Modelling and Simulation in Physics

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

1. Introduction

2. Numerical error, stability, algorithmic complexity

3. Numerical integrals and derivatives

4. Ordinary differential equations

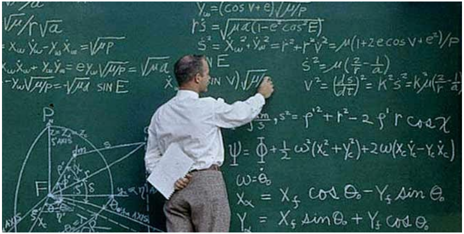
5. Partial differential equations

6. Monte-Carlo methods

Introduction

Computational physics

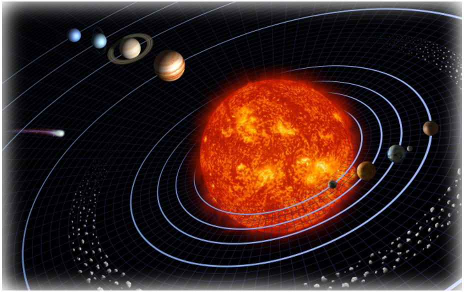
A common situation in physics: The equations describing a physical system are known but cannot be solved analytically.



- 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

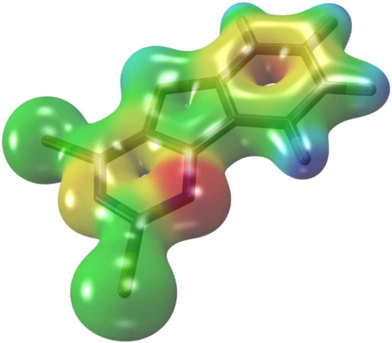
Example: celestial mechanics



- 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

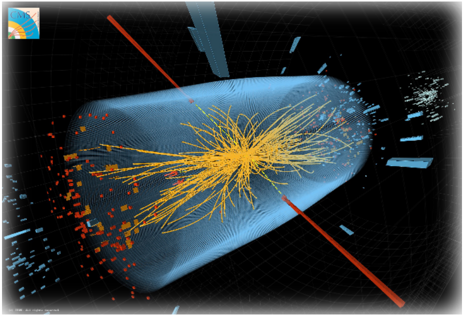
Example: quantum chemistry



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

Computational physics

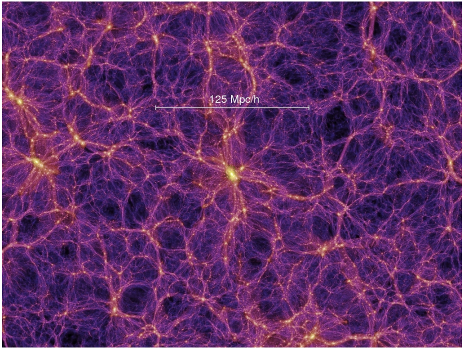
Example: elementary particle physics



- 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 → 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

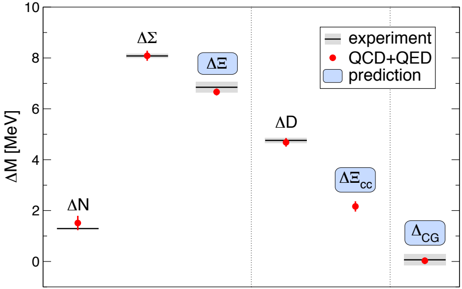
Computational general relativity → Ossokine/Buonanno/Dietrich/Haas, SXS project 2017

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 ΔN (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

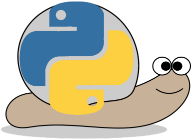
Computational physics with Python

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
⇒ 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`)

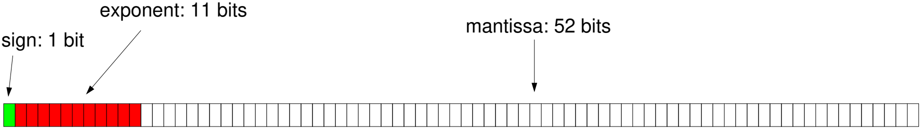
Unlike most other programming languages, there is (theoretically) no limit to the size of an `int` in Python: arbitrary-precision arithmetic.
In practice it is of course limited by the machine's memory.

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: $\sqrt{2} = 1.4142135623730950488\ldots$
Rounding error: $0.0000000000000000512\ldots$

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:

```python
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:

```python
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
```

⇒ 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 ϵ, ϵ_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 σ the standard deviation of ϵ and by C the standard deviation of ϵ_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 σ) in order to estimate the typical size of ϵ_r (or ϵ).

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 σ_1 and σ_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.

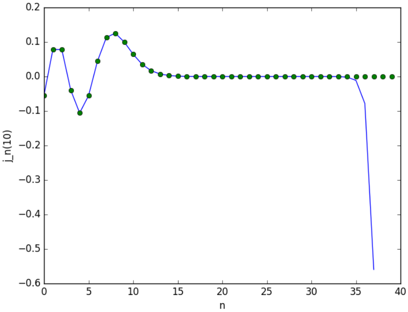
Numerical stability

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 ⇒ 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 $B(n)\dots$)

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.

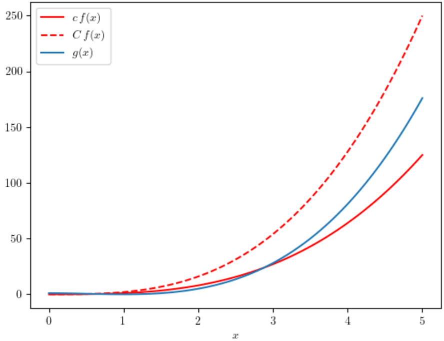
We say of some other function $g : \mathbb{R}_+ \to \mathbb{R}_+$ that

- $g \in \mathcal{O}(f)$
  $\Leftrightarrow$ 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 Θ 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

```python
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 $T(n) = \text{const.}$, 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.

```python
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 μs	10 μs	10 μs	10 μs	10 μs	10 μs
$\Theta(\log n)$	10 μs	13 μs	15 μs	20 μs	30 μs	40 μs
$\Theta(\sqrt{n})$	10 μs	14 μs	17 μs	32 μs	100 μs	320 μs
$\Theta(n)$	10 μs	20 μs	30 μs	100 μs	1 ms	10 ms
$\Theta(n^2)$	10 μs	40 μs	90 μs	1 ms	100 ms	10 s
$\Theta(n^3)$	10 μs	80 μs	270 μs	10 ms	10 s	3 h
$\Theta(e^n)$	10 μs	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)

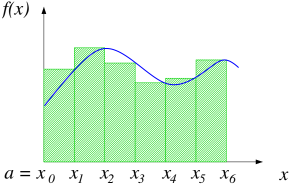
Possible complications (→ later):

- 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 → Monte-Carlo methods, chapter 6

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

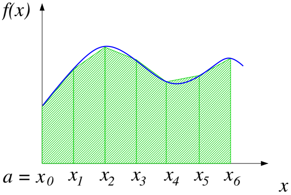
$$\int_a^b f(x)\,dx = \lim_{N\to\infty} \sum_{k=1}^{N} h\, f_k\,, \qquad h = \frac{b-a}{N}\,, \quad f_k = f(x_k)\,, \quad x_k = a + kh$$

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

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

Newton-Cotes methods: Trapezoid method

Simple function for calculating integrals with the trapezoid method:

```python
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:

```python
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}:

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

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:

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

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
- ϵ_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

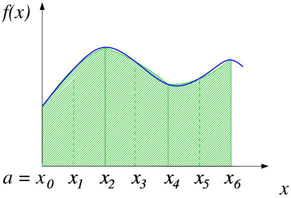
$$I = I_2 + c\,h_2^2 + \ldots$$

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}):

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

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

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

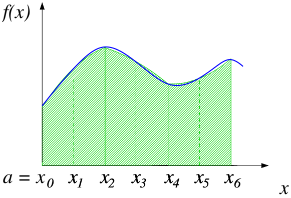
Solution:

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

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

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

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

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 $1/n$,

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

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,

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

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 genenеral, 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 δ:

- 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 → save computing resources.)
- Compute ϵ_{i+1}. When $|\epsilon_{i+1}| < \delta$, terminate.

Adaptive trapezoid method

To re-use the points calculated previously, note that

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

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

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

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:

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

- $f_k = f(x_k)$ 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 $N-1$:

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

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

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 Φ:

- 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 $[a, b]$.

Gaussian quadrature

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

$$\begin{aligned}
\int_{-1}^{1} f(x)\ \mathrm{d}x &= \int_{-1}^{1} P_N(x)Q(x)\ \mathrm{d}x + \int_{-1}^{1} R(x)\ \mathrm{d}x \\
&= \int_{-1}^{1} R(x)\ \mathrm{d}x && \text{since } Q \perp P_N \\
&= \int_{-1}^{1} \sum_{k=1}^{N} R(x_k)\phi_{(k)}(x)\ \mathrm{d}x && \text{since } \sum_k R(x_k)\phi_{(k)} \text{ is the unique polynomial} \\
& && \text{of degree } < N \text{ whose values at } x_k \text{ are } R(x_k), \\
& && \text{so it must be equal to } R \\
&= \sum_{k=1}^{N} R(x_k) \int_{-1}^{1} \phi_{(k)}(x)\ \mathrm{d}x \\
&= \sum_{k=1}^{N} \big(\underbrace{P_N(x_k)}_{=0} Q(x_k) + R(x_k)\big) w_k \\
&= \sum_{k=1}^{N} f(x_k) w_k\ .
\end{aligned}$$

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

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

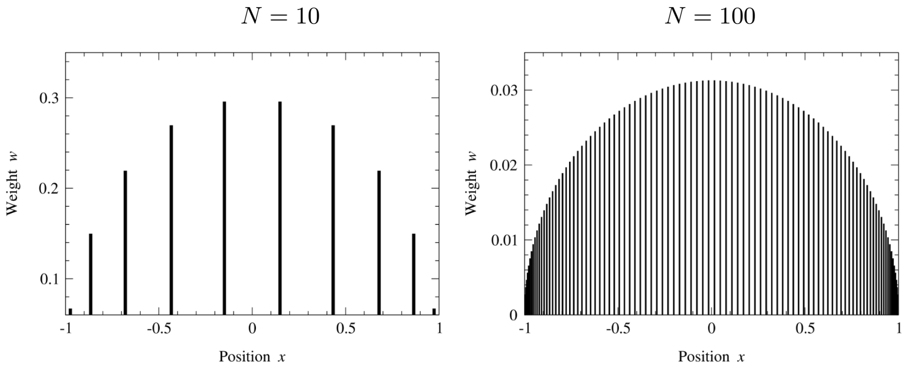
and that

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

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

Gaussian quadrature

Weights and nodes for Gauss-Legendre quadrature:



(Images taken from the book by M. Newman)

Gaussian quadrature

```python
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:

```python
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 Θ_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

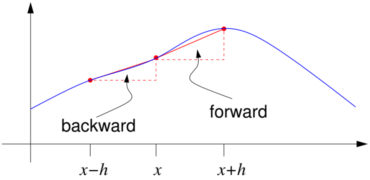
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

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:

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

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

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:

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

Subtracting these two equations gives

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

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:

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

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?

Ordinary differential equations

In this chapter

- Euler's method
- Runge-Kutta methods
- Higher-order ODEs and systems of ODEs
- Adaptive Runge-Kutta method
- Boundary value problems

Ordinary differential equations

Reminder: ODE = differential equation whose unknown function(s) depend on a single parameter t.

Example: harmonic oscillator,

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}x(t) + \omega^2 x(t) = 0\,, \qquad \text{solution: } x(t) = A\cos\omega t + B\sin\omega t\,.$$

For now focus on ODEs of the first order with a single unknown function. Assume the ODE can be brought into standard form:

$$\boxed{\frac{\mathrm{d}}{\mathrm{d}t}x(t) = f\left(x(t), t\right)\,.}$$

Initial-value problem: Given an ODE and an initial condition

$$\boxed{x(0) = x_0}\,,$$

what is the function $x(t)$ which satisfies both?

Euler's method

To find a numerical solution of the initial-value problem

$$\frac{\mathrm{d}}{\mathrm{d}t}x(t) = f(x,t)\,, \qquad x(0) = x_0\,,$$

perform a Taylor expansion:

$$x(h) = x(0) + h\frac{\mathrm{d}x}{\mathrm{d}t}(0) + \frac{1}{2}h^2\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(0) + \mathcal{O}(h^3) = x(0) + h\,f(x,0) + \mathcal{O}(h^2)\,.$$

For small h, neglect $\mathcal{O}(h^2)$ terms:

$$x(h) \approx x(0) + h\,f(x(0),0)\,.$$

With the same approximation, calculate $x(2h)$:

$$x(2h) \approx x(h) + h\,f(x(h),h)$$

and then $x(3h)$, $x(4h)$ etc. Euler's method. Generally:

$$\boxed{x(t+h) \approx x(t) + h\,f(x(t),\,t)\,.}$$

Euler's method

Example: Given the ODE

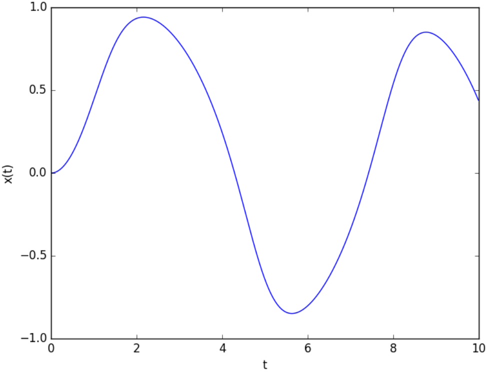
$$\frac{\mathrm{d}x}{\mathrm{d}t} = -x^3 + \sin t$$

with the initial condition $x(0) = 0$, find $x(t)$ for 1001 points between $t = 0$ and $t = 10$:

