other function $g : \mathbb{R}_+ \to \mathbb{R}_+$ that

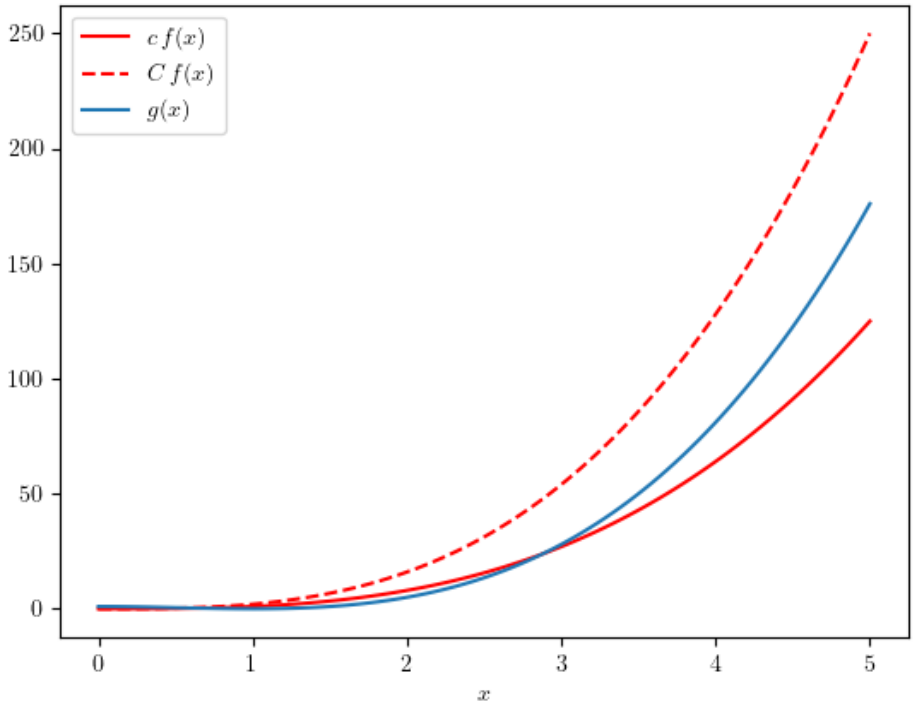
- $g \in \mathcal{O}(f)$
  $\Leftrightarrow$ $g$ grows at most as fast as $f$ asymptotically
  $\Leftrightarrow$ there exists a constant $C > 0$ s.t. for sufficiently large $x$, $g(x) \leq C\, f(x)$.
- $g \in \Omega(f)$
  $\Leftrightarrow$ $g$ grows at least as fast as $f$ asymptotically
  $\Leftrightarrow$ $\exists\, c > 0,\, x_0 > 0\, \forall\, x > x_0 \;:\; c\, f(x) \leq g(x)$
- $g \in \Theta(f)$
  $\Leftrightarrow$ $g$ grows as fast as $f$ asymptotically
  $\Leftrightarrow$ $g \in \mathcal{O}(f)$ and $g \in \Omega(f)$
  $\Leftrightarrow$ $c\, f(x) \leq g(x) \leq C\, f(x)$ for suitable constants $c$ and $C$ and sufficiently large $x$

# Analysis of algorithms: Asymptotic growth, Landau symbols

**Example**: Consider $f(x) = x^3$.

The function $g(x) = 2\,x^3 - 3\,x^2 + 1$ is in $\Theta(x^3)$

(for large $x$, can neglect $-3\,x^2$ and $1$ w.r.t. $2\,x^3$; $2\,x^3 \in \Theta(x^3)$ since constant factors don't matter)



# Asymptotic growth, Landau symbols

## Exercise

Show that for any positive constants $a, b, c$, one has

$$\Theta(\log(x^a)) = \Theta(\log_b x) = \Theta(\log(cx)) = \Theta(\log(x)) \,.$$

# Analysis of algorithms

**Goal of the analysis of algorithms**: Characterize the asymptotic growth of the function $T(n)$ = run-time as a function of problem size; how does $T(n)$ behave at large $n$?

$\Rightarrow$ count the number of elementary steps necessary to carry out the algorithm

Elementary step = assignment, arithmetic operation on a `float`, comparison, branching... any simple instruction that does not depend on $n$

**Remark 1:** In computer science, it is common to use $\mathcal{O}$ instead of $\Theta$ even though, strictly speaking, their meaning is different. E.g. if $T(n) \in \Theta(n \log n)$, on frequently finds the statement that "$T(n) \in \mathcal{O}(n \log n)$" (or even, by abuse of notation, "$T(n) = \mathcal{O}(n \log n)$"). Correct (since $\Theta \subset \mathcal{O}$) but imprecise.

**Remark 2:** For our discussion, we defined $\mathcal{O}$ in the limit where the argument of a function tends to infinity. By contrast, in calculus one often defines $\mathcal{O}$ in the limit where it tends to zero (see next chapter on integrals and derivatives).

# Analysis of algorithms, example: Linear search

- Input data: a list `L` of length $n$ which contains the element `x`
- Desired output: the position of `x` in `L`
- Algorithm: iterate over `L`, compare each element with `x`, terminate iteration upon equality

```
def linear_search(L, x):
    # use enumerate(L) to obtain a sequence of pairs
    #  (0, L[0]), (1, L[1]), (2, L[2]), etc.
    for index, item in enumerate(L):
        if item == x:
            return index
```

**Analysis**: Count the number of elementary steps for some given $n$.

- Best case: First element = `x`, hence $T(n) =$ const., hence $T(n) \in \Theta(1)$.
- Worst case: Last element = `x`, so need to iterate over the entire list to find `x`, hence $T(n) \propto n$, hence $T(n) \in \Theta(n)$.
- Average case: Need to iterate over half of the list to find $x$, $T(n) \propto \frac{n}{2}$, hence still $T(n) \in \Theta(n)$.

# Analysis of algorithms, second example: Binary search

- Input data: a sorted list `L` of length $n$ which contains the element `x`
- Desired output: the position of `x` in `L`
- Algorithm: compare the element `m` at the center of `L` with `x`. If `m > x`, repeat with the half of the list on the left of `m`. Otherwise, repeat with the half on the right of `m`. Terminate when the remaining sublist contains only a single element.

```
def binary_search(L, x):
  left, right = 0, len(L)      # L[left:right] contains x
  while right - left > 1:      # does it contain >1 element?
    mid = (right + left) // 2 # index of the center
    if L[mid] > x:           # is x in the left half ?
      right = mid            #  -> repeat with L[left:mid]
    else:                    # otherwise it is in the right half
      left = mid             #  -> repeat with L[mid:right]
  return left
```

**Analysis**:

$\log_2 n$ loop iterations $\Rightarrow T(n) \propto \log_2 n$, hence $T(n) \in \Theta(\log(n))$.

# Analysis of algorithms

## Exercise

The following program tests if $n$ is prime. Analyse its run-time complexity: what is the worst-case growth of $T(n)$?

```
def is_prime(n):
    k = 2
    while k**2 <= n:
        if n % k == 0:
            return False
        k += 1
    return True
```

## Exercise

Recall that the matrix product between two $n \times n$ matrices $A$ and $B$ is

$$(A \cdot B)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \, .$$

Analyse the run-time complexity of a routine which calculates the matrix product with this formula as a function of $n$.