```python
import numpy as np
import matplotlib.pyplot as plt

def f(x, t):
    return -x**3 + np.sin(t)

a, b, = 0.0, 10.0          # interval
N = 1000                   # number of steps
h = (b - a) / N            # step size
x = 0.0                    # initial condition
tpoints, xpoints = np.linspace(a, b, N+1), []
for t in tpoints:
    xpoints += [x]
    x += h * f(x, t)

plt.plot(tpoints, xpoints)
plt.show()
```

Euler's method

Result:



Euler's method

Error estimate:

- At each step, neglecting a $\mathcal{O}(h^2)$ term $\frac{1}{2}h^2\frac{\mathrm{d}^2x}{\mathrm{d}t^2}$ (leading local error term).
- There are $N = (b-a)/h$ steps in total. Hence

$$\begin{aligned}\sum_{k=0}^{N}\frac{1}{2}h^2\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t_k) &= \frac{1}{2}h\sum_{k=0}^{N}h\frac{\mathrm{d}f}{\mathrm{d}t}(x_k,t_k) \approx \frac{1}{2}h\int_a^b\frac{\mathrm{d}f}{\mathrm{d}t}\,\mathrm{d}t\\ &= \frac{1}{2}h\left(f(x(b),b)-f(x(a),a)\right)\,.\end{aligned}$$

  We have approximated the sum by an integral at small h. The accumulated (global) error term is therefore $\mathcal{O}(h)$.

- Euler's method is simple but not very precise.
- It can also become unstable as we will see next.

Euler's method

Numerical stability

Example:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\lambda x\,, \qquad x(0) = 1$$

with $\lambda > 0$. Analytic solution:

$$x(t) = e^{-\lambda t}$$

Numerical solution with Euler:

$$\begin{aligned} x(h) &= x(0) - h\lambda\, x(0) = x(0)(1-h\lambda) = 1 - h\lambda \\ x(2h) &= x(h) - h\lambda\, x(h) = x(h)(1-h\lambda) = (1-h\lambda)^2 \\ &\dots \\ x(n\,h) &= (1-h\lambda)^n \qquad\qquad \text{cf. } e^y = \lim_{n\to\infty}\left(1+\frac{y}{n}\right)^n \end{aligned}$$

- If $0 < h < \frac{2}{\lambda}$, then $|1-h\lambda| < 1$, thus $\lim_{n\to\infty}(1-h\lambda)^n = 0$ as it should be
- But if $h > \frac{2}{\lambda}$, then $(1-h\lambda)^n$ diverges as $n \to \infty$.

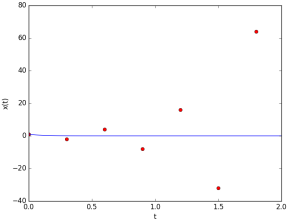
Stability condition: $h < \frac{2}{\lambda}$.

Euler's method

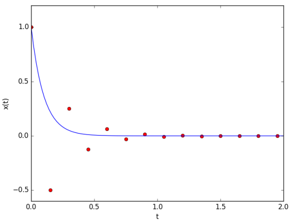
Numerical stability

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\lambda x\,, \qquad \lambda > 0\,, \qquad x(0) = 1\,.$$

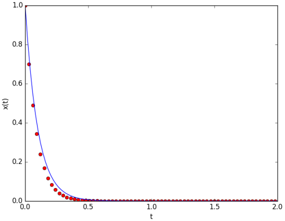
Solution with Euler for $\lambda = 10$ et $h = 0.3$, $h = 0.15$, $h = 0.05$: blue curve = $e^{-\lambda t}$, red points = numerical approximation



$h = 0.3$



$h = 0.15$



$h = 0.05$

Implicit Euler's method

The version of Euler's method we used is explicit: knowing the state of the system at t, one can directly calculate its state at $t + h$,

$$x(t+h) \approx x(t) + h\, f(x(t),\, t)$$

More stable: Implicit (backward) Euler's method, where

$$x(t+h) \approx x(t) + h\, f(x(t+h),\, t+h)$$

Harder to implement: one must first solve this equation with respect to $x(t+h)$.

Example as before: $f(x) = -\lambda x$, $x(0) = 1$:

$$x(h) = x(0) - h\lambda x(h) = 1 - h\lambda x(h) \quad \Rightarrow \quad x(h) = \frac{1}{1+h\lambda}$$

$$\ldots$$

$$x(nh) = x((n-1)h) - h\lambda x(nh) = \frac{1}{(1+h\lambda)^{n-1}} - h\lambda(nh) \quad \Rightarrow \quad x(nh) = \frac{1}{(1+h\lambda)^{n}}$$

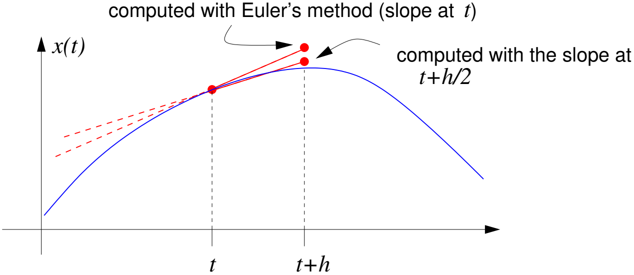
Tends towards 0 as $n \to \infty$ for all $h > 0$ (as it should: $\lim_{x\to\infty} e^{-\lambda x} = 0$). Stable.

Runge-Kutta methods: Second order

Euler's method = Runge-Kutta method of the first order.

Second-order Runge-Kutta method ("midpoint method"):

Extrapolate from $x(t)$ to the next point $x(t+h)$ using the slope at $t + h/2$, not at t:


computed with Euler's method (slope at t)
x(t)
computed with the slope at t+h/2
t
t+h

Better approximation.

Runge-Kutta methods: Second order

Taylor expansion around $t + \frac{h}{2}$:

$$x(t+h) = x(t+h/2) + \frac{1}{2}h\frac{\mathrm{d}x}{\mathrm{d}t}(t+h/2) + \frac{1}{8}h^2\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t+h/2) + \mathcal{O}(h^3)\,.$$

Similarly:

$$x(t) = x(t+h/2) - \frac{1}{2}h\frac{\mathrm{d}x}{\mathrm{d}t}(t+h/2) + \frac{1}{8}h^2\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t+h/2) + \mathcal{O}(h^3)\,.$$

Subtracting:

$$x(t+h) = x(t) + h\frac{\mathrm{d}x}{\mathrm{d}t}(t+h/2) + \mathcal{O}(h^3) = x(t) + hf(x(t+h/2), t+h/2) + \mathcal{O}(h^3)\,.$$

The $\mathcal{O}(h^2)$ terms have cancelled. The error is $\mathcal{O}(h^3)$.

Problem: We need to know $x(t+h/2)$ to evaluate $hf(x(t+h/2), t+h/2)$. But we haven't yet computed x beyond t.

Solution: Approximate as in Euler's method, $x(t+h/2) \approx x(t) + \frac{1}{2}hf(x,t)$. The error remains of order h^3 since (after Taylor expansion)

$$f\left(x(t) + \tfrac{1}{2}h\,f(x,t), t+h/2\right) = f\left(x(t+h/2), t+h/2\right) + \mathcal{O}(h^2)\,.$$

Runge-Kutta methods: Second order

Algorithm for second-order Runge-Kutta:

- Compute
$$k_1 \equiv h\, f(x,t).$$

- Use this to approximate $x(t+h/2) \approx x(t) + \frac{1}{2} h f(x,t) = x(t) + \frac{1}{2} k_1$. Insert into f to obtain
$$k_2 \equiv h f(x + k_1/2, t + h/2)\,.$$

- Finally, compute
$$x(t+h) = x(t) + k_2\,.$$

Exercises

Adapt the program on p. 90 to calculate the solution of the initial-vale problem

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -x^3 + \sin t\,, \qquad x(0) = 0$$

with the second-order Runge-Kutta method. Plot the solution for $N = 10$, $N = 20$, $N = 100$. Compare with the result obtained in the lecture with Euler's method and $N = 1000$.

Runge-Kutta methods: RK4

Fourth-order Runge-Kutta method

This scheme can be generalized (several Taylor expansions at suitable intermediate points, taking linear combinations such that all terms up to some order h^n cancel): Runge-Kutta methods.

A good compromise which runs very efficiently but is still easy to implement is the famous fourth-order Runge-Kutta method. The equations are still quite simple:

$$\boxed{\begin{aligned} k_1 &= hf(x,t)\,, \\ k_2 &= hf(x+k_1/2, t+h/2)\,, \\ k_3 &= hf(x+k_2/2, t+h/2)\,, \\ k_4 &= hf(x+k_3, t+h)\,, \\ x(t+h) &= x(t) + \frac{1}{6}(k_1 + 2\,k_2 + 2\,k_3 + k_4)\,. \end{aligned}}$$

Runge-Kutta methods: RK4

Code:

```python
import numpy as np

def rk4(f, a, b, x0, N):
    h = (b - a) / N    # step size
    x = x0             # at t=a, x=x0
    tpoints = np.linspace(a, b, N+1)
    xpoints = []

    for t in tpoints:
        xpoints += [x]
        k1 = h * f(x, t)
        k2 = h * f(x + k1/2, t + h/2)
        k3 = h * f(x + k2/2, t + h/2)
        k4 = h * f(x + k3, t + h)
        x = x + (k1 + 2*k2 + 2*k3 + k4)/6

    # return arrays containing t and x(t)
    return tpoints, np.array(xpoints)
```

Euler vs. RK4

Consider the initial-value problem

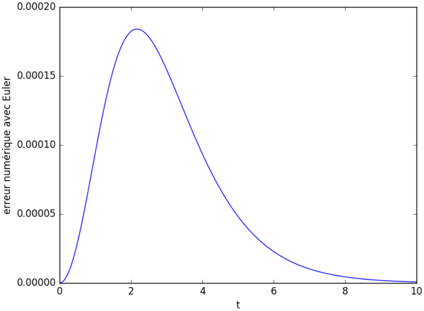
$$\frac{\mathrm{d}x}{\mathrm{d}t} = x - \frac{x^2}{2}\,, \qquad x(0) = 1\,.$$

Analytic solution:

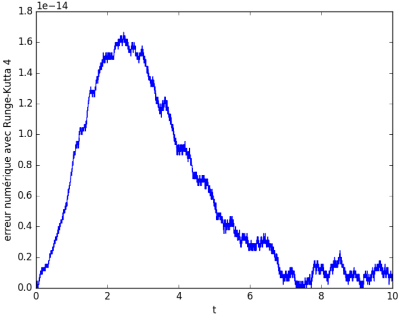
$$x(t) = \frac{2}{1+e^{-t}}$$

Numerical solution with $N = 5000$ steps between 0 and 10

Error with Euler:



Error with RK4:



smooth shape ⇐ truncation error dominates

noisy shape ⇐ rounding error

Higher-order ODEs

- A second-order ODE can always be reduced to two first-order ODEs.
  Example:
  $$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} x - x \frac{\mathrm{d}x}{\mathrm{d}t} - x^2 - t = 0$$
  Defining $v(t) = \frac{\mathrm{d}x}{\mathrm{d}t}$, one obtains the equivalent equations
  $$\frac{\mathrm{d}x}{\mathrm{d}t} = v\,, \qquad \frac{\mathrm{d}v}{\mathrm{d}t} = v + x + \frac{t}{x}\,.$$

- An ODE of the n-th order can always be reduced to n ODEs of the first order.
  Same idea: defining $v(t) = \frac{\mathrm{d}x}{\mathrm{d}t}$, $a(t) = \frac{\mathrm{d}v}{\mathrm{d}t}$ etc., find the solution of the system of coupled first-order equations with unknown functions x, v, a, $\ldots$

Systems of coupled ODEs

Now we are dealing with several unknown functions (which still depend on a single parameter t).

- Standard form for a first-order system with two unknown functions:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f_x(x,\, y,\, t)\,, \qquad \frac{\mathrm{d}y}{\mathrm{d}t} = f_y(x,\, y,\, t)\,.$$

- With n unknown functions, in vector notation

$$\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} = \mathbf{f}(\mathbf{r},\, t)$$

  where $\mathbf{r} = (x(t),\, y(t),\, \ldots)$ and $\mathbf{f}(\mathbf{r},\, t) = (f_x(\mathbf{r},\, t),\, f_y(\mathbf{r},\, t),\, \ldots)$

- The numerical methods to solve such systems are no more difficult in principle than those for a single ODE, they are simple generalizations

Systems of coupled ODEs

Fourth-order Runge-Kutta method for n coupled equations:

Let

$$\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} = \mathbf{f}(\mathbf{r},\, t)$$

with $\mathbf{r}(0) = \mathbf{r}_0$.

At each step, compute

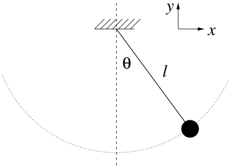
$$\begin{aligned}
\mathbf{k}_1 &= h\,\mathbf{f}(\mathbf{r},\, t)\,,\\
\mathbf{k}_2 &= h\,\mathbf{f}(\mathbf{r} + \mathbf{k}_1/2,\, t + h/2)\,,\\
\mathbf{k}_3 &= h\,\mathbf{f}(\mathbf{r} + \mathbf{k}_2/2,\, t + h/2)\,,\\
\mathbf{k}_4 &= h\,\mathbf{f}(\mathbf{r} + \mathbf{k}_3,\, t + h)\,,
\end{aligned}$$

Finally, obtain the next point as

$$\mathbf{r}(t+h) = \mathbf{r}(t) + \frac{1}{6}(\mathbf{k}_1 + 2\,\mathbf{k}_2 + 2\,\mathbf{k}_3 + \mathbf{k}_4)\,.$$

Same equations as before for a single unknown $x(t)$. With Python and `NumPy`, can even reuse the same code thanks to built-in vectorization.

Example: Simple gravity pendulum



- Lagrangian: $L = T - V = \frac{1}{2}m\left(\dot{x}^2 + \dot{y}^2\right) - mgy = \frac{1}{2}m\ell^2\dot{\theta}^2 + mg\ell\cos\theta$
- Equations of motion: $\frac{\partial L}{\partial \theta} - \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{\theta}} = 0$, hence

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{\ell}\sin\theta$$

Second-order ODE for $\theta = \theta(t)$. Anharmonic oscillator.

- With $\omega = \frac{\mathrm{d}\theta}{\mathrm{d}t}$, equivalent to two first-order ODEs:

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = -\frac{g}{\ell}\sin\theta\,, \qquad \frac{\mathrm{d}\theta}{\mathrm{d}t} = \omega\,.$$

Example: Simple gravity pendulum

We want to solve

$$\frac{\mathrm{d}\omega}{\mathrm{d}t} = -\frac{g}{\ell}\sin\theta\,, \qquad \frac{\mathrm{d}\theta}{\mathrm{d}t} = \omega\,.$$

with initial conditions $\theta(0) = \theta_0$, $\omega(0) = \omega_0$.

Start by implementing the function `f` which returns the right-hand side of the ODE:

```python
import numpy as np

g = 9.81  # gravitational acceleration
L = 0.1   # pendulum length = 10 cm

def f(r, t):
    theta, omega = r  # r is the array [theta(t), omega(t)]
    ftheta = omega
    fomega = -g / L * np.sin(theta)
    return np.array([ftheta, fomega])
```

Example: Simple gravity pendulum

Now for the setup of the initial-value problem:

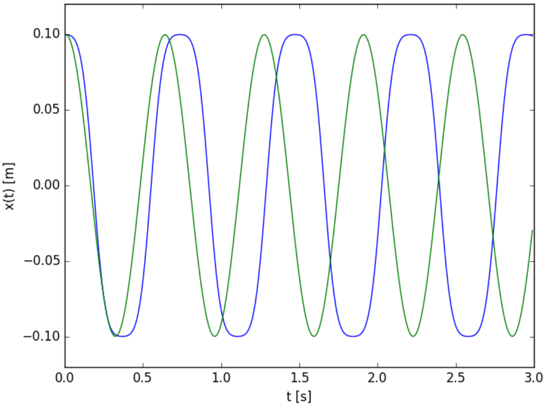
```
theta0 = 1.5 # angle at t=0
omega0 = 0.  # angular velocity at t=0
a, b = 0, 3  # time interval
N = 300      # number of steps
h = (b - a) / N              # step size
tpoints = np.linspace(a, b, N+1) # intermediate times
thetapoints = []             # we'll put theta(t) in here
omegapoints = []             # we'll put omega(t) in here
r = np.array([theta0, omega0]) # initial conditions
```

Finally calculate the solution with RK4:

```
for t in tpoints:
    thetapoints += [r[0]]
    omegapoints += [r[1]]
    k1 = h * f(r, t)
    k2 = h * f(r + k1/2, t + h/2)
    k3 = h * f(r + k2/2, t + h/2)
    k4 = h * f(r + k3, t + h)
    r = r + (k1 + 2*k2 + 2*k3 + k4)/6
```

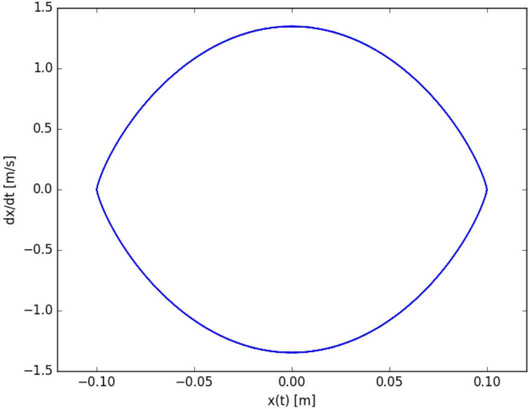
Example: Simple gravity pendulum

Plotting $x(t) = \ell \sin\theta(t)$ (blue graph) and, for comparison, the sinusoidal function obtained in the harmonic approximation (green line):



Example: Simple gravity pendulum

Plotting $x(t) = \ell \sin\theta(t)$ against $\dot{x}(t) = \ell\omega(t)\cos\theta(t)$: phase-space diagram

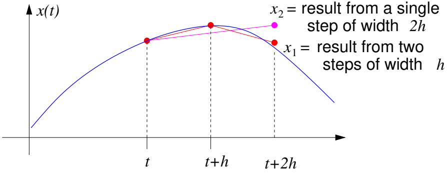


Adaptive methods: RK4 with variable step size

Idea: vary h such that the local error remains approximately constant at each step.

Motivation: In x, t regions where the right-hand side $f(x,\,t)$ is varying slowly, a relatively large h can already give a good precision. However, if the variation of $f(x,t)$ is fast, h must be chosen smaller.

Local error estimation: Compute two successive steps of size h. Then, from the same starting point, a single step of size $2h$.



x(t)
x_2 = result from a single step of width 2h
x_1 = result from two steps of width h
t
t+h
t+2h

The RK4 method is exact up to order h^4, hence

$$x(t+2h) = x_1 + 2ch^5 + \mathcal{O}(h^6)\,, \qquad c = \text{const.}$$
$$= x_2 + c(2h)^5 + \mathcal{O}(h^6)$$

Conclusion: the local error $\epsilon \approx ch^5$ is given by

$$\epsilon \approx \frac{1}{30}(x_1 - x_2)\,.$$

Adaptive methods: RK4 with variable step size

Define the “optimal” step size h' such that the error per unit t is given by some fixed precision δ $\Rightarrow$ the local error at each step is $\delta h'$.

- If $h < h'$: the calculation is more precise than necessary $\Rightarrow$ can save computing time by increasing h
- If $h > h'$: the calculation is not precise enough $\Rightarrow$ need to reduce h

The optimal step size h' satisfies the relation

$$\delta h' = |ch'^5| = |ch^5| \left(\frac{h'}{h}\right)^5 = \frac{1}{30}|x_1 - x_2| \left(\frac{h'}{h}\right)^5$$

Rearranging gives

$$\boxed{h' = h\rho^{1/4}, \qquad \rho = \frac{30h\delta}{|x_1 - x_2|}.}$$

Adaptive methods: RK4 with variable step size

Algorithm:

- Given $x(t)$, compute two approximations for $x(t+2h)$: First x_1 (with two steps of size h) and then x_2 (with a single step of size $2h$).
- Compute ρ from x_1, x_2 and h: $\rho = 30h\delta/|x_1 - x_2|$
    - If $\rho > 1$, then $h < h'$, so we may increase h by a factor $\rho^{1/4}$. Keep the approximation x_1 for $x(t+2h)$, and for the following step, replace $h \leftarrow h\rho^{1/4}$.
    - If however $\rho < 1$, the target accuracy has not been reached. Hence recompute x_1 after replacing $h \leftarrow h\rho^{1/4}$, and use this new x_1 for $x(t+2h)$.
- To prevent h from growing too quickly, if accidentally $\rho \gg 1$, one should limit the maximal rescaling factor in practice. E.g. increase h at most by a factor 3 at each step, even if $\rho^{1/4} > 3$.

This method can also be used for entire systems of ODEs.

Example: Simple gravity pendulum with adaptive RK4

In the above example: replace the loop over t by

```python
t = a          # initial time
tpoints = []   # intermediate times (a priori unknown!)
delta = 1.0E-5    # desired precision

while t < b:
   thetapoints += [r[0]]
   omegapoints += [r[1]]
   tpoints += [t]
   r1 = rk4step(rk4step(r, t, h), t + h, h) # two RK4 steps
   r2 = rk4step(r, t, 2*h)  # one double-size RK4 step
   rho = 30 * h * delta / abs(r2[0] - r1[0])
   if rho < 1: # not precise enough: reduce h, recompute r1
      h *= rho**(1/4)
      r1 = rk4step(rk4step(r, t, h), t + h, h)
      t += 2*h
   else:      # too precise: increase h for next iteration
      t += 2*h
      h *= min(rho**(1/4), 3.0)  # (by a factor 3 at most)
   r = r1
```

Example: Simple gravity pendulum with adaptive RK4

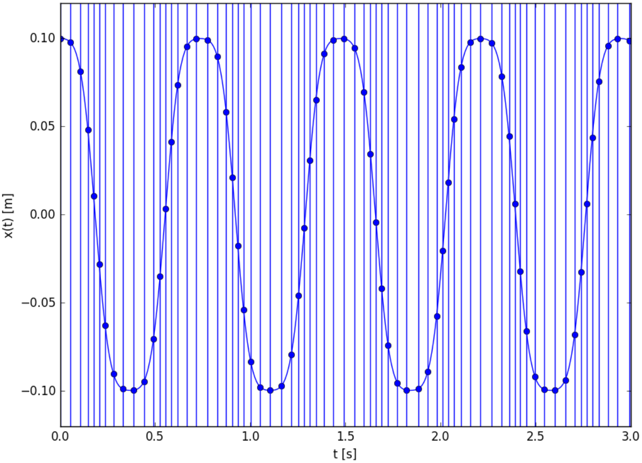
Here the task of the function `rk4step()` is to compute a single RK4 step:

```python
def rk4step(r, t, h):
    k1 = h * f(r, t)
    k2 = h * f(r + k1/2, t + h/2)
    k3 = h * f(r + k2/2, t + h/2)
    k4 = h * f(r + k3, t + h)
    return r + (k1 + 2*k2 + 2*k3 + k4)/6
```

In this example, the error has been calculated using the discrepancy in θ only. Depending on the problem to be solved, it may be necessary to take also the error in ω into account (and, more generally, the error in all the unknown functions).

Example: Simple gravity pendulum with adaptive RK4

Setting $\delta = 10^{-7}$, and plotting one out of five points in t:



Example: Simple gravity pendulum with adaptive RK4

Exercises

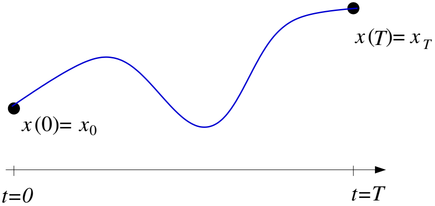
Use the adaptive RK4 method to compute the trajectory of a comet. Neglect the force exerted by the comet on the sun, as well as all other celestial bodies, and use a coordinate system such that the sun is at the center and the movement is in the (x, y) plane. Newton's gravitational constant is $G = 6.67408 \times 10^{-11}$ m^3 kg^{-1} s^{-2} and the solar mass is $M = 1.989 \cdot 10^{30}$ kg.

- Find the ODEs governing the comet's coordinates $x(t)$ and $y(t)$ as a function of time.
- Transform them into a system of four first-order ODEs.
- Write a program which solves these ODEs for the initial conditions $x(0) = 4 \cdot 10^9$ km, $y(0) = 0$, $\dot{x}(0) = 0$, $\dot{y}(0) = 500$ m s^{-1}, using the fixed-step RK4 method. Plot the trajectory $x(y)$; choose h small enough such that the orbit does not visibly change between two turns.
- Write a program which repeats this computation with the adaptive RK4 method. The precision is $\delta =$1 km/year. Compare the run-time with that of your first program.

Boundary-value problems

In general, for n first-order ODEs, one needs to specify n integration constants to have a unique solution. But what if these are not all given at the same point $t = 0$?

E.g. two-point boundary value problem for second-order ODE: $x(0)$ and $x(T)$ given, but $\dot{x}(0)$ initially unknown



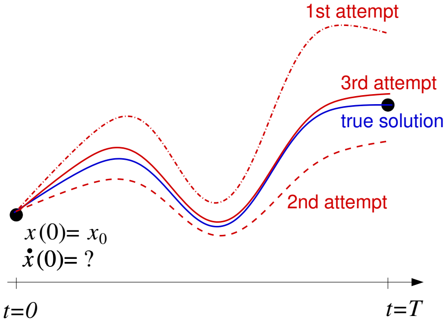
Two options:

- Start with a trial solution satisfying the ODE but not (all) the boundary conditions. Iteratively adjust to also satisfy the boundary conditions. Shooting method.
- Start with a trial solution satisfying the boundary conditions but not the ODE. Iteratively adjust to also satisfy the ODE. Relaxation method, will discuss this more in the chapter on partial differential equations.

Boundary-value problems: Shooting method

Shooting method: $x(0) \equiv x_0$ and $x(T) \equiv x_T$ given, but $\dot{x}(0)$ initially unknown

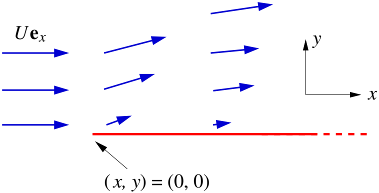
- Start at $t = 0$ with x_0 and some trial $\dot{x}(0)$
- Solve initial value problem to compute $x(T)$
- Adjust $\dot{x}(0)$ and repeat until $x(T) = x_T$ is obtained (using a root-finding algorithm)



Boundary-value problems

Example: Blasius boundary layer flow

Consider an incompressible fluid flowing past a semi-infinite rigid plate in the $(x \geq 0, z)$ half-plane. For $x < 0$, the fluid velocity is $\mathbf{u} = U\,\mathbf{e}_x$. What's $\mathbf{u}$ for $x > 0$?



It can be shown → (hydrodynamics lecture, F. Geniet) that the Navier-Stokes equations reduce to an ODE for this system. More precisely: There exists a function $f(\eta)$ such that

$$u_x = U\,f'(\eta)\,, \qquad u_y = \frac{1}{2}\sqrt{\frac{U\nu}{x}}\,\big(\eta f'(\eta) - f(\eta)\big)$$

where $\eta = \sqrt{\frac{U}{\nu}}\frac{y}{\sqrt{x}}$ and ν is the viscosity. This function solves the Blasius equation

$$2\,f'''(\eta) + f''(\eta)f(\eta) = 0$$

subject to the boundary conditions

$$f(0) = 0\,, \qquad f'(0) = 0\,, \qquad \lim_{\eta\to\infty} f'(\eta) = 1\,.$$

Boundary-value problems: Shooting method

Transform to a system of three first-order equations:

$$2h' + hf = 0\,, \qquad g' = h\,, \qquad f' = g$$

with boundary conditions

$$f(0) = 0\,, \qquad g(0) = 0\,, \qquad g(\infty) = 1\,.$$

We expect the problem to be well-posed (3 ODEs, 3 boundary conditions) — but we cannot simply integrate starting from $\eta = 0$ because $h(0)$ is not given.

Strategy:

- Solving the ODE with some arbitrary value h_0 for $h(0)$ gives a solution with $g(\infty) = g_\infty$ (in general, $g_\infty \neq 1$).
- This defines a function $g_\infty(h_0)$.
- Find a zero of the function $h_0 \mapsto (g_\infty(h_0) - 1)$ by a standard root-finding method, e.g. bisection. For this value of h_0, one has $g_\infty(h_0) = 1$, and therefore the corresponding solution solves the boundary-value problem.

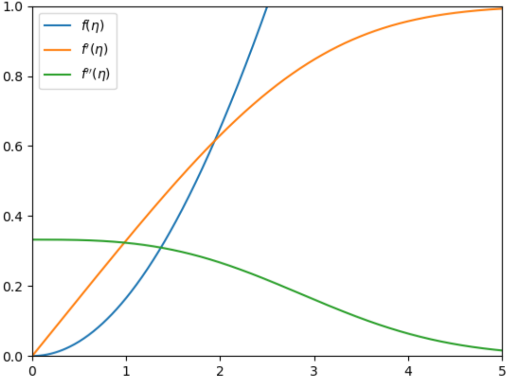
Boundary-value problems: Shooting method