# Analysis of algorithms

**Hypothetical example:** Suppose that some algorithm needs a run-time of $T(10) = 10\ \mu$s for some input data of size $n = 10$. Then, for $n > 10$, the run-time will be approximately:

| | $n = 10$ | $n = 20$ | $n = 30$ | $n = 100$ | $n = 1000$ | $n = 10\,000$ |
|---|---|---|---|---|---|---|
| $\Theta(1)$ | 10 $\mu$s | 10 $\mu$s | 10 $\mu$s | 10 $\mu$s | 10 $\mu$s | 10 $\mu$s |
| $\Theta(\log n)$ | 10 $\mu$s | 13 $\mu$s | 15 $\mu$s | 20 $\mu$s | 30 $\mu$s | 40 $\mu$s |
| $\Theta(\sqrt{n})$ | 10 $\mu$s | 14 $\mu$s | 17 $\mu$s | 32 $\mu$s | 100 $\mu$s | 320 $\mu$s |
| $\Theta(n)$ | 10 $\mu$s | 20 $\mu$s | 30 $\mu$s | 100 $\mu$s | 1 ms | 10 ms |
| $\Theta(n^2)$ | 10 $\mu$s | 40 $\mu$s | 90 $\mu$s | 1 ms | 100 ms | 10 s |
| $\Theta(n^3)$ | 10 $\mu$s | 80 $\mu$s | 270 $\mu$s | 10 ms | 10 s | 3 h |
| $\Theta(e^n)$ | 10 $\mu$s | 220 ms | 1.5 h | $10^{26}$ yrs* | $10^{417}$ yrs* | $10^{4326}$ yrs* |

(* age of the universe $\approx 10^{10}$ years)

**Useful orders of magnitude:** Python on an ordinary PC can do $\sim 10^9$ elementary steps in a "reasonable" time ($\sim$ seconds).

Time needed for $10^6$ elementary steps = "instantaneous" ($\ll$ 1s)

Time needed for $10^{12}$ elementary steps = "infinite" ($\gtrsim$ hours)

# Numerical integrals and derivatives

# In this chapter

- The trapezoidal method
- Simpson's method and other Newton-Cotes methods
- Adaptive methods
- Gaussian quadrature
- Numerical first and second derivatives

# Numerical integration

**Goal**: Compute $\int_a^b f(x)\,dx$ for some given function $f$ (which cannot be analytically integrated)

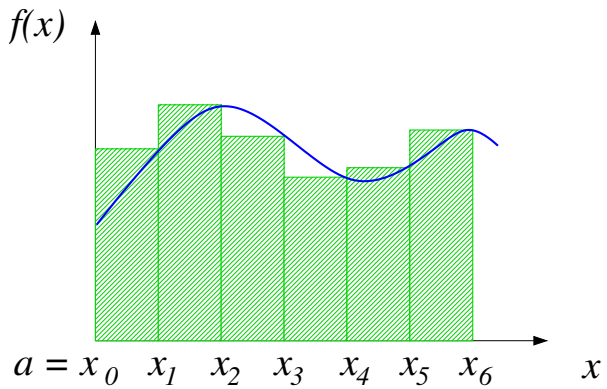
Possible complications ($\rightarrow$ later):

- Improper integrals ($f$ not defined at $a$ or $b$, or $a = -\infty$ or $b = \infty$)
- Singularities or discontinuities within the domain of integration
- Multi-dimensional integrals $\rightarrow$ Monte-Carlo methods, chapter 6

**Definition of the integral** by Riemann sum (here: "right rule")

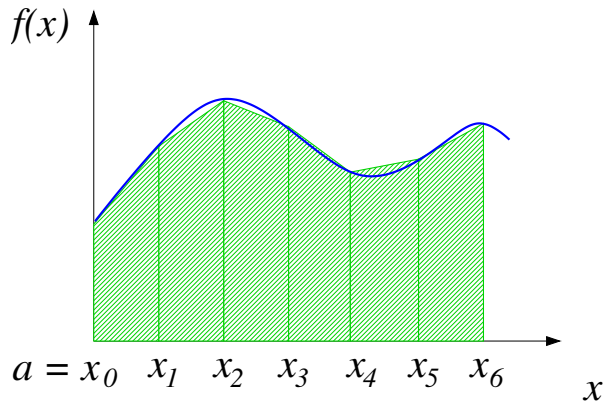
$$\int_a^b f(x)\,dx = \lim_{N\to\infty} \sum_{k=1}^{N} h\, f_k\,, \qquad h = \frac{b-a}{N}\,, \quad f_k = f(x_k)\,, \quad x_k = a + kh$$

Approximate the area between $f(x)$ and the $x$-axis by $N$ rectangles of area $h\, f_k$.



# Newton-Cotes methods: Trapezoid method

**Better**: instead of rectangles, use trapezoids




**Trapezoidal rule**:

$$\int_{x_k}^{x_{k+1}} f(x)\,dx \approx \frac{h}{2}(f_{k+1} + f_k)$$

and therefore

$$\int_a^b f(x)\,dx = \sum_{k=0}^{N-1} \int_{x_k}^{x_{k+1}} f(x)\,dx \approx h\left(\frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{k=1}^{N-1} f_k\right).$$

# Newton-Cotes methods: Trapezoid method

Simple function for calculating integrals with the trapezoid method:

```
def int_trapez(f, a, b, N):
    h = (b - a) / N
    result = f(a)/2 + f(b)/2 # boundary points
    for k in range(1, N):    # interior points
        result += f(a + k*h)
    result *= h
    return result
```

Test:

```
from math import sin, pi
print("I =", int_trapez(sin, 0, pi, 10000))
```

# Error estimate for the trapezoidal rule

Taylor series expansion of $f(x)$ around $x_k$ (notation reminder: $f_k \equiv f(x_k)$)

$$f(x) = f_k + (x - x_k) f'_k + \frac{1}{2}(x - x_k)^2 f''_k + \dots$$

Integrate between $x_k$ and $x_{k+1}$:

$$\begin{aligned}
&\int_{x_k}^{x_{k+1}} f(x)\, \mathrm{d}x \\
&\qquad = f_k \int_{x_k}^{x_{k+1}} \mathrm{d}x + f'_k \int_{x_k}^{x_{k+1}} (x - x_k)\, \mathrm{d}x + \frac{1}{2} f''_k \int_{x_k}^{x_{k+1}} (x - x_k)^2\, \mathrm{d}x + \dots \\
&\qquad = h\, f_k + \frac{1}{2} h^2\, f'_k + \frac{1}{6} h^3\, f''_k + \mathcal{O}(h^4)
\end{aligned}$$

Similarly, for an expansion of $f(x)$ around $x_{k+1}$,

$$\int_{x_k}^{x_{k+1}} f(x)\, \mathrm{d}x = h\, f_{k+1} - \frac{1}{2} h^2 f'_{k+1} + \frac{1}{6} h^3\, f''_{k+1} + \mathcal{O}(h^4)\,.$$

Adding and dividing by 2:

$$\int_{x_k}^{x_{k+1}} f(x)\, \mathrm{d}x = \frac{1}{2} h\, (f_k + f_{k+1}) + \frac{1}{4} h^2\, (f'_k - f'_{k+1}) + \frac{1}{12} h^3\, (f''_k + f''_{k+1}) + \mathcal{O}(h^4)$$

# Error estimate for the trapezoidal rule

Taking the sum over all the slices:

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{k=0}^{N-1} \int_{x_k}^{x_{k+1}} f(x)\,\mathrm{d}x$$

$$= \underbrace{\frac{1}{2}h \sum_{k=0}^{N-1} (f_k + f_{k+1})}_{\text{trapezoidal rule}} + \frac{1}{4}h^2 \left(f'(a) - f'(b)\right) + \frac{1}{12}h^3 \sum_{k=0}^{N-1} \left(f''_k + f''_{k+1}\right) + \mathcal{O}(Nh^4)$$