```python
import numpy as np
import matplotlib.pyplot as plt
import rk4   # for the function rk4()
import zeros # for the function bisection()

max_eta = 100.  # choose this sufficiently big ("infinity")
g0, f0 = 0., 0. # Boundary conditions at eta=0
N = 10000       # Number of steps

def rhs(r, eta):  # the right-hand side of the ODE
   f, g, h = r # r = array containing f(eta), g(eta), h(eta)
   return np.array([g, h, -h*f/2]) # return [f', g', h']

def ginf(h0):  # compute g(max_eta) for some given h(0)
   r0 = np.array([f0, g0, h0]) # initial conditions at eta=0
   etapoints, rpoints = rk4.rk4(rhs, 0, max_eta, r0, N)
   rinf = rpoints[-1] # this is [f, g, h] at eta=max_eta
   return rinf[1]     # return g(max_eta)
```

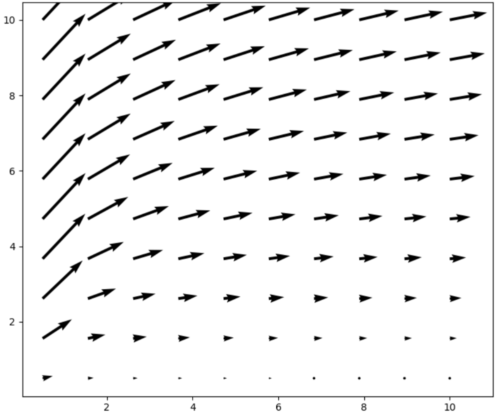
Boundary-value problems: Shooting method

```python
# find a zero of the function ginf(h0)-1 for 0 < h0 < 1
def ginf_minus_one(h0):
    return ginf(h0) - 1
h0 = zeros.bisection(ginf_minus_one, 0, 1)

# use this zero to construct the solution of the BVP
r0 = np.array([f0, g0, h0]) # ICs leading to the right BCs
etapoints, rpoints = rk4.rk4(rhs, 0, max_eta, r0, N)
fpoints = rpoints[:, 0]   # 1st column = values of f
dfpoints = rpoints[:, 1]  # 2nd column = values of g = f'
ddfpoints = rpoints[:, 2] # 3rd column = values of h = f''

# plot the result
plt.plot(etapoints, fpoints)
plt.plot(etapoints, dfpoints)
plt.plot(etapoints, ddfpoints)
plt.show()
```

Boundary-value problems: Shooting method



Boundary-value problems: Shooting method



Fluid velocity above the plate in the (x, y) plane

Ordinary differential equations

Some subjects we did not treat:

- advanced adaptive methods, e.g. the Bulirsch-Stoer method based on Richardson extrapolation
- predictor/corrector methods
- specialized algorithms for conservative problems, such as the leapfrog method
- specialized algorithms for stiff problems (involving several vastly different scales)
- and lots more. . .

Partial differential equations

In this chapter

- Classifying PDEs
- Finite-difference methods
- Boundary value problems: Jacobi method, successive overrelaxation
- Cauchy problems: FTCS method, Crank-Nicolson method, general stencils

Classifying PDEs

Consider a quasi-linear second-order PDE

$$\mathcal{D}\phi(t, x_i) = f(\phi,\, t,\, x_i)$$

where $\mathcal{D}$ is a second-order differential operator.

Mathematical classification with respect to the second-derivative structure:

- hyperbolic PDEs, e.g. the wave equation

$$\mathcal{D} = \frac{\partial^2}{\partial t^2} - c^2 \sum_i \frac{\partial^2}{\partial x_i^2}, \quad f = 0$$

- parabolic PDEs, e.g. the heat (diffusion) equation

$$\mathcal{D} = \frac{\partial}{\partial t} - a \sum_i \frac{\partial^2}{\partial x_i^2}, \quad f = 0$$

- elliptic PDEs, e.g. the Laplace equation

$$\mathcal{D} = \sum_i \frac{\partial^2}{\partial x_i^2}, \quad f = 0$$

Classifying PDEs

Classification in numerical analysis according to the type of numerical problem:

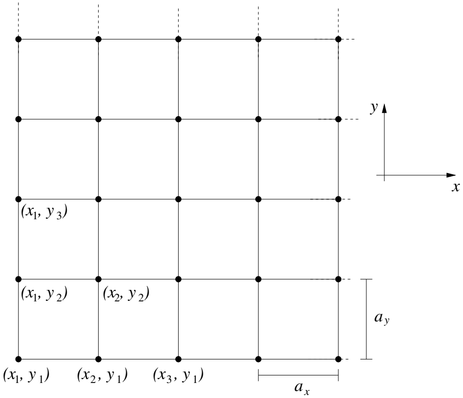
- Cauchy problem for hyperbolic and parabolic equations. To formulate a well-posed problem, specify initial conditions at some given t, and boundary conditions at the boundary of the spatial domain, then calculate the system's evolution with t. Dynamical problem.
- Boundary-value problem for elliptic equations. Specify boundary conditions at the boundary of the spatial domain, then calculate the values in its interior. Static problem.

Types of boundary conditions:

- Dirichlet conditions: specify the values of the unknown function ϕ.
- Neumann conditions: specify its derivatives.
- Mixed boundary conditions also possible.

Finite-difference methods

Discretize the solution region by constructing a lattice, e.g. a regular rectangular lattice in two dimensions:



Then approximate the derivatives at the nodes by difference quotients, see pp. 83, 84. One obtains a system of algebraic equations for the values of ϕ at the lattice nodes.

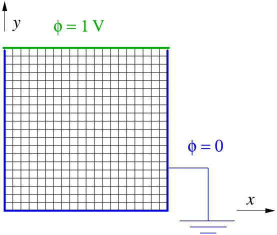
Numerical solution of the PDE = solution of this system of equations (for sufficiently small lattice spacing).

An exemplary boundary value problem: The 2D Laplace equation

We wish to solve the Laplace equation

$$\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)\phi(x,y)=0$$

on a square of 1×1 m, where the potential ϕ satisfies the boundary conditions $\phi(x, y = 1\,\mathrm{m}) = 1\,\mathrm{V}$ and $\phi(x, y = 0) = \phi(x = 0, y) = \phi(x = 1\,\mathrm{m}, y) = 0$.



An exemplary boundary value problem: The 2D Laplace equation

With a the lattice spacing for both dimensions, the discrete Laplacian is (see p.84)

$$\begin{aligned}
&\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)\phi(x,y)\\
&\quad\approx\frac{\phi(x+a,y)+\phi(x-a,y)-2\phi(x,y)}{a^2}+\frac{\phi(x,y+a)+\phi(x,y-a)-2\phi(x,y)}{a^2}\\
&\quad=\frac{\phi(x+a,y)+\phi(x-a,y)+\phi(x,y+a)+\phi(x,y-a)-4\phi(x,y)}{a^2}
\end{aligned}$$

hence the Laplace equation becomes

$$\phi(x,y)\approx\frac{1}{4}\left(\phi(x+a,y)+\phi(x-a,y)+\phi(x,y+a)+\phi(x,y-a)\right) \qquad (\star)$$

For $a=1$ cm, we have a 101×101 lattice (x,y). At the 400 points at the boundary, ϕ is given by the boundary conditions, $\phi(x,y)=0$ or $\phi(x,y)=1$ V. The other 9801 points satisfy $(\star)\Rightarrow$ a system of 9801 coupled linear equations.

An exemplary boundary value problem: The 2D Laplace equation

To solve the 9801 equations

$$\phi(x,y) \approx \frac{1}{4}\left(\phi(x+a,y)+\phi(x-a,y)+\phi(x,y+a)+\phi(x,y-a)\right)$$

we can use the Jacobi method:

- Start with some first guess for $\phi(x,y)$ at each node (not necessarily very precise)
- Compute

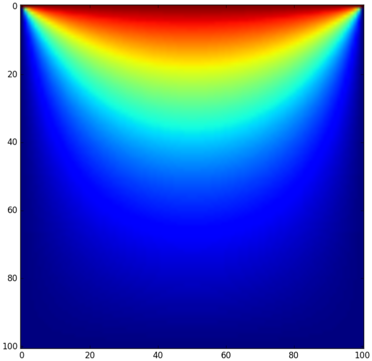
  $$\phi'(x,y) = \frac{1}{4}\left(\phi(x+a,y)+\phi(x-a,y)+\phi(x,y+a)+\phi(x,y-a)\right)$$

  at each interior node (x,y)
- If $||\phi-\phi'||<\delta$ (where δ is the desired precision): terminate.
  Otherwise repeat with $\phi \leftarrow \phi'$.

An exemplary boundary value problem: The 2D Laplace equation

```
import numpy as np

N = 100   # lattice = (N+1) x (N+1) square grid
delta = 1.E-5  # desired precision
U = 1.0        # potential at upper boundary

phi = np.zeros((N+1, N+1)) # the lattice
phi[0, :] = U              # boundary conditions
phiprime = np.copy(phi)    # second lattice of the same size
eps = delta + 1. # error, arbitrary here but must be > delta

while eps > delta:
   for i in range(1, N): # don't touch the boundary points
      for j in range(1, N):  # same here
         phiprime[i, j] = (phi[i+1, j] + phi[i-1, j] \
                         + phi[i, j+1] + phi[i, j-1]) / 4
   eps = np.max(np.abs(phi - phiprime))
   phiprime, phi = phi, phiprime
```

An exemplary boundary value problem: The 2D Laplace equation



Solution computed in ≈ 1 min. on a laptop.

The Jacobi (relaxation) method for boundary-value problems

Properties of the Jacobi method:

- Relaxation method: search for a fixed point ϕ_* of the operator $\frac{a^2}{4}\Delta_d + \mathbb{1}$, by applying it repeatedly on the starting configuration. Here Δ_d is the discretized Laplacian. If

$$\left(\frac{a^2}{4}\Delta_d + \mathbb{1}\right)\phi_* = \phi_*,$$

  then $\Delta_d \phi_* = 0$.

- This will converge for (practically) any initial configuration $\Rightarrow$ stable.
- Two sources of numerical error:
  1. limited precision of the iterative solution $(\delta > 0)$
  2. derivatives approximated by finite differences $(a > 0)$.
- Main drawback: slow.

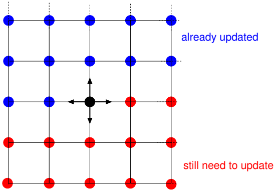
Successive overrelaxation

Same principle as for Jacobi method but two improvements:

- For the Jacobi method, at each iteration we computed

$$\phi'(x,y) = \frac{1}{4}\left(\phi(x+a,y) + \phi(x-a,y) + \phi(x,y+a) + \phi(x,y-a)\right)$$

  New value of ϕ at (x,y) = average of old values at neighbouring points.
  Idea ("Gauss-Seidel method"): Compute this average with the new values of ϕ (to the extent that they are already known) ⇒ better approximation



- Overrelaxation: to accelerate convergence, choose some parameter $\omega > 0$ and set

$$\phi(x,y) \leftarrow \frac{1+\omega}{4}\left(\phi(x+a,y) + \phi(x-a,y) + \phi(x,y+a) + \phi(x,y-a)\right) - \omega\phi(x,y)$$

Successive overrelaxation

Algorithm for solving the 2D Laplace equation on a square grid:

- Start with some first guess for $\phi(x, y)$ at each node (not necessarily very precise)
- At each interior node, successively replace

  $$\phi(x, y) \leftarrow \frac{1+\omega}{4}\left(\phi(x+a, y)+\phi(x-a, y)+\phi(x, y+a)+\phi(x, y-a)\right)-\omega\phi(x, y)$$

  Memorize the maximal discrepancy $\epsilon_{\max}$ between the old and new values among all lattice points.
- If $\epsilon_{\max} < \delta$, terminate. Otherwise, repeat.

Remarks

- Choice of ω: One can show that the method is stable for $\omega < 1$. Larger values of ω will speed up convergence. So choose $\omega < 1$ but close to 1.
- Gauss-Seidel: No need to allocate an extra lattice ϕ' → algorithm more efficient “in space”, consumes less memory
- Disadvantage of Gauss-Seidel: not parallelizable

Successive overrelaxation

Exercises

- Write a program which recalculates the solution of the boundary-value problem of p. 132, using successive overrelaxation. Experiment with several values of ω and study the effect on the speed of convergence. Plot the solution.
- Write a program which solves the Poisson equation in two dimensions,

  $$\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)\phi(x,y)=-\frac{\rho(x,y)}{\epsilon_0}$$

  where ρ is a charge density and ϵ_0 is the permittivity. Use the successive overrelaxation method.

The Jacobi method revisited

The Jacobi method is parallelizable: The computer can calculate several values of ϕ' simultaneously (since they only depend on ϕ, which does not change while computing ϕ').

Compare our old code for the Jacobi method

```
for i in range(1, N):
    for j in range(1, N):
        phiprime[i, j] = (phi[i+1, j] + phi[i-1, j]
                         + phi[i, j+1] + phi[i, j-1]) / 4
```

with the following code which exploits Numpy's vectorization capabilities:

```
phiprime[1:N, 1:N] = (phi[:N-1, 1:N] + phi[2:, 1:N]
                      + phi[1:N, :N-1] + phi[1:N, 2:]) / 4
```

With this modification, the code will run even faster than with successive overrelaxation.

But: Overrelaxation with Jacobi (i.e. without Gauss-Seidel) ⇒ instability

Comparison

For the above problem in electrostatics, 101×101 points, precision 10^{-5}, on my laptop:

method	running time
Jacobi, no vectorization	64 s
Jacobi with vectorization	0.75 s
successive overrelaxation $\omega = 0.95$	4.4 s

- **Jacobi:** stable, parallelizable, may be unstable with overrelaxation, needs two lattices ϕ and ϕ'
- **Gauss-Seidel:** stable, not parallelizable, stable with overrelaxation (= successive overrelaxation), a single lattice ϕ is enough

Matrix representation of elliptic operators

The Laplace equation discretized on an $(N+1)\times(N+1)$ lattice can be written

$$\phi_{i+1,\,j}+\phi_{i-1,\,j}+\phi_{i,\,j+1}+\phi_{i,\,j-1}-4\phi_{i,j}=0 \qquad (i,\,j=1\ldots N-1)$$

Set of linear equations with unknowns ϕ_{ij}. Rewrite by putting the $\phi_{i,\,j}$ into a vector $\mathbf{u}=(u_n)$:

$$u_n=\phi_{ij} \text{ where } n=i(N+1)+j+1 \qquad (n=1,\ldots,(N+1)^2)$$

The discretized Laplace equation (satisfied at the interior points, $1\le i\le N-1$ and $1\le j\le N-1$) becomes

$$u_{n+N+1}+u_{n-N-1}+u_{n+1}+u_{n-1}-4u_n=0\,,$$

while at the boundary we have

$$u_n=b_n \text{ (fixed by boundary conditions)}\,, \qquad i=0 \text{ or } N, \text{ or } j=0 \text{ or } N\,.$$

Now write these linear equations for the components of the vector $\mathbf{u}$ using matrix notation.

Matrix representation of elliptic operators

Matrix equation:

$$\mathbf{A}\mathbf{u} = \mathbf{b}$$

where $\mathbf{b}$ contains the boundary data, and $\mathbf{A}$ is given in block matrix form as

$$\mathbf{A} = \begin{pmatrix} \mathbb{1} & & & & & \\ \tilde{\mathbb{1}} & \mathbf{T} & \tilde{\mathbb{1}} & & & \\ & \tilde{\mathbb{1}} & \mathbf{T} & \tilde{\mathbb{1}} & & \\ & & & \ddots & & \\ & & & \tilde{\mathbb{1}} & \mathbf{T} & \tilde{\mathbb{1}} \\ & & & & & \mathbb{1} \end{pmatrix}$$

$\mathbb{1} =$ the $(N+1) \times (N+1)$ unit matrix, and $\mathbf{T}$ and $\tilde{\mathbb{1}}$ are $(N+1) \times (N+1)$ matrices:

$$\mathbf{T} = \begin{pmatrix} 1 & & & & & \\ 1 & -4 & 1 & & & \\ & 1 & -4 & 1 & & \\ & & & \ddots & & \\ & & & 1 & -4 & 1 \\ & & & & & 1 \end{pmatrix}, \qquad \tilde{\mathbb{1}} = \begin{pmatrix} 0 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & \ddots & & \\ & & & & 1 & \\ & & & & & 0 \end{pmatrix}$$

To compute $\phi(x, y)$ at the interior points, one must solve this matrix equation.

Matrix representation of elliptic operators

- A similar structure is obtained for any boundary-value problem in d dimensions, when discretized on a rectangular $N_1 \times \ldots \times N_d$ lattice
- Possible strategies:
  - relaxation methods, such as Jacobi or Gauss-Seidel
  - direct methods from numerical linear algebra:
    Gaussian elimination, or better methods optimized for sparse coefficient matrices $\mathbf{A}$
  - Fourier analysis (expanding in eigenfunctions of the differential operator)

Cauchy problems

Dynamical problem for parabolic and hyperbolic PDEs:

- Specify initial conditions at $t = 0$, then compute the solution for $0 \leq t \leq T$.
- At the spatial boundary of the solution region, must also provide boundary conditions.
- This will (usually) determine the solution uniquely. In contrast to elliptic PDEs, boundary conditions are not given at the entire boundary of the solution region: no conditions at $t = T$.
- Existence/unicity/regularity: finding necessary/sufficient conditions can be a hard mathematical problem (e.g. Navier-Stokes equations. . .)

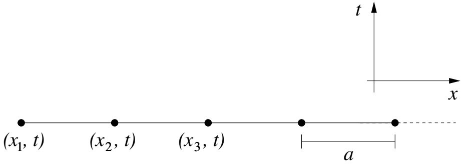
FTCS method

Prime example of a parabolic PDE: heat equation in $1+1$ dimensions

$$\frac{\partial \phi}{\partial t} - D\frac{\partial^2 \phi}{\partial x^2} = 0$$

D = diffusion coefficient.

Discretization in x: N evenly-spaced points (x_n) in the solution interval (distance a).



Finite-difference approximation in space:

$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{\phi(x+a,t)+\phi(x-a,t)-2\phi(x,t)}{a^2}$$

→ need to solve N ordinary differential equations ($\phi_n \equiv \phi(x_n)$)

$$\frac{\partial \phi_n}{\partial t}(t) = \frac{D}{a^2}\left(\phi_{n+1}(t)+\phi_{n-1}(t)-2\phi_n(t)\right)$$

FTCS method

$$\frac{\partial \phi_n}{\partial t}(t) = \frac{D}{a^2}\left(\phi_{n+1}(t) + \phi_{n-1}(t) - 2\phi_n(t)\right)$$

Solution with Euler's method. Imprecise, but the error is typically dominated by the spatial discretization → no need for higher-order methods. (Otherwise: Crank-Nicolson method, later.)

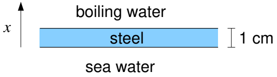
For a time increment h:

$$\boxed{\phi_n(t+h) \approx \phi_n(t) + h\frac{D}{a^2}\left(\phi_{n+1}(t) + \phi_{n-1}(t) - 2\phi_n(t)\right)}$$

This is called the foward-time centered-space (FTCS) method.

Example of a Cauchy problem: Heat equation

After a boiler explosion, the floor of the boiler room in a steamship is covered with water of temperature 100° C. The ship's hull is constructed from steel plates of width 1 cm; the temperature of the sea water on the other side is 10° C. We wish to find the temperature profile within the steel plates as a function of time. Initially at $t = 0$, the temperature was uniformly given by that of the sea water. The heat diffusion coefficient of steel is $4.25 \cdot 10^{-6}$ m^2 s^{-1}.



Subdivide x into 100 subintervals:

- 101 points $x_0 \dots x_{100}$
- Boundary conditions: $T(x_0) = 10^\circ\,\mathrm{C}$ and $T(x_{100}) = 100^\circ\,\mathrm{C}$ independently of time
- Initial conditions: The temperature at $t = 0$ is $10^\circ\,\mathrm{C}$ (except for x_{100})

Example of a Cauchy problem: Heat equation

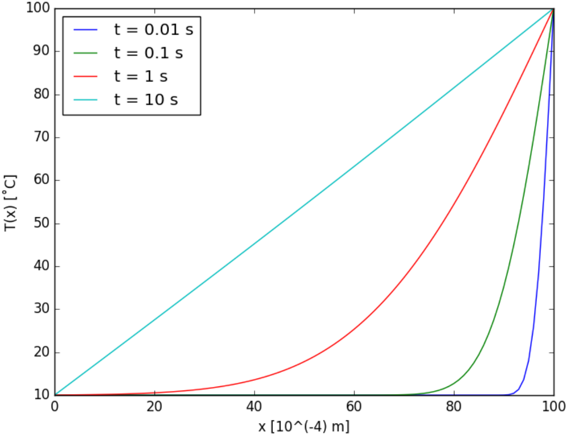
```
import numpy as np
import matplotlib.pyplot as plt

D = 4.25E-6  # diffusion coefficient
w = .01      # width of spatial interval
T0 = 283     # T at x=0
T1 = 373     # T at x=w
N = 100      # spatial discretization
a = w/N      # lattice constant
tmax = 10    # solution sought for 0 <= t <= tmax
h = 1.E-3    # time step
t = 0
c = h * D / a**2

# initial temperature profile at t = 0
T = np.ones(N+1)
T *= T0
T[N] = T1
tplot = [.01, .1, 1, 10]  # intermediate times for plotting
```

Example of a Cauchy problem: Heat equation

```
# main loop
while t < tmax:
  T[1:N] = T[1:N] + c * (T[:N-1] + T[2:] - 2 * T[1:N])
    for tp in tplot: # t (close to) where we want to plot?
      if abs(t - tp) < 0.1 * h:
      plt.plot(T - 273., label = "t = " + str(tp) + " s")
  t += h