- All terms $\propto h^2$ cancel out, except $\frac{1}{4}h^2(f'(a) - f'(b))$.
- One can show: Terms $\propto h^4$ also cancel $\Rightarrow$ the $\mathcal{O}(Nh^4)$ terms are in fact $\mathcal{O}(h^4)$.
- The $\propto h^3$ terms correspond to the trapezoidal rule for the integrand $\frac{h^2}{6}f''(x)$:

$$\frac{1}{12}h^3 \sum_{k=0}^{N-1} \left(f''_k + f''_{k+1}\right) = \int_a^b \left(\frac{h^2}{6}f''(x)\right)\mathrm{d}x + \mathcal{O}(h^4) = \frac{h^2}{6}\left(f'(b) - f'(a)\right) + \mathcal{O}(h^4)\,.$$

**Summary:**

$$\int_a^b f(x)\,\mathrm{d}x = \underbrace{\frac{1}{2}h \sum_{k=0}^{N-1} (f_k + f_{k+1})}_{\text{trapezoidal rule}} + \underbrace{\frac{1}{12}h^2 \left(f'(a) - f'(b)\right)}_{\text{leading-order error term}} + \mathcal{O}(h^4)\,.$$

# Error estimate for the trapezoidal rule

**Euler-MacLaurin formula** for truncation error:

$$\boxed{\epsilon \approx \frac{1}{12} h^2 (f'(a) - f'(b))\,.}$$

- Order-$h$ method: The result is exact up to terms of order $h^2$.
- Comparing with rounding error: With a relative precision of $C \sim 10^{-16}$, the errors are comparable when

$$\frac{1}{12} h^2 (f'(a) - f'(b)) \simeq C \int_a^b f(x)\ \mathrm{d}x$$

or, with $h = (b-a)/N$,

$$N \sim (b-a) \sqrt{\frac{f'(a) - f'(b)}{12 \int_a^b f(x)\ \mathrm{d}x}} C^{-1/2}\,.$$

If the prefactor is $\mathcal{O}(1)$, then it takes $N \simeq 10^8$ subdivisions for the truncation error to become negligible. For a reasonable number of subdivisions, the truncation error is dominant.

- Analysis: $1/n$ precision requires at least $\Theta(\sqrt{n})$ elementary steps (provided that evaluating $f(x)$ takes $\Theta(1)$ time — the most optimistic case).

# Error estimate for the trapezoidal rule

**More practical way** to estimate the error: vary the number of points.

Let

- $I$ be the integral's exact value, $I = \int_a^b f(x)\,\mathrm{d}x$
- $N_1$ be the number of slices of witdh $h_1 = (b-a)/N_1$
- $I_1$ be the numerical approximation obtained with the trapezoidal method
- $\epsilon_1$ be the numerical error to first approximation, $I \approx I_1 + \epsilon_1$

Knowing that the trapezoidal method is of order $h$:

$$I = I_1 + \epsilon_1 + \mathcal{O}(h_1^4) = I_1 + c\,h_1^2 + \mathcal{O}(h_1^4)\,, \qquad c = \text{const.}$$

Doubling the number of points, $N_2 = 2\,N_1$ and $h_2 = h_1/2$, one finds similarly

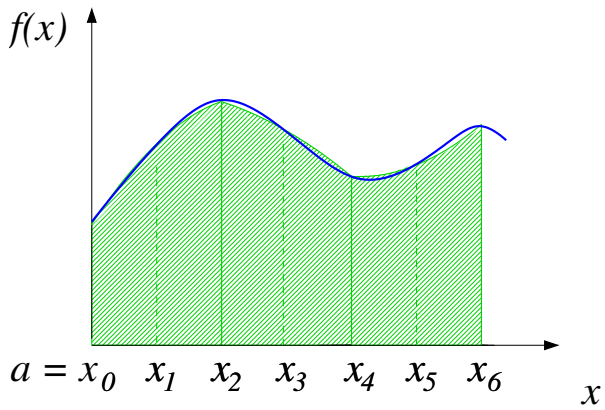
$$I = I_2 + c\,h_2^2 + \ldots$$

and therefore

$$I_2 - I_1 = c\,(h_1^2 - h_2^2) \approx 3c\,h_2^2$$

$$\Rightarrow \qquad \boxed{\epsilon_2 \approx \frac{1}{3}(I_2 - I_1)}$$

# Newton-Cotes methods: Simpson's method




**Even better**: approximate the integrand on every slice neither by a constant (Riemann sum) nor by a straight line (trapezoidal rule) but by a parabola: Simpson's method.

# Newton-Cotes methods: Simpson's method

Quadratic function defined on two consecutive slices, interpolating between the points $(x_{k-1}, f_{k-1})$, $(x_k, f_k)$, and $(x_{k+1}, f_{k+1})$:

$$\left.\begin{array}{rcrcrcl} \alpha\, x_{k-1}^2 & + & \beta\, x_{k-1} & + & \gamma & = & f_{k-1} \\ \alpha\, x_k^2 & + & \beta\, x_k & + & \gamma & = & f_k \\ \alpha\, x_{k+1}^2 & + & \beta\, x_{k+1} & + & \gamma & = & f_{k+1} \end{array}\right\}$$ 3 linear equations, 3 unknowns $\alpha$, $\beta$, $\gamma$

For simplicity: $x_{k-1} = -h$, $x_k = 0$, $x_{k+1} = h$:

$$\begin{aligned} \alpha\, h^2 - \beta\, h + \gamma &= f(-h) \\ \gamma &= f(0) \\ \alpha\, h^2 + \beta\, h + \gamma &= f(h) \end{aligned}$$

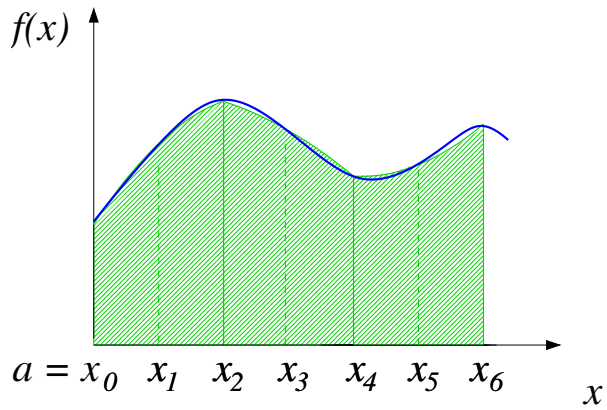
Solution:

$$\gamma = f(0)\,, \quad \beta = \frac{f(h) - f(-h)}{2\,h}\,, \quad \alpha = \frac{f(h) + f(-h) - 2\,f(0)}{2\,h^2}\,.$$

The polynomial $\alpha x^2 + \beta x + \gamma$ is easily integrated analytically:

$$\int_{-h}^{h} \alpha x^2 + \beta x + \gamma \, \mathrm{d}x = \frac{h}{3}\left(f(-h) + 4\,f(0) + f(h)\right)\,.$$

# Newton-Cotes methods: Simpson's method




We have found:

$$\int_{x_{k-1}}^{x_{k+1}} f(x)\,\mathrm{d}x \approx \frac{h}{3}\left(f_{k-1} + 4\,f_k + f_{k+1}\right)$$

And we have

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{\substack{1\le k\le N-1\\ k\ \textbf{odd}}} \int_{x_{k-1}}^{x_{k+1}} f(x)\,\mathrm{d}x$$

$$\Rightarrow \quad \boxed{\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\left(f(a) + f(b) + 4\sum_{\substack{1\le k\le N-1\\ k\ \textbf{odd}}} f_k + 2\sum_{\substack{2\le k\le N-2\\ k\ \textbf{even}}} f_k\right)}$$

**Simpson's rule.**

# Error estimate for Simpson's method

Similar calculation as for trapezoidal method: Euler-MacLaurin formula for Simpson's method,

$$\epsilon \approx \frac{1}{90} h^4 \left( f'''(a) - f'''(b) \right)$$

- Order-$h^3$ method: Result is exact up to terms of order $h^4$.
- The truncation error becomes comparable to the double-precision rounding error for $N \simeq 10\,000$ points. Further increasing $N$ will not increase the precision.
- Converges much more quickly than the trapezoidal method for well-behaved integrands (bounded derivatives...)
- Algorithm analysis: for a target precision of $1/n$,

$$\frac{1}{n} \stackrel{!}{=} \epsilon \propto h^4 \propto \frac{1}{N^4}$$

need to evaluate $f$ at $N \propto n^{1/4}$ points $\Rightarrow$ at least $\propto n^{1/4}$ elementary steps $\Rightarrow$ run-time complexity $\Theta(n^{1/4})$ in the best case.