plt.show()
```

Note that by the instruction
`T[1:N] = T[1:N] + c * (T[:N-1] + T[2:]  - 2 * T[1:N])`
we add to all interior values `T[n]` the quantities `c * (T[n-1] + T[n+1] - 2 * T[n])`
simultaneously (vectorisation).

Example of a Cauchy problem: Heat equation



Instability of the FTCS method

The FTCS method works well for simple parabolic problems, but it may become unstable for other Cauchy problems, e.g. hyperbolic ones. Try to apply it to the wave equation in 1+1 dimensions,

$$\frac{\partial^2}{\partial t^2}\phi(t,x) = c^2\,\frac{\partial^2}{\partial x^2}\phi(x,t)\,,$$

using the same strategy as for the heat equation:

- discretize in space, replace $\frac{\partial^2}{\partial x^2}$ by N finite differences

$$\frac{\partial^2}{\partial x^2}\phi \approx \frac{\phi(x+a,t)+\phi(x-a,t)-2\phi(x,t)}{a^2}$$

- obtain N coupled ODEs in time (now of second order)

$$\ddot{\phi}_n(t) = \frac{c^2}{a^2}\left(\phi_{n+1}(t)+\phi_{n-1}(t)-2\phi_n(t)\right)$$

- therefore, transform into $2N$ first-order ODEs:

$$\psi_n(t) = \dot{\phi}_n(t)\,, \qquad \dot{\psi}_n(t) = \frac{c^2}{a^2}\left(\phi_{n+1}(t)+\phi_{n-1}(t)-2\phi_n(t)\right)$$

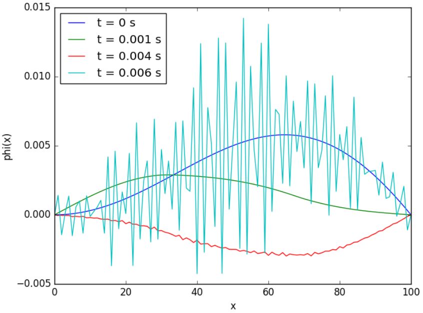
Instability of the FTCS method

FTCS equations:

$$\phi_n(t+h) = \phi_n(t) + h\psi_n(t)$$

$$\psi_n(t+h) = \psi_n(t) + \frac{hc^2}{a^2}\Big(\phi_{n+1}(t) + \phi_{n-1}(t) - 2\phi_n(t)\Big)$$

Application: vibrating string of length L, both ends fixed (boundary conditions) and some given elongation profile with zero velocity at $t = 0$ (initial conditions). Plotting the solution at four different times, ≤ 1 oscillation period:



Visibly unstable.

Von Neumann stability analysis

Expand $\phi(x,t)$ in Fourier modes,

$$\phi(x,t) = \sum_k c_k(t) e^{ikx} \,.$$

For differential operators which are diagonal in Fourier space, no mixing between the modes, and thus

$$c_k(t+h) = \xi_k(h,a) c_k(t)$$

If the amplification factors ξ_k satisfy $|\xi_k| \leq 1 \; \forall\, k$, the method is stable. This is the case for the heat equation if $h < \frac{a^2}{2D}$. Proof:

$$\begin{aligned} a\frac{\partial}{\partial t}\sum_k c_k e^{ikx} &= D\,\frac{\partial^2}{\partial x^2}\sum_k c_k e^{ikx} \\ \Rightarrow \quad \dot{c}_k e^{ikx} &= D\, c_k \left(\frac{e^{ik(x+a)} + e^{ik(x-a)} - 2e^{ikx}}{a^2} \right) \\ \Rightarrow \quad \dot{c}_k &= \frac{D}{a^2}\, c_k \left(e^{ika} + e^{-ika} - 2 \right) = \frac{2D}{a^2} c_k \left(\cos(ka) - 1\right) = \frac{4D}{a^2} \sin^2\left(\frac{ka}{2}\right) c_k \end{aligned}$$

and since $c_k(t+h) \approx c_k(t) + h\dot{c}_k(t)$:

$$c_k(t+h) \approx \left(1 - h\frac{4D}{a^2} \sin^2\left(\frac{ka}{2}\right)\right) c_k(t) \quad \Rightarrow \quad |\xi_k| = \left| 1 - h\frac{4D}{a^2} \sin^2\left(\frac{ka}{2}\right) \right| .$$

Von Neumann stability analysis

Still with

$$\phi(x,t) = \sum_k c_k(t) e^{ikx}\,, \qquad c_k(t+h) = \xi_k(h,a) c_k(t) \;:$$

- If $|\xi_k| > 1$ for at least one k, the method is unstable (exponentially growing mode).
- For coupled systems, the ξ_k are matrices and the stability conditions apply to their eigenvalues.
- Example: Wave equation, the ξ_k are 2×2 matrices acting on the Fourier modes of $(\phi(x,t),\, \psi(x,t))$.
  Eigenvalues $= 1 \pm i \frac{2hc}{a} \sin \frac{ka}{2} \Rightarrow$ unstable.

Implicit method

With the implicit (backward) Euler's method: "BTCS" (backward-time centered-space) equations

$$
\begin{aligned}
\phi_n(t+h) &= \phi_n(t) + h\psi_n(t+h) \\
\psi_n(t+h) &= \psi_n(t) + \frac{hc^2}{a^2}\Big(\phi_{n+1}(t+h) + \phi_{n-1}(t+h) - 2\phi_n(t+h)\Big)
\end{aligned}
$$

$\phi_n(t+h)$ and $\psi_n(t+h)$ no longer explicitly given by the state of the system at time t, but implicitly by a system of $2N$ linear equations which must be solved:

$$
\begin{pmatrix}
 & & \ddots & & & & & \\
-\alpha & 0 & 2\alpha & 1 & -\alpha & & & \\
 & & & & 1 & -h & & \\
 & & -\alpha & 0 & 2\alpha & 1 & -\alpha & \\
 & & & & & & 1 & -h \\
 & & & & & & & \ddots
\end{pmatrix}
\begin{pmatrix}
\vdots \\ \phi_{n-1} \\ \psi_{n-1} \\ \phi_n \\ \psi_n \\ \phi_{n+1} \\ \psi_{n+1} \\ \vdots
\end{pmatrix}_{t+h}
=
\begin{pmatrix}
\vdots \\ \phi_{n-1} \\ \psi_{n-1} \\ \phi_n \\ \psi_n \\ \phi_{n+1} \\ \psi_{n+1} \\ \vdots
\end{pmatrix}_{t}
$$

where $\alpha = hc^2/a^2$. This can be solved e.g. by Gaussian elimination, or (better) by a method optimized for sparse matrices.

Crank-Nicolson method

One may show: The BTCS method is always stable. Its promlems are its much slower speed (since a linear system must be solved at each step) and, for the wave equation, the damping of the Fourier modes (amplification factors $< 1 \Rightarrow$ exponentially decreasing modes, still not the physical solution).

A more efficient scheme (of order h^2 in time) in which this latter problem is absent is the Crank-Nicolson method: Take the average of the FTCS and BTCS formulas. For the wave equation this reads

$$\boxed{\begin{aligned}\phi_n(t+h) &= \phi_n(t) + h\,\frac{1}{2}\left(\psi_n(t+h) + \psi_n(t)\right)\\ \psi_n(t+h) &= \psi_n(t) + \frac{hc^2}{a^2}\,\frac{1}{2}\Big(\phi_{n+1}(t+h) + \phi_{n-1}(t+h) - 2\phi_n(t+h)\\ &\qquad + \phi_{n+1}(t) + \phi_{n-1}(t) - 2\phi_n(t)\Big)\end{aligned}}$$

This scheme is still implicit: at each step a system of equations must be solved for $\phi_n(t+h)$ and $\psi_n(t+h)$.

Crank-Nicolson method

Matrix equation for the Crank-Nicolson method applied to the wave equation: Set

$$\mathbf{u}(t) = \begin{pmatrix} \phi_0(t) \\ \psi_0(t) \\ \phi_1(t) \\ \psi_1(t) \\ \vdots \\ \phi_N(t) \\ \psi_N(t) \end{pmatrix}$$

C-N equations:

$$\mathbf{A}\mathbf{u}(t+h) = \mathbf{B}\mathbf{u}(t)$$

where

$$\mathbf{A} = \begin{pmatrix} & & \ddots & & & & & \\ -\alpha & 0 & 2\alpha & 1 & -\alpha & & & \\ & & & & 1 & -h/2 & & \\ & & -\alpha & 0 & 2\alpha & 1 & -\alpha & \\ & & & & & & 1 & -h/2 \\ & & & & & & & \ddots \end{pmatrix},$$

$\alpha = hc^2/2a^2$, and $\mathbf{B}$ is obtained from $\mathbf{A}$ by substituting $h \to -h$, $\alpha \to -\alpha$.

Crank-Nicolson method

Matrix equation for the Crank-Nicolson method applied to the wave equation: Given $\mathbf{u}(t)$, find $\mathbf{u}(t+h)$ as a solution of

$$\mathbf{Au}(t+h) = \mathbf{Bu}(t)$$

The $\mathbf{A}$ and $\mathbf{B}$ matrices have a sparse structure: Almost all elements are zero, except for some near the diagonal. Specialized methods can therefore solve the C-N equations in run-time $\Theta(N)$ (rather than the $\Theta(N^3)$ of general methods, such as Gaussian elimination).

We have not discussed numerical linear algebra in this course, therefore we will simply use the function `numpy.linalg.solve()` provided by `NumPy` (although it is inefficient for sparse matrices) to solve the C-N equations.

Algorithm:

- Initialize the vector $\mathbf{u}$, construct the matrices $\mathbf{A}$ and $\mathbf{B}$
- At each t, solve the C-N equations to obtain $\mathbf{u}(t+h)$. The quantities ϕ_0, ψ_0, ϕ_N, and ψ_N are always given by the boundary conditions.

Crank-Nicolson method

Initialization:

```
import numpy as np
import matplotlib.pyplot as plt

L = .3       # string length
c = 100.     # phase velocity
N = 100      # number of discretization intervals
a = L/N      # lattice constant
tmax = .1    # compute the solution for 0 <= t <= tmax
h = 1.E-4    # time increment
alpha = h * c**2 / 2 / a**2

# some initial profile
u = np.zeros(2*N + 2)
for k in range(1, N+1):
    u[2*k] = np.sin(k / N * np.pi) * k / N / 100
tplot = [0, .01, .02, .03]  # plot at these times
```

Crank-Nicolson method

Constructing the matrices A and B:

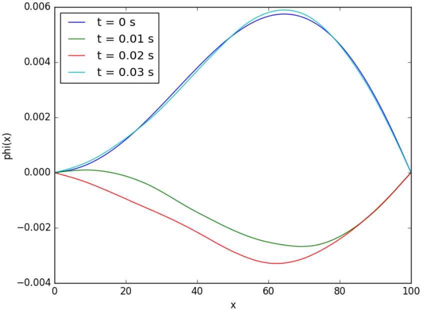
```python
A = np.zeros([2*N + 2, 2*N + 2])
B = np.zeros([2*N + 2, 2*N + 2])
# 2x2 blocks on top left and bottom right: unit matrices
# (at both ends, sol. always given by boundary conditions)
A[0, 0] = A[1, 1] = A[2*N, 2*N] = A[2*N + 1, 2*N + 1] = 1.0
B[0, 0] = B[1, 1] = B[2*N, 2*N] = B[2*N + 1, 2*N + 1] = 1.0
# Other elements:
for i in range(2, 2*N):
    A[i, i] = 1.0
    B[i, i] = 1.0
    if i % 2 == 0:   # lines of even i: phi_n
        A[i, i+1] = -h/2
        B[i, i+1] = h/2
    else:            # lines of odd i: psi_n
        A[i, i+1] = -alpha
        B[i, i+1] = alpha
        A[i, i-1] = 2*alpha
        B[i, i-1] = -2*alpha
        A[i, i-3] = -alpha
        B[i, i-3] = alpha
```

Crank-Nicolson method

Main loop:

```
t = 0.0
while t < tmax:
    # in absence of an efficient solver for sparse systems:
    # use numpy.linalg.solve() for demonstration
    u = np.linalg.solve(A, B @ u)
    t += h
```

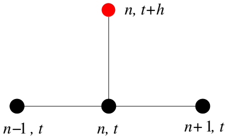
Result: no more instability (but the numerical error is visible after just 5 periods!)



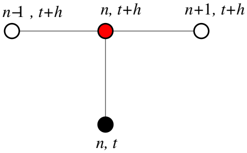
Stencil graphs

The different finite-difference methods can be represented graphically with the help of stencil diagrams:

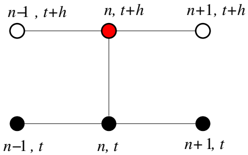
- FTCS



- BTCS



- Crank-Nicolson

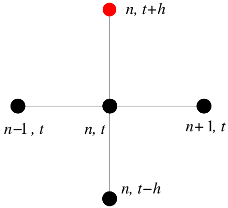


Explanation: To compute the value at ●, one needs the values at ● (known) and at ○ (initially unknown ⇒ implicit method). Horizontal/vertical line = discretized derivative in space/time

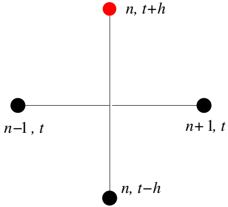
Stencil graphs

Other methods use two time steps:

- Richardson (unstable!)



- Dufort-Frankel (stable)



Neumann boundary conditions

Up to now, we have only regarded boundary conditions giving the value of the unknown function at the boundary of the solution region: Dirichlet boundary conditions.
Examples:

- electrostatic potential Φ on a conductive surface
- temperature on a surface separating two media
- zero amplitude on both ends of a string

It may also happen that, instead, its derivative is given: Neumann boundary conditions.
Examples:

- a fixed surface charge determines the electric field $\vec{E} = -\vec{\nabla}\Phi$ in electrostatics
- standing sound wave in an organ pipe with a closed end
  $\Rightarrow$ pressure maximum, $\partial p = 0$

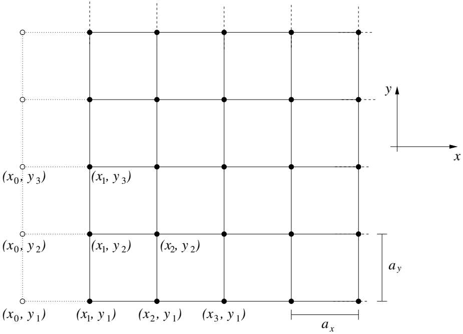
Neumann boundary conditions

Simplest case:

$$\left.\frac{\partial \phi}{\partial x}\right|_{\text{boundary}} = 0$$

i.e. the function $\phi(x, y)$ is approximately constant along the x direction at the boundary.

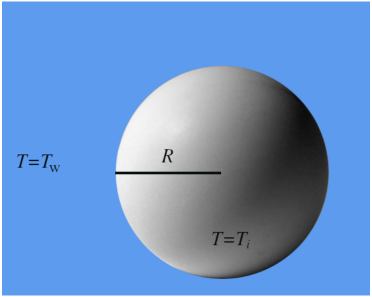
Numerical treatment with ghost lattice cells



Set $\phi(x_0, y) = \phi(x_1, y)$ (backward difference at x_1) or $\phi(x_0, y) = \phi(x_2, y)$ (central difference at x_1). The value of ϕ at x_0 is unphysical (outside the solution region), it only serves to enforce $\left.\frac{\partial \phi}{\partial x}\right|_{x_1} = 0$.

Example: Cooling of a homogeneous ball

A homogeneous ball of radius R and initial temperature $T_{\rm i}$ is submerged in water of temperature $T_{\rm w}$.



We would like to compute $T(r,\theta,\phi,t) = T(r,t)$ (spherical symmetry) in the interior. The $3+1$-dimensional heat equation in spherical coordinates reads

$$\frac{\partial T}{\partial t} - D\Delta T = 0 \Leftrightarrow \frac{\partial T}{\partial t} - D\left(\frac{\partial^2 T}{\partial r^2} + \frac{2}{r}\frac{\partial T}{\partial r}\right) = 0$$

where we have used that the angular derivatives are zero by spherical symmetry.

Example: Cooling of a homogeneous ball

The initial condition is

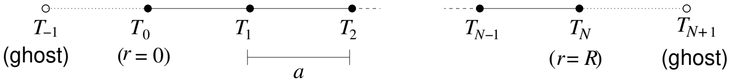
$$T(r,0) = T_{\mathrm{i}}.$$

Neumann boundary conditions:

- At $r = 0$:
$$\left.\frac{\partial T}{\partial r}\right|_{r=0} = 0 \qquad\qquad \text{(symmetry/continuity)}$$
- At $r = R$:
$$\left.\frac{\partial T}{\partial r}\right|_{r=R} = c\,(T_{\mathrm{w}} - T(R,t))$$
where $c =$ cooling constant. Unlike the previous example, here the heat exchange with the environment is no longer instantaneous ($(T(R,t) = T_{\mathrm{w}})$) but the thermal transport coefficient across the surface is finite.

Example: Cooling of a homogeneous ball

Spatial discretization:



Discrete Laplacian in spherical coordinates:

At $r = na \neq 0$:

$$\frac{\partial^2 T}{\partial r^2} + \frac{2}{r}\frac{\partial T}{\partial r} \rightarrow \frac{T_{n+1} + T_{n-1} - 2T_n}{a^2} + \frac{T_{n+1} - T_{n-1}}{n\, a^2}$$

At $r = 0$:

$$\lim_{r \to 0}\left(\frac{\partial^2 T}{\partial r^2} + \frac{2}{r}\frac{\partial T}{\partial r}\right) \overset{\text{L'Hôpital}}{=} 3\left.\frac{\partial^2 T}{\partial r^2}\right|_{r=0} \rightarrow 3\frac{T_1 + T_{-1} - 2T_0}{a^2}$$

Discretized boundary conditions:

At $r = 0$:

$$\left.\frac{\partial T}{\partial r}\right|_{r=0} = 0 \;\Rightarrow\; T_{-1} = T_1 \quad \text{(central derivative for } n = 0 \text{ vanishes)}$$

At $r = R$:

$$\left.\frac{\partial T}{\partial r}\right|_{r=R} = c\,(T_{\mathrm{w}} - T(R)) \;\Rightarrow\; T_{N+1} = T_{N-1} + 2\,ac(T_{\mathrm{w}} - T_N)$$

Example: Cooling of a homogeneous ball

Use the Crank-Nicolson method.

For $n = -1$ (ghost node):

$$T_{-1}(t+h) = T_1(t+h) \qquad \text{(boundary condition)}$$

For $n = 0$:

$$T_0(t+h) = T_0(t) + \frac{Dh}{2a^2}\Big(3T_1(t) + 3T_{-1}(t) - 6T_0(t) + 3T_1(t+h) + 3T_{-1}(t+h) - 6T_0(t+h)\Big)$$

$$\Rightarrow \left(-3\alpha T_{-1} + (1+6\alpha)T_0 - 3\alpha T_1\right)\Big|_{t+h} = \left(3\alpha T_{-1} + (1-6\alpha)T_0 + 3\alpha T_1\right)\Big|_{t}$$

with $\alpha = \frac{Dh}{2a^2}$.

For $1 \le n \le N$:

$$\left(-\alpha\left(1-\frac{1}{n}\right)T_{n-1} + (1+2\alpha)T_n - \alpha\left(1+\frac{1}{n}\right)T_{n+1}\right)\Big|_{t+h} = \left(\alpha\left(1-\frac{1}{n}\right)T_{n-1} + (1-2\alpha)T_n + \alpha\left(1+\frac{1}{n}\right)T_{n+1}\right)\Big|_{t}$$

For $n = N+1$ (ghost node):

$$T_{N+1}(t+h) = T_{N-1}(t+h) - 2acT_N(1+h) + 2acT_{\mathrm{w}} \quad \text{(boundary condition)}$$

Example: Cooling of a homogeneous ball

Matrix form of C-N equations:

$$\mathbf{A}T(t+h) = \mathbf{B}T(t) + C$$

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & -1 & & & \\ -3\alpha & 1+6\alpha & -3\alpha & & & \\ & -\alpha(1-\frac{1}{1}) & 1+2\alpha & -\alpha(1+\frac{1}{1}) & & \\ & & -\alpha(1-\frac{1}{2}) & 1+2\alpha & -\alpha(1+\frac{1}{2}) & \\ & & & \ddots & \ddots & \\ & & & -\alpha(1-\frac{1}{N}) & 1+2\alpha & -\alpha(1+\frac{1}{N}) \\ & & & -1 & 2ac & 1 \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} 0 & & & & & \\ 3\alpha & 1-6\alpha & 3\alpha & & & \\ & \alpha(1-\frac{1}{1}) & 1-2\alpha & \alpha(1+\frac{1}{1}) & & \\ & & \alpha(1-\frac{1}{2}) & 1-2\alpha & \alpha(1+\frac{1}{2}) & \\ & & & \ddots & \ddots & \\ & & & \alpha(1-\frac{1}{N}) & 1-2\alpha & \alpha(1+\frac{1}{N}) \\ & & & & & 0 \end{pmatrix}$$

$$C = (0,\, \ldots\, 0,\, 2\,ac\,T_{\mathrm{w}})^{\mathrm{T}}$$

Example: Cooling of a homogeneous ball

Code:

```python
# Load libraries, define constants
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg as la

D = 4.E-6    # diffusion coefficient
c = 20.      # cooling constant
R = 5.E-2    # ball radius
Ti = 373.    # temperature at t=0
Tw = 290.    # temperature of surrounding medium

N = 50             # spatial discretization
a = R / N          # lattice constant
t, tmax = 0., 301. # time interval
h = .1             # time increment

# Construct an array to store the solution
T = Ti * np.ones(N + 3)  # N+1 points + 2 ghost points
```

Example: Cooling of a homogeneous ball

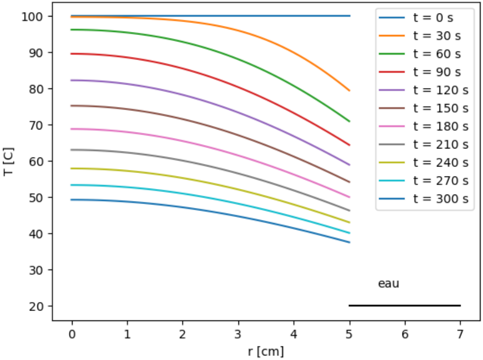
```
# Construct the matrix A
alpha = D * h / (2 * a**2)
A = (1 + 2 * alpha) * np.identity(N + 3)
for n in range(1, N+1):
    A[n+1, n] = -alpha * (1 - 1/n)
    A[n+1, n+2] = -alpha * (1 + 1/n)
A[0, 0] = 1.     # 1st line: boundary conds. at r=0
A[0, 2] = -1.
A[1, 0] = -3 * alpha     # 2nd line: Laplacian at r=0
A[1, 1] = 1 + 6 * alpha
A[1, 2] = -3 * alpha
A[-1, -1] = 1.   # last line: boundary conds. at r=R
A[-1, -2] = 2 * a * c
A[-1, -3] = -1.
```

Similarly for the matrix $\mathbf{B}$ and the vector C (not shown here)

Example: Cooling of a homogeneous ball

```
# solve the PDE and plot the result
nextplot, dtplot = 0., 30.    # plot every 30 secondes
r = np.linspace(0, R, N+1)    # spatial solution region

while t < tmax:
   if t >= nextplot:
      plt.plot(r, T[1:-1] - 273.)  # T[0], T[-1] = ghosts
      nextplot += dtplot
   t += h
   T = la.solve(A, B @ T + C)

plt.show()
```

Example: Cooling of a homogeneous ball



Partial differential equations

Subjects for which there was no time to mention:

- Pretty much everything — the literature about PDEs fills many library shelves.
- Other finite-differencing methods, in particular for first-order problems with flux conservation, problems in > 2 dimensions, multigrid methods. . .
- Other classes of methods: Finite elements, spectral methods . . .
- Applications to fluid dynamics, electrodynamics, general relativity . . .

Monte-Carlo methods

In this chapter

- Monte-Carlo integration
- Monte-Carlo sampling (Markov chain Monte Carlo)
- Simulated annealing
- Kinetic Monte Carlo

Monte-Carlo methods

Monte-Carlo (MC) methods are non-deterministic methods using random numbers to obtain numerical approximations. For certain problems, they are much more efficient than their deterministic counterparts. For example:

- **Integration**: Compute multi-dimensional integrals, integrate over nontrivial domains, deal with highly singular integrands
- **Sampling**: Generate samples of complicated multivariate probability density functions
- **Optimization**: Find an approximate global maximum of a complicated function with many local maxima

They are among the most important tools of modern computational physics. (Other applications in engineering, biology, finance, mathematics. . .)

Pseudo-random numbers

MC methods need large quantities of random numbers.

Problem:

It is fundamentally impossible to obtain truly random numbers from a deterministic algorithm.

Solutions:

- either use a physical random number generator based on quantum mechanics (true random numbers; may be slow and/or difficult)
- or use a deterministic algorithm to produce a sequence of numbers with approximately the same stochastic properties as a true random sequence: pseudo-random numbers

Pseudo-random numbers in Python

We will use pseudo-random numbers provided by the `NumPy` library:

- `numpy.random.random()` returns a pseudo-random number in the interval $[0.0, 1.0)$ (uniformly distributed)
- `numpy.random.random(n)` returns n pseudo-random numbers in the interval $[0.0, 1.0)$ (uniformly distributed)
- `numpy.random.choice(a)` returns a pseudo-randomly chosen element of the array `a`
- `numpy.random.randint(n)` returns a pseudo-random integer between 0 and $n - 1$ (uniformly distributed)

We will treat these functions as “black boxes” without studying the algorithms behind them.

The law of large numbers

Let $\{X_i\}_{i=1\ldots n}$ be a set of n independent random variables drawn from some probability distribution of expectation value $\langle X\rangle$.

The law of large numbers states that the average of the X_i (sample mean) tends towards $\langle X\rangle$ (expectation value of the distribution) as $n \to \infty$ "almost certainly", i.e. with probability 1:

$$\boxed{\Pr\left(\lim_{n\to\infty} \frac{X_1+\ldots+X_n}{n} = \langle X\rangle\right) = 1\,.}$$

In practice: To compute $\langle X\rangle$, randomly draw a large number of X_i and take their average.

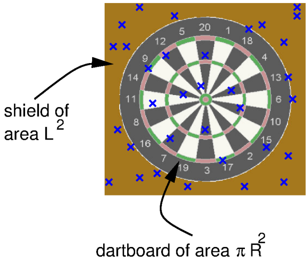
Error estimate: The difference $\frac{X_1+\ldots+X_n}{n} - \langle X\rangle$ tends to zero as $\frac{1}{\sqrt{n}}$ on average (central limit theorem; additive variances for independent random variables).

MC integration: Invitation

How to calculate the numerical value of π by Monte Carlo integration

You need:

- a dartboard of known radius
- a rectangular shield of known area
- a very bad darts player (= random number generator)



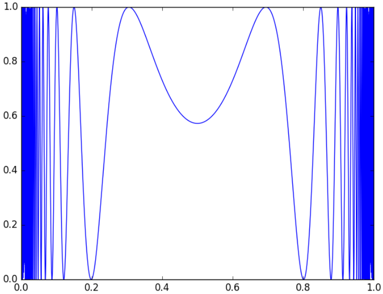
After a large number of darts have been thrown:

$$\frac{\text{number of darts on dartboard } N_d}{\text{number of darts on shield } N_s} \to \frac{\text{area of dartboard}}{\text{area of shield}} = \frac{\pi R^2}{L^2} \quad \Rightarrow \quad \pi \approx \frac{L^2 N_d}{R^2 N_s}$$

MC integration: difficult integrands

Consider the integral

$$I = \int_0^1 \sin^2\left(\frac{1}{x(1-x)}\right) \mathrm{d}x\,.$$



Integrand bounded and continuous ⇒ I exists ($I = 0.61515\ldots$). Results for $\delta \lesssim 10^{-3}$:

trapezoid	Simpson	Gauss-Legendre
0.6158	0.7207	$0.615\,(?)$
± 0.0001	± 0.0002	$\pm 0.001\,(?)$

Better result obtained by a Monte-Carlo method.

MC integration

Compute $I = \int_0^1 f(x)\,\mathrm{d}x$ by a stochastic method.

Naïve method:

- Generate N pairs (x_i, y_i) of random numbers, uniformly distributed over $(0,1) \times (0,1)$.
- Count the number K of points with $f(x_i) \geq y_i$.
- If N is large, the fraction of points which fall below the graph of f corresponds to the fraction of the area below the graph of f.
- Hence $I \approx K/N$.

```
N = 1000000
K = 0
for i in range(N):
    if f(np.random.random()) > np.random.random():
        K += 1
I = K/N
```

MC integration

Improved method:

Definition of the mean value of f between a and b:

$$\langle f \rangle = \frac{1}{b-a} \int_a^b f(x) \, \mathrm{d}x$$

Estimate $\langle f \rangle$ by

$$\langle f \rangle \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i)$$

where the x_i are uniformly distributed random numbers in $[a, b]$. Thus

$$I \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i) \, .$$