# Simpson's method

## Exercises

- Just as we did for the trapezoidal method (see p. 48), one may estimate the dominant error term for Simpson's method by doubling the number of points. Show that one obtains the estimate

$$\epsilon_2 \approx \frac{1}{15}(I_2 - I_1)\,.$$

- Write a function `int_simpson(f, a, b, N)` similar to the function `int_trapez`, but using Simpson's method.
- Compute

$$I = \int_0^\pi x^2 \sin x \, \mathrm{d}x$$

  with the trapezoid method and with Simpson's method for $N = 10, 100, 1000, 2000$. Compare with the exact result $I = \pi^2 - 4$. For $N = 2000$, compare the actual numerical error with the error estimate given by the above formula (or rather by the formula of p. 48 for the trapezoid method).
- Implement an adaptive version of Simpson's method (similar to the one presented below for the trapezoid method).

# Newton-Cotes methods of degree $p$

**Generalization:**

- $p$ consecutive slices between $x_k$ and $x_{k+p}$ define a polynomial of degree $p$
- One may therefore approximate

$$\int_{x_k}^{x_{k+p}} f(x)\,\mathrm{d}x \approx \int_{x_k}^{x_{k+p}} \left(c_p x^p + c_{p-1}x^{p-1} + \ldots + c_0\right)\mathrm{d}x$$

  where the coefficients $c_i$ are determined by the $p+1$ linear equations

$$\begin{aligned} c_p x_k^p + \ldots + c_0 &= f_k \\ &\ldots \\ c_p x_{k+p}^p + \ldots + c_0 &= f_{k+p} \end{aligned}$$

- The polynomial can be integrated analytically.

**Result:** Newton-Cotes method of degree $p$.

## Newton-Cotes methods of degree $p$

- $p = 1$: Trapezoid rule,

$$\int_a^b f(x)\,\mathrm{d}x \approx h\left(\frac{1}{2}f(a) + f_1 + f_2 + f_3 + \ldots + f_{N-1} + \frac{1}{2}f(b)\right).$$

- $p = 2$: Simpson's rule,

$$\int_a^b f(x)\,\mathrm{d}x \approx h\left(\frac{1}{3}f(a) + \frac{4}{3}f_1 + \frac{2}{3}f_2 + \frac{4}{3}f_3 + \frac{2}{3}f_4 + \ldots + \frac{4}{3}f_{N-1} + \frac{1}{3}f(b)\right).$$

- $p = 3$: Simpson's $3/8$ rule,

$$\int_a^b f(x)\,\mathrm{d}x \approx h\left(\frac{3}{8}f(a) + \frac{9}{8}f_1 + \frac{9}{8}f_2 + \frac{3}{4}f_3 + \frac{9}{8}f_4 + \frac{9}{8}f_5 + \frac{3}{4}f_6 + \ldots + \frac{3}{8}f(b)\right).$$

- $p = 4$: Boole's rule,

$$\int_a^b f(x)\,\mathrm{d}x \approx h\left(\frac{14}{45}f(a) + \frac{64}{45}f_1 + \frac{8}{15}f_2 + \frac{64}{45}f_3 + \frac{28}{45}f_4 + \frac{64}{45}f_5 + \frac{8}{15}f_6 + \frac{64}{45}f_7 + \ldots + \frac{64}{45}f_{N-1} + \frac{14}{45}f(b)\right).$$

# Newton-Cotes methods

- The $p$-th degree method gives the exact result if the integrand $f$ is itself a polynomial of degree $\leq p$.
  (Even better if $p$ is even: exact method for degrees $\leq p+1$ $\leftarrow$ more difficult to show.)
- In practice: Initially the speed of convergence grows with $p$ if $f$ is "well-behaved", i.e. if $f$ is well approximated by a polynomial; no discontinuities and/or singularities. In geneneral, there exists some optimal $p$ beyond which the polynomial approximation becomes worse ("Runge's phenomenon").
- For discontinuous, rapidly fluctuating or singular integrands: trapezoidal rule may still be the best choice

# Adaptive trapezoid method

Back to the trapezoid method; recall the notation of p. 48:

$$
\begin{aligned}
I &= \int_a^b f(x)\,\mathrm{d}x \\
&= I_i + \epsilon_i + \mathcal{O}(h_i^4) \qquad \text{computed with } N_i \text{ slices of width } h_i = \frac{b-a}{N_i} \\
&= h_i \left( \frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{k=1}^{N_i - 1} f_k \right) + \epsilon_i + \mathcal{O}(h_i^4)\,,
\end{aligned}
$$

Recall also the error estimate: If $N_{i+1} = 2\,N_i$, then

$$
\epsilon_{i+1} \approx \frac{1}{3}\,(I_{i+1} - I_i)\ .
$$

**Adaptive method** to obtain a given precision $\delta$:

- Compute $I_1$ with some initial choice for $N_1$
- Successively double the number of points, $N_{i+1} = 2N_i$, and compute $I_{i+1}$.
  (One may re-use the points calculated previously $\rightarrow$ save computing resources.)
- Compute $\epsilon_{i+1}$. When $|\epsilon_{i+1}| < \delta$, terminate.

# Adaptive trapezoid method

To re-use the points calculated previously, note that

$$I_i = h_i \left( \frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{k=1}^{N_i-1} f(a+kh_i) \right)$$

$$= h_i \left( \frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{\substack{1\le k\le N_i-1 \\ k \text{ odd}}} f(a+kh_i) + \sum_{\substack{2\le k\le N_i-2 \\ k \text{ even}}} f(a+kh_i) \right)$$

We have

$$\sum_{\substack{2\le k\le N_i-2 \\ k \text{ even}}} f(a+k\,h_i) = \sum_{\ell=1}^{N_i/2-1} f(a+2\ell h_i) = \sum_{\ell=1}^{N_{i-1}-1} f(a+\ell h_{i-1})$$

where we have changed variables, $k = 2\ell$, and used that $2\,h_i = h_{i-1}$ and $N_i/2 = N_{i-1}$. One obtains a recurrence formula,

$$I_i = \underbrace{\frac{1}{2}\,h_{i-1} \left( \frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{\ell=1}^{N_{i-1}-1} f(a+\ell h_{i-1}) \right)}_{I_{i-1}} + h_i \sum_{\substack{1\le k\le N_i-1 \\ k \text{ odd}}} f(a+kh_i)\,.$$

# Adaptive trapezoid method

$$I_i = \frac{1}{2} I_{i-1} + h_i \sum_{\substack{1 \le k \le N_i - 1 \\ k \text{ odd}}} f(a + k h_i)$$

Code:

```
def int_trapez_ad(f, a, b, delta=1.0E-5, N=10):
    oldI = 1.0E308                  # "infinity"
    h = (b - a) / N
    newI = 0.5*f(a) + 0.5*f(b) # compute I_1
    for k in range(1, N):
        newI += f(a + h*k)
    newI *= h                       # end of computation of I_1
    while abs(oldI - newI)/3 > delta: # compute next I_i:
        h /= 2                      # decrease increment
        N *= 2                      # increase number of points
        oldI = newI                 # memorize I_(i-1)
        newI *= 0.5                   # first term = I_(i-1) / 2
        for k in range(1, N, 2): # add h f_k terms (k odd)
            newI += h * f(a + k*h)
    return newI
```

# Gaussian quadrature

**Recap:**

- Newton-Cotes methods are based on subdividing the integration interval into $N$ slices of the same width $h$.
  (The $p$-th degree method requires that $N$ is a multiple of $p$.)
- Moreover, the $p$-th degree Newton-Cotes method is exact if the integrand is a polynomial of degree $\leq p$. In this case, $N = p$ slices are sufficient.
- The integrand $f$ is evaluated at $N + 1$ points (nodes).

**Gaussian quadrature:**

- A method with $N$ nodes which is exact for polyomial integrands of even higher degree, up to $\leq 2N - 1$.
- It is correspondingly more precise for general integrands (that are well approximated by polynomials).
- Essential idea: instead of evenly spaced nodes, optimize the spacing between them.

# Gaussian quadrature

**General integration rule**:

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{k=0}^{N} w_k f_k$$

- $f_k = f(x_k)$ with the nodes $x_k \in [a, b]$, not necessarily evenly spaced, not necessarily $x_0 = a$ or $x_N = b$
- $\{w_k\}$ = weights

**Example:** Trapezoid rule, $x_k = a + kh$ and weights $w_0 = w_N = \frac{h}{2}$, $w_{1 \leq k \leq N-1} = h$

**Example:** Simpson's rule, $x_k = a + kh$ and $w_0 = w_N = \frac{h}{3}$, others $w_k = \frac{4h}{3}$ or $\frac{2h}{3}$

## Gaussian quadrature

To find the weights $w_k$, given a set of $N$ nodes $x_k$ $(1 \leq k \leq N)$, consider the interpolating polynomials of degree $N-1$:

$$\phi_{(k)}(x) = \prod_{\substack{m=1\ldots N \\ m\neq k}} \frac{x - x_m}{x_k - x_m}$$

$$= \left(\frac{x - x_1}{x_k - x_1}\right) \cdots \left(\frac{x - x_{k-1}}{x_k - x_{k-1}}\right) \cancel{\left(\frac{x - x_k}{x_k - x_k}\right)} \left(\frac{x - x_{k+1}}{x_k - x_{k+1}}\right) \cdots \left(\frac{x - x_N}{x_k - x_N}\right)$$

The essential property of the $\phi_{(k)}$:

$$\phi_{(k)}(x_n) = \delta_{nk} \equiv \begin{cases} 1\,, & n = k \\ 0\,, & n \neq k \end{cases}$$

Define

$$\Phi(x) = \sum_{k=1}^{N} f(x_k)\phi_{(k)}(x)$$

Properties of $\Phi$:

- Polynomial of degree $\leq N-1$ (linear combination of polynomials of degree $N-1$)
- $\Phi(x_m) = \sum_{k=1}^{N} f(x_k)\phi_{(k)}(x_m) = \sum_{k=1}^{N} f(x_k)\delta_{km} = f(x_m)$
- Unique with theses two properties, since its $N$ coefficients are fixed by $N$ constraints

# Gaussian quadrature

Approximating $f(x) \approx \Phi(x)$ on the domain of integration, we find

$$\int_a^b f(x)\,\mathrm{d}x \approx \int_a^b \Phi(x)\,\mathrm{d}x = \int_a^b \sum_{k=1}^{N} f(x_k)\phi_{(k)}(x)\,\mathrm{d}x = \sum_{k=1}^{N} f(x_k) \int_a^b \phi_{(k)}(x)\,\mathrm{d}x$$

and therefore

$$\boxed{w_k = \int_a^b \phi_{(k)}(x)\,\mathrm{d}x\,.}$$

- This gives the weights $\{w_k\}$ for a given generic set of nodes $\{x_k\}$, such that $\int_a^b f(x)\,\mathrm{d}x = \sum_k w_k f_k$ holds exactly for polynomial $f$
- Unfortunately, one cannot just compute them by integrating $\phi_{(k)}(x)$ analytically (polynomial — but defined by $2^{N-1}$ terms! Far too many for $N \gtrsim 30$). Must restrict to special cases where closed-form expressions exist. We will discuss an important example shortly.
- Fortunately, the $w_k$ need to be computed only once for a fixed choice of $a$ and $b$. Afterwards, one may easily adapt them to integrate any function $f(x)$ on any interval $[a, b]$.

**Adapting the nodes and weights to an arbitrary interval**

- Suppose we are given a set of nodes $\{x_k\}$ and corresponding weights $\{w_k\}$ on the reference interval $[-1, 1]$
- To adapt them to any other integration interval $[a, b]$: Redefine the nodes

$$x'_k = \underbrace{\frac{1}{2}(b-a)x_k}_{\text{compress/stretch}} + \underbrace{\frac{1}{2}(b+a)}_{\text{shift}} \qquad \text{(affine transformation)}$$

  and rescale the weights,

$$w'_k = \frac{1}{2}(b-a)w_k \,.$$

- Now we may integrate any function $f(x)$ on any interval $[a, b]$:

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{k=1}^{N} w'_k f(x'_k)\,.$$

# Gaussian quadrature

**How to choose the nodes** $x_k$ on the reference interval $[-1, 1]$ optimally?
**What are the corresponding weights** $w_k$?

Optimal choice, giving the exact result if $f(x)$ is a polynomial of degree $\leq 2N-1$:

$$\boxed{\begin{aligned} x_k &= \text{roots of the } N\text{th Legendre polynomial } P_N(x) \\ w_k &= \frac{2}{(1-x_k^2)\,P_N'(x_k)^2} \end{aligned}}$$

(Proof for the interested: see following slides.) See also TD 1.3.

Gauss-Legendre quadrature.

Other choices of $x_k$ and $w_k$ give exact results for

$$f(x) = W(x) \times \text{polynonial}$$

($\Rightarrow$ optimized results if $f$ is well approximated by such an expression)
where e.g. $W(x) = \frac{1}{\sqrt{1-x^2}}$ (Gauss-Chebyshev), $W(x) = x^\alpha e^{-x}$ (Gauss-Laguerre), $W(x) = e^{-x^2}$ (Gauss-Hermite)...

# Parenthesis: Proof of the Gauss-Legendre formulas, I — Nodes

To show that the nodes $x_k$ are the roots of the $N$-th Legendre polynomial, we need an important property of the latter which we quote without proof:

**Proposition:** Let $Q$ be a polynomial of degree $< n$. Then $Q$ and the $n$-th Legendre polynomial $P_n$ are orthogonal on $[-1, 1]$, i.e.

$$\int_{-1}^{1} P_n(x) Q(x) \, \mathrm{d}x = 0 \, .$$

(In fact, the usual definition of $P_n$ starts from this property.)

Now let us prove the following
**Theorem:** Let

- $f$ be a polynomial of degree $< 2N$
- $\{x_k \,|\, k = 1 \ldots N\}$ the roots of $P_N$
- $\phi_{(k)}$ the corresponding interpolating polynomials, i.e. the unique polynomials of degree $< N$ which satisfy $\phi_{(k)}(x_\ell) = \delta_{k\ell}$, see p. 62
- $w_k = \int_{-1}^{1} \phi_{(k)}(x) \, \mathrm{d}x$ (we will prove the explicit formula for $w_k$ afterwards)

Then

$$\int_{-1}^{1} f(x) \, \mathrm{d}x = \sum_{k=1}^{N} w_k \, f(x_k) \, .$$

# Parenthesis: Proof of the Gauss-Legendre formulas, I — Nodes

**Proof:** After polynomial division, $f(x) = P_N(x)Q(x) + R(x)$ where $Q$ and $R$ are polynomials of degree $< N$. We have

$$\int_{-1}^{1} f(x)\,\mathrm{d}x = \int_{-1}^{1} P_N(x)Q(x)\,\mathrm{d}x + \int_{-1}^{1} R(x)\,\mathrm{d}x$$

$$= \int_{-1}^{1} R(x)\,\mathrm{d}x \qquad \text{since } Q \perp P_N$$

$$= \int_{-1}^{1} \sum_{k=1}^{N} R(x_k)\phi_{(k)}(x)\,\mathrm{d}x \qquad \text{since } \sum_k R(x_k)\phi_{(k)} \text{ is the unique polynomial of degree } < N \text{ whose values at } x_k \text{ are } R(x_k), \text{ so it must be equal to } R$$

$$= \sum_{k=1}^{N} R(x_k) \int_{-1}^{1} \phi_{(k)}(x)\,\mathrm{d}x$$

$$= \sum_{k=1}^{N} \big(\underbrace{P_N(x_k)}_{=0} Q(x_k) + R(x_k)\big) w_k$$

$$= \sum_{k=1}^{N} f(x_k) w_k \,.$$

# Parenthesis: Proof of the Gauss-Legendre formulas, II — Weights

**Preliminary remarks** on the derivation of the weight formula:

- We will use the orthogonality property, as well as the recurrence relations of ex. 1.3

$$P_n'(x) = -\frac{nx}{1-x^2}P_n(x) + \frac{n}{1-x^2}P_{n-1}(x), \quad P_n(x) = \frac{2n-1}{n}xP_{n-1}(x) - \frac{n-1}{n}P_{n-2}(x)$$

- Note that the Legendre polynomials are not normalized via the scalar product of p. 66 but by the condition $P_n(1) = 1$. Indeed,

$$\int_{-1}^{1} P_n^2(x)\,\mathrm{d}x = \frac{2}{2n+1}\,.$$

- Finally, we denote by $a_n$ the leading coefficient of $P_n$, i.e. the prefactor of the $x^n$ term. Thus, if $\{x_m\}$ are the roots of $P_n$, then

$$P_n(x) = \prod_{m=1}^{n} \frac{x - x_m}{1 - x_m} = \underbrace{\prod_m \left(\frac{1}{1-x_m}\right)}_{=a_n} \prod_m (x - x_m) = a_n\, x^n + (\text{terms of degree } < n)$$

# Parenthesis: Proof of the Gauss-Legendre formulas, II — Weights

**Lemma 1:** Using the notation of the theorem of p. 66, we can write the interpolating polynomials $\phi_{(k)}$ as

$$\phi_{(k)}(x) = \frac{P_N(x)}{x - x_k} \frac{1}{P'_N(x_k)} \, .$$

**Proof:**

$$P_N(x) = a_N \prod_{m=1}^{N} (x - x_m) = a_N (x - x_k) \prod_{m \neq k} (x - x_m) = a_N (x - x_k) \phi_{(k)}(x) \prod_{m \neq k} (x_k - x_m)$$

where we have used the definition of $\phi_{(k)}$, see p. 62. Combining this with the definition of the derivative $P'_N(x_k)$,

$$P'_N(x_k) = \lim_{x \to x_k} \frac{P_N(x) - \overbrace{P_N(x_k)}^{=0}}{x - x_k} = a_N \underbrace{\phi_{(k)}(x_k)}_{=1} \prod_{m \neq k} (x_k - x_m)$$

and reinserting into the expression for $P_N(x)$ above gives the desired formula.

# Parenthesis: Proof of the Gauss-Legendre formulas, II — Weights

To obtain the weights $w_k = \int_{-1}^{1} \phi_{(k)}(x)\,\mathrm{d}x$, we still need to calculate $\int_{-1}^{1} \frac{P_N(x)}{x-x_k}\,\mathrm{d}x$.

**Lemma 2:** Any polynomial $Q$ of degree $\leq N$ satisfies the identity

$$Q(x_k)\int_{-1}^{1} \frac{P_N(x)}{x-x_k}\,\mathrm{d}x = \int_{-1}^{1} \frac{Q(x)P_N(x)}{x-x_k}\,\mathrm{d}x\,.$$

**Proof:** It is sufficient to consider $Q =$ some monomial $x^m$ with $m \leq N$. We have

$$\int_{-1}^{1} \frac{P_N(x)}{x-x_k}\,\mathrm{d}x = \int_{-1}^{1} P_N(x)\left(\frac{\left(\frac{x}{x_k}\right)^m}{x-x_k} + \frac{1-\left(\frac{x}{x_k}\right)^m}{x-x_k}\right)\mathrm{d}x\,.$$

The term in blue is a polynomial of degree $m-1 < N$. It is therefore orthogonal to $P_N$, hence it does not contribute to the integral, and one obtains

$$x_k^m \int_{-1}^{1} \frac{P_N(x)}{x-x_k}\,\mathrm{d}x = \int_{-1}^{1} \frac{x^m\,P_N(x)}{x-x_k}\,\mathrm{d}x\,.$$

# Parenthesis: Proof of the Gauss-Legendre formulas, II — Weights

Choosing $Q(x) = P_{N-1}(x)$ in Lemma 2, we can now finally prove the following

**Proposition:** The weights $w_k$ are given by

$$w_k = \frac{2}{1-x_k^2} \frac{1}{P_N'(x_k)^2} \,.$$

**Proof:** According to Lemma 2,

$$\begin{aligned}
P_{N-1}(x_k) \int_{-1}^{1} \frac{P_N(x)}{x-x_k} \,\mathrm{d}x &= \int_{-1}^{1} P_{N-1}(x) \underbrace{\frac{P_N(x)}{x-x_k}}_{= a_N x^{N-1} + (\text{ terms } \perp P_{N-1})} \mathrm{d}x \\
&= a_N \int_{-1}^{1} x^{N-1} P_{N-1}(x) \,\mathrm{d}x \\
&= a_N \int_{-1}^{1} \left( \frac{P_{N-1}(x)}{a_{N-1}} + (\text{ terms } \perp P_{N-1}) \right) P_{N-1}(x) \,\mathrm{d}x \\
&= \frac{a_N}{a_{N-1}} \int_{-1}^{1} P_{N-1}(x)^2 \,\mathrm{d}x \\
&= \frac{2}{2N-1} \frac{a_N}{a_{N-1}} \,.
\end{aligned}$$

# Parenthesis: Proof of the Gauss-Legendre formulas, II — Weights

**Continuation of the proof:**

Inserting this last expression into Lemma 1, one obtains

$$w_k = \int_{-1}^{1} \phi_{(k)}(x)\, \mathrm{d}x = \frac{1}{P'_N(x_k)} \int_{-1}^{1} \frac{P_N(x)}{x - x_k}\, \mathrm{d}x = \frac{2}{2N-1} \frac{a_N}{a_{N-1}} \frac{1}{P'_N(x_k) P_{N-1}(x_k)}\,.$$

Finally, use the recurrence relations to show that

$$\frac{a_N}{a_{N-1}} = \frac{2N-1}{N}$$

and that

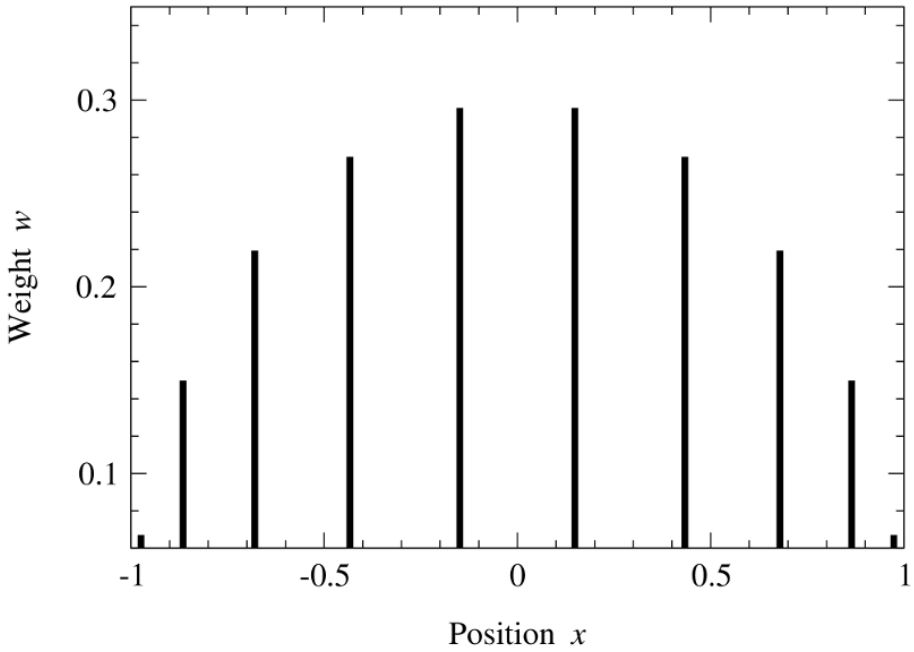
$$P_{N-1}(x_k) = \frac{1-x^2}{N} P'_N(x_k)$$

and insert into the above expression for $w_k$, which concludes the proof.

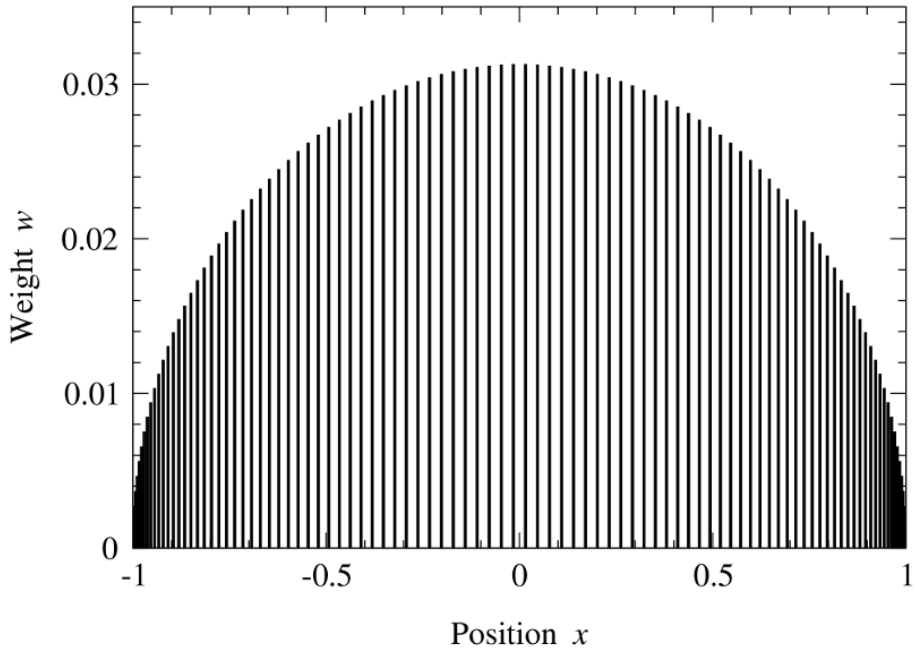
# Gaussian quadrature

Weights and nodes for Gauss-Legendre quadrature:

$N = 10$




$N = 100$




(Images taken from the book by M. Newman)

## Gaussian quadrature

```
def int_gauss(f, nodes, weights):
    result = 0.0
    for x, w in zip(nodes, weights):
        result += w * f(x)
    return result
```

The file gaussxw.py contains a function gaussxw(N) which computes the nodes and weights for Gauss-Legendre quadrature on the interval $[-1, 1]$ for any given $N$. Example:

```
from gaussxw import gaussxw
N = 100
x, w = gaussxw(N)

# adapt x -> x' and w -> w' to the interval [a, b]:
a, b = 0, 1             # using [a, b] = [0, 1] as an example
xp = 0.5*(b - a)*x + 0.5*(b + a)
wp = 0.5*(b - a)*w

# integrate some function (e.g. arctanh(x)) on [a, b]:
from math import atanh
print("Result:", int_gauss(atanh, xp, wp))
```

# Gaussian quadrature

**Advantages:**

- Excellent convergence for integrands which are well approximated by polynomials (or by $W(x) \times$ polynomial for suitable $W(x)$)
- Very few function calls of $f(x)$ are necessary $\Rightarrow$ ideal if evaluating the integrand is expensive
- Open method: no need to evaluate the boundary points $f(a)$ and $f(b)$

**Drawbacks:**

- Poor convergence for irregular integrands
- Computing nodes and weights may be expensive (but needs to be done only once)
- Impossible to re-use previously calculated points after an increase of $N$
  $\Rightarrow$ error estimation can be difficult and costly

**In practice:**

- Instead of `gaussxw(N)`, one may use the `NumPy` function
  `numpy.polynomial.legendre.leggauss(N)`
- Nodes and weights for Gauss-Chebyshev, Gauss-Laguerre, Gauss-Hermite are also found in `NumPy`

# Gaussian quadrature

## Exercises

In the Debye model, the heat capacity of a solid is given by

$$C_V = 9\,n\,V\,k_B \left(\frac{T}{\Theta_D}\right)^3 \int_0^{\Theta_D/T} \frac{x^4\,e^x}{(e^x-1)^2}\,\mathrm{d}x$$

where $V$ is the volume, $n$ is the number density, $k_B = 1.38 \cdot 10^{-23}\ \mathrm{JK}^{-1}$ is Boltzmann's constant, $T$ is the temperature, and $\Theta_D$ is a constant.

- Write a function `CV(T)` which calculates $C_V$ as a function of temperature, for a cube of aluminium of $(10 \times 10 \times 10)\mathrm{cm}^3$ ($n = 6.022 \cdot 10^{28}\ \mathrm{m}^{-3}$, $\Theta_D = 428\ \mathrm{K}$). Use Gauss-Legendre quadrature with $N = 50$ nodes.
- Plot $C_V(T)$ between $T = 5\ \mathrm{K}$ and $T = 500\ \mathrm{K}$.

# Comparison of numerical integration methods

- Trapezoidal method:
  - Easy to implement
  - Slow convergence
  - Good for irregular integrands
- Simpson's method:
  - Easy to implement
  - Rather fast convergence
  - Poor choice for irregular integrands
- Gaussian quadrature:
  - Implementation requires computing nodes and weights
  - Very fast
  - Poor choice for irregular integrands

Other methods exist, notably Romberg integration which relies on Richardson extrapolation to accelerate convergence.

# Numerical integration: Improper integrals

To calculate an improper integral,

$$\int_0^\infty f(x)\,\mathrm{d}x$$

the standard procedure is to change variables:

$$y = \frac{x}{1+x}\,, \qquad x = \frac{y}{1-y}\,.$$

Thus

$$\mathrm{d}x = \frac{\mathrm{d}y}{(1-y)^2}\,, \qquad \int_0^\infty f(x)\,\mathrm{d}x = \int_0^1 \frac{1}{(1-y)^2} f\left(\frac{y}{1-y}\right)\mathrm{d}y\,.$$

- To calculate $\int_a^\infty f(x)\,\mathrm{d}x$: calculate $\int_0^\infty f(x)\,\mathrm{d}x$ and subtract $\int_0^a f(x)\,\mathrm{d}x$.
- To calculate $\int_{-\infty}^\infty f(x)\,\mathrm{d}x$: calculate the sum of $\int_0^\infty f(x)\,\mathrm{d}x$ and $\int_{-\infty}^0 f(x)\,\mathrm{d}x$.
- Depending on the integrand, other choices of variables may give better results, for example

$$y = \frac{x^\alpha}{\beta + x^\alpha} \qquad \text{with suitable constants } \alpha,\, \beta\,.$$

# Numerical integration: Singularities

The integrand may exhibit singularities within the domain of integration, or at its boundary.

If the behaviour near the singularities is known, convergence may be improved by subtracting the singular terms and calculating them separately.

**Example:** Calculate

$$I = \int_{-1}^{1} \frac{1}{\sqrt{|\sin(x)|}}\,\mathrm{d}x$$

- Integrand singular at $x = 0$, where $\sin x \sim x$.
- Subtracting $\frac{1}{\sqrt{|x|}}$:

$$I = \underbrace{\int_{-1}^{1} \left( \frac{1}{\sqrt{|\sin(x)|}} - \frac{1}{\sqrt{|x|}} \right) \mathrm{d}x}_{\text{regular}} + \underbrace{\int_{-1}^{1} \frac{1}{\sqrt{|x|}}\,\mathrm{d}x}_{=2\int_0^1 \frac{1}{\sqrt{x}}\,\mathrm{d}x = 2\left[2\sqrt{x}\right]_0^1 = 4}$$

- Now the first term can be calculated reliably with our numerical integration methods.

# Numerical derivatives

**Goal**: Given a differentiable function $f(x)$ (which can be evaluated numerically), compute $f'(x)$.

**Preferred solution** if possible: compute $f'$ analytically and evaluate the result numerically.

- If $f$ is any combination of elementary functions, then $f'$ can be easily computed analytically
- Simplest techniques for calculating numerical derivatives are rather imprecise.

But sometimes we don't have an explicit expression for $f(x)$ (if the values of $f$ are themselves obtained by some numerical procedure). In this case, one may need to compute $f'$ purely numerically.

# Numerical derivatives: Forward and backward differences

**Definition** of the derivative:

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

Approximation

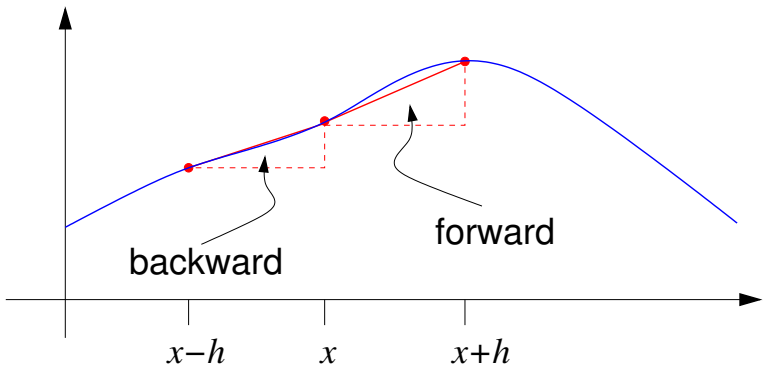
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

for $h$ sufficiently small: forward difference.

Equivalent:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h} ,$$

for $h$ sufficiently small: backward difference.



# Numerical derivatives: error estimate

**Error** on the derivative obtained by forward differencing:

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \dots \text{ (Taylor expansion)}$$

$$\Rightarrow \qquad f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}h\, f''(x) + \dots$$

Error $\mathcal{O}(h)$.

**Problem**: choosing $h$ small, the truncation error shrinks, but the rounding error grows.

Reason: subtracting $f(x)$ from $f(x+h)$, two numbers that are very close → see chapter 2 and exercise 1.3. Extreme example: $f(x) = x^2$, derivative at $x = 1$ with $h = 10^{-16}$:

```
h = 1.0E-16
print(((1.0+h)**2 - 1.0**2) / h)
```

This gives 0.0 although the result should be 2!

Optimal choice for this method if $f(x) = \mathcal{O}(1)$: $h \approx 10^{-8}$, not very precise. Similar for backward differencing.

# Central difference

**Average** of forward and backward differences with a step width $h/2$:

$$\boxed{f'(x) \approx \frac{f\left(x+\frac{h}{2}\right) - f\left(x-\frac{h}{2}\right)}{h}}$$

Taylor expansion:

$$f\left(x+\frac{h}{2}\right) = f(x) + \frac{1}{2}hf'(x) + \frac{1}{8}h^2 f''(x) + \frac{1}{48}h^3 f'''(x) + \dots$$
$$f\left(x-\frac{h}{2}\right) = f(x) - \frac{1}{2}hf'(x) + \frac{1}{8}h^2 f''(x) - \frac{1}{48}h^3 f'''(x) + \dots$$

Subtracting these two equations gives

$$f'(x) = \frac{f\left(x+\frac{h}{2}\right) - f\left(x-\frac{h}{2}\right)}{h} - \frac{1}{24}h^2 f'''(x) + \dots$$

**Better** than forward and backward differences: error $\mathcal{O}(h^2)$.

Optimal choice for $f(x) = \mathcal{O}(1)$: $h \approx 10^{-5}$, error $\epsilon \approx 10^{-10}$.

# Second derivative

Central difference:

$$f''(x) \approx \frac{f'\left(x+\frac{h}{2}\right) - f'\left(x-\frac{h}{2}\right)}{h}$$

With

$$f'\left(x+\frac{h}{2}\right) \approx \frac{f(x+\frac{h}{2}+\frac{h}{2}) - f\left(x+\frac{h}{2}-\frac{h}{2}\right)}{h} = \frac{f(x+h)-f(x)}{h}$$

and

$$f'\left(x-\frac{h}{2}\right) \approx \frac{f(x)-f(x-h)}{h}$$

one finds

$$\boxed{f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}\,.}$$

Error:

$$\epsilon = -\frac{1}{12}h^2\, f''''(x) + \ldots \qquad (\rightarrow \text{ exercices})$$

Optimal choice for $f(x) = \mathcal{O}(1)$: $h \approx 10^{-4}$, error $\epsilon \approx 10^{-8}$.

## Exercises

- Show that

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{1}{12} h^2 \, f''''(x) + \mathcal{O}(h^3) \, .$$

- Let us study the numerical derivative of $f(x) = 1 + \frac{1}{2} \tanh(2\,x)$.
  - Write a corresponding Python function `f(x)` (use the pre-defined function `numpy.tanh`). Plot its graph on the interval $[-2, 2]$.
  - Compute $f'(x)$ analytically.
  - Plot the difference between your analytic expression for $f'(x)$ and the numerical derivative of $f(x)$ on the interval $[-2, 2]$. Compute the numerical derivative using central differencing with $h = 10^{-4}$, $h = 10^{-5}$, and $h = 10^{-6}$. Compare the three graphs; which choice of the step width gives the best result?