```
N = 10000000
favg = 0.0
for i in range(N):
    favg += f(np.random.random())
I = favg/N
```

Result $I = 0.6151 \pm 0.0002$.

MC integration

- The relative error for both methods is $\sim 1/\sqrt{N}$, but the coefficient is smaller for the improved method ($\Rightarrow$ cheaper in terms of computing time).
- More sophisticated MC integration methods exist for very unevenly distributed integrands ("importance sampling"). Error always $\mathcal{O}(1/\sqrt{N})$.
- Comparison: the error e.g. for the trapezoid method is $\sim 1/N^2$, for Simpson's method $\sim 1/N^4$
- One should prefer deterministic methods if possible (e.g. for regular integrands, simple integration domains in $\lesssim 3$ dimensions).
- But in $d \gg 1$ dimensions, or for ill-behaved integrands, or complicated domains of integration, MC methods may well be the only feasible ones despite their slow convergence in low dimensions.

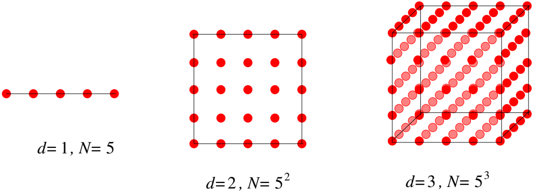
MC integration: $d \gg 1$ dimensions

The curse of dimensionality:

Numerical integration in d dimensions of a function $f : D \to \mathbb{R}$ defined on $D \subset \mathbb{R}^d$,

$$I = \int_D f(x)\, d^d x\,.$$

- Deterministic methods are extremely inefficient if d is large.
- This is because the number of nodes grows exponentially with d:



E.g. in $d = 20$ with only 5 nodes/dimension: $N = 5^{20} \approx 10^{14}$ nodes!

- In physics: dimension of phase space = number of degrees of freedom, easily $\gg 1$
- Solution: stochastic MC algorithms whose speed of convergence does not directly depend on d but which follow the $\frac{1}{\sqrt{N}}$ law.

MC integration

Exercises

Write a program which calculates the volume of the unit ball in d dimensions by MC integration. Compare with the exact values $\frac{4\pi}{3}$, $\frac{8\pi^2}{15}$, and $\frac{\pi^5}{120}$ for $d = 3,\ 5,\ 10$.

Sampling

Reminder:

- A probability distribution $P(x)$ on a measurable space M is a map $P: \; M \to [0, \infty[$ such that $\sum\!\!\!\!\!\!\int_M P(x) = 1$ (+ technical conditions).
- The probability for a random variable X following the distribution P to take a value in $E \subseteq M$ is
$$\Pr \left(X \in E \right) = \sum\!\!\!\!\!\!\!\int_E P(x)\,.$$
- The expectation value or mean $\langle f \rangle$ of a function $f(x)$ on M according to P is
$$\langle f \rangle = \sum\!\!\!\!\!\!\!\int_M f(x) P(x)\,.$$

Sampling

Examples for probability distributions in physics:

- The probability density function

  $$P(x) = |\psi(x)|^2$$

  where $\psi(x)$ is a wave function in quantum mechanics.

- The Boltzmann distribution

  $$P(x) = \frac{1}{Z} e^{-\frac{E(x)}{k_B T}}$$

  of statistical physics, which describes a canonical ensemble in thermal equilibrium. T = temperature, k_B = Boltzmann's constant, Z = normalization (partition function), $P(x)$ = probability that the microstate x with energy $E(x)$ is realized.

- $P(x|d)$ = Bayesian posterior probability that a theoretical model with certain parameter values x describes some experimental data d.

  $$P(x|d) \sim P(d|x)P(x)$$

  $P(d|x)$: goodness of fit, $P(x)$: "prior" (a priori probability) = theoretical bias.

Sampling

Goal: Obtain a sample of some probability distribution $P(x)$ on M
= a finite set of N random variables distributed according to $P(x)$

Motivations:

With a sample of size N for N sufficiently large, one can study the properties of P numerically. E.g.

- compute expectation values $\leftrightarrow$ physical observables
- compute correlations
- marginalization (integrate/sum over a subspace of x)

Sampling: Inverse transform sampling

Given a random number generator for the uniform distribution on $[0,1]$:

How can we use it to sample some other PDF $P(x)$?

Analytic method: Inverse transform sampling in one dimension.

Let y be a random variable which is uniformly distributed on $[0,1]$. Then $x = F^{-1}(y)$ is distributed according to P, where $F(x) = \int^x P(x')\,\mathrm{d}x'$ is the cumulative distribution function and F^{-1} is its inverse.

Proof:

$$\begin{aligned}\Pr\,(x \in [a,b]) = \Pr\,\left(F^{-1}(y) \in [a,b]\right) &= \Pr\,(y \in [F(a),F(b)]) = F(b) - F(a) \\ &= \int_a^b P(x')\,\mathrm{d}x'\,.\end{aligned}$$

In the next-to-last step we have used that y is uniformly distributed on $[0,1]$, and thus $\Pr\,(y \in [A,B]) = B - A$.

Conclusion: To find $x(y)$, we need to calculate the integral

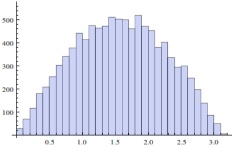
$$y = F(x(y)) = \int_{x(0)}^{x(y)} P(x')\,\mathrm{d}x'$$

and to solve for $x(y)$.

Sampling: Inverse transform sampling

Example: $P(x) = \frac{1}{2}\sin(x)$ on $[0, \pi)$.

$$y = \int_0^x \frac{1}{2}\sin(x')\,\mathrm{d}x' = \frac{1}{2}(1 - \cos(x)) \quad \Rightarrow \quad x = \arccos(1 - 2y)$$



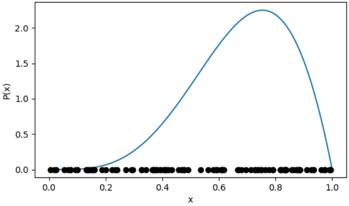
Second example: $P(t) = Re^{-Rt}$ on $[0, \infty)$, with $R > 0$ a constant.

$$y = R\int_0^t e^{-Rt'}\,\mathrm{d}t' = -\left[e^{-Rt'}\right]_{t=0}^{t'=t} = 1 - e^{-Rt} \quad \Rightarrow \quad t = -\frac{\log(1-y)}{R} = -\frac{\log u}{R}$$

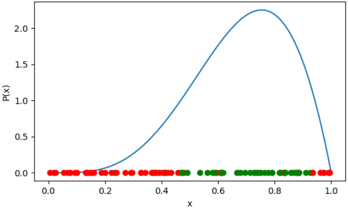
(where $u = 1 - y$ is also uniformly distributed on $[0, 1]$).

The inverse transform sampling method works only for those $P(x)$ whose cumulative distribution function is known and can be inverted: limited scope. Other analytic methods exist for some special $P(x)$ (e.g. Box-Muller transform for the normal distribution).

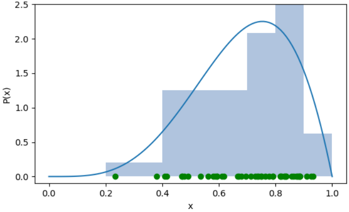
Monte-Carlo sampling: Rejection samplinσ



1. Generate random variables x_i ● ● ● uniformly distributed on the support of P.



2. For each x_i, accept with probability $\propto P(x_i)$, reject the others



3. The accepted points form a sample of P.

Monte-Carlo sampling: Rejection sampling

Draw random variable x from the uniform distribution on $[a, b]$ (which should include the support of P). Keep it with probability $P(x)/C$ (where $C \geq \max P$ is a constant), reject it otherwise.

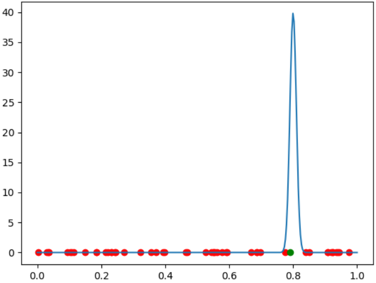
```python
import numpy as np

def rejection_sampling(P, C, a, b):
    while True:
        # generate x uniformly distributed on [a, b):
        x = (b - a) * np.random.random() + a
        # accept with probability P(x) / C; reject otherwise
        if P(x) / C > np.random.random():
            return x
```

To create a sample of `N` points:

```python
sample = [rejection_sampling(P, c, a, b) for _ in range(N)]
```

Monte-Carlo sampling: Rejection sampling

Rejection sampling is inefficient if $P(x)$ is narrowly peaked (need to generate many points, only to reject almost all of them).



This is often the case for multivariate distributions depending on many variables.

Possible improvement: Importance sampling, draw the x_i not from a uniform distribution but from a distribution with a similar shape as $P(x)$.

Monte-Carlo sampling: MCMC

Markov Chain Monte Carlo (MCMC)

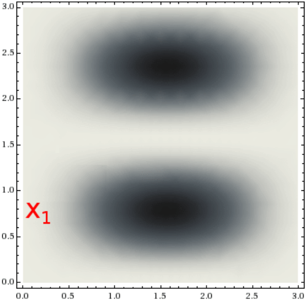
Idea: Once we have found the region of M where the distribution P is localized, stay close to this region and explore it by a random walk. Construct a Markov Chain = a sequence of random steps.

Metropolis-Hastings algorithm:

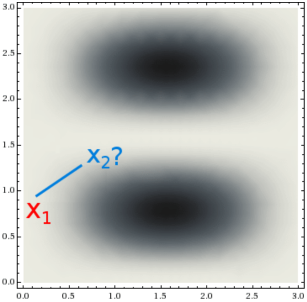
- Start from $x_1 \in M$ chosen randomly.
- Given x_n, choose a random point $y \in M$ close by.
  - If $P(y) \geq P(x_n)$, accept y as the next point in the chain.
  - If $P(y) < P(x_n)$, accept y with probability $P(y)/P(x_n)$.
- If y has been accepted, repeat with $x_{n+1} = y$. Otherwise repeat with $x_{n+1} = x_n$.

It can be shown: The $\{x_n\}_{1 \leq n \leq N}$ sample $P(x)$ if N is sufficiently large.

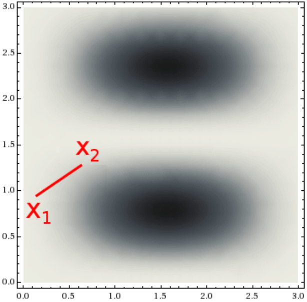
Monte-Carlo sampling: MCMC



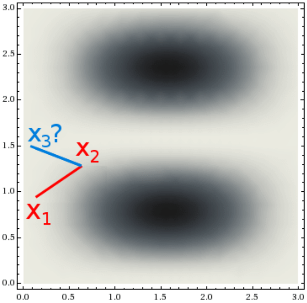
Monte-Carlo sampling: MCMC



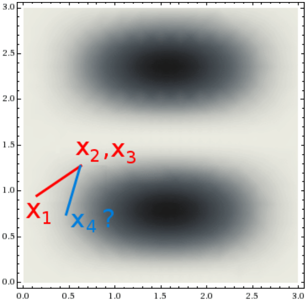
Monte-Carlo sampling: MCMC



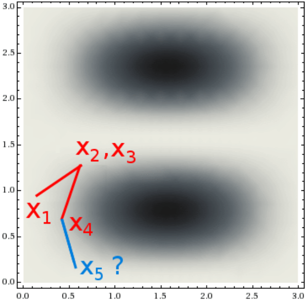
Monte-Carlo sampling: MCMC



Monte-Carlo sampling: MCMC



Monte-Carlo sampling: MCMC



Monte-Carlo sampling: MCMC

Remarks:

- The procedure only depends on ratios of probabilities
  $\Rightarrow$ usable even if the overall normalization of $P(x)$ is unknown (often the case).
- Detailed balance property: Let $P(x)$ be the probability to be in the state x and $W(x;x')$ the transition probability to pass from x to x', then

  $$P(x)W(x;x') = P(x')W(x';x)\,.$$

  This property guarantees (together with some more technical conditions) that the distribution of the $\{x_n\}$ converges towards $P(x)$.
- A randomly chosen starting point may be in a region of very small $P(x)$. The first few iterations, before finding a more interesting region where $P(x)$ is localized, will therefore not be representative of P as a whole, and the first part of the chain is commonly discarded. "Burn-in" of the Markov chain.
- If $P(x)$ exhibits multiple "islands" of large P separated by an "ocean" of low P, it may be difficult to transition between the islands. One may run several chains with random starting points to obtain a representative sample.

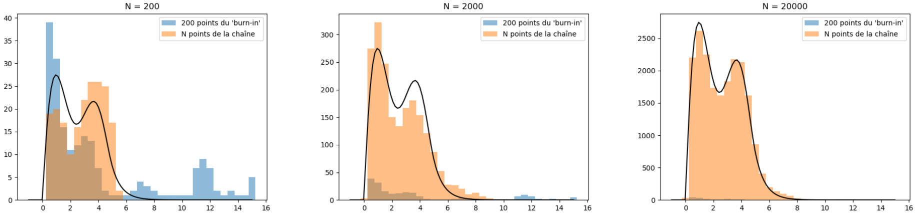
A simple one-dimensional example

```
import numpy as np

n_chain = 20000 # number of points in the chain
n_burnin = 200  # number of points to discard at first
d = 0.5         # step size

def P(x):  # an unnormalized PDF
   [...]   # (precise form is unimportant)

# Starting from x, propose x +/- d as a new point.
# Accept or reject according to Metropolis-Hastings.
# Return the new point (if accepted) or x again (if not).
def mcmcstep(x):
    xnew = x + d * np.random.choice([-1, 1])
    if P(xnew) > P(x):
        return xnew
    elif np.random.random() < P(xnew) / P(x):
        return xnew
    else:
        return x
```

A simple one-dimensional example

To run the algorithm:

```
x = 20 * np.random.random() # starting value (arbitrary)
chain = np.empty(n_chain)    # array for the Markov chain

for i in range(n_burnin): # run without memorizing results
    x = mcmcstep(x)        #    (burn-in)
for i in range(n_chain):  # then run building up the chain
    chain[i] = x
    x = mcmcstep(x)
```



Ising model

The Ising model is of central importance in statistical physics. It is defined by a rectangular lattice of N spins with nearest-neighbour interactions, and can be used to model e.g. a ferromagnet.

- Each spin is in one of the states $|+\rangle$ or $|-\rangle$: 2^N possible configurations.
- The energy of the system depends on the spins' orientation w.r.t. their nearest neighbours. The Hamiltonian is

  $$H = -J \sum_{\langle ij \rangle} S_i S_j$$

  where $S_i = \pm 1$ for spin i in the state $|\pm\rangle$, the sum includes all pairs $\langle ij \rangle$ of neighbouring spins in the lattice, and $J > 0$ is the exchange energy.
- The probability for a configuration ϕ to be realized is given by the Boltzmann distribution:

  $$P(\phi) = \frac{1}{Z} e^{-\frac{H(\phi)}{k_B T}}$$

  where T is the temperature, k_B is Boltzmann's constant, and the normalization Z is the partition function

  $$Z = \sum_{\phi} e^{-\frac{H(\phi)}{k_B T}} .$$

Ising model

- Magnetization in some spin configuration ϕ:

$$M = \mu \sum_i S_i \qquad (\mu = \text{magnetic moment per spin, constant})$$

- Mean magnetisation:

$$\langle M \rangle = \sum_\phi M(\phi) P(\phi) = \frac{1}{Z} \sum_\phi M(\phi) e^{-\frac{H(\phi)}{k_B T}}$$

- Magnetic susceptibility ↔ fluctuations of M:

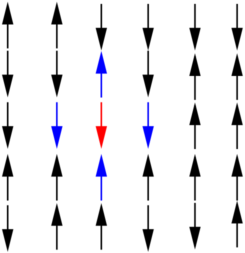
$$\chi = \frac{N}{k_B T} \left(\langle M^2 \rangle - \langle M \rangle^2 \right)$$

- Heat capacity ↔ fluctuations of the energy:

$$C_V = \frac{1}{N k_B T^2} \left(\langle H^2 \rangle - \langle H \rangle^2 \right)$$

We want to find $\langle M \rangle$, $\langle H \rangle$, χ, and C_V as functions of temperature. We will study the 2-dimensional case.

2D Ising model



2d spin lattice.
Each spin's energy depends on its orientation with respect to its nearest neighbours.

On a 2D $L \times L$ square lattice:

$$H = -J \sum_{i=1}^{L} \sum_{j=1}^{L} (S_{ij} S_{i+1,j} + S_{ij} S_{i,j+1})$$

Define $S_{L+1,j} \equiv S_{1,j}$ and $S_{i,L+1} \equiv S_{i,1}$ (periodic boundary conditions).

Problem: The number of possible configuration grows exponentially with the system's size. For $N = L \times L = 10 \times 10$ spins, one already has $2^N = 2^{100} \approx 10^{30}$ configurations $\Rightarrow$ impossible to directly calculate Z, $\langle M \rangle$, $\langle H \rangle$, $\langle M^2 \rangle$, or $\langle H^2 \rangle$ by direct summation.

Solution: Compute these observables approximately using a sample of the Boltzmann distribution obtained with the MCMC method.

MCMC for the 2d Ising model

Metropolis-Hastings algorithm applied to the Ising model:

- Start with either a random or ordered configuration (“warm/cold start”).
- Randomly choose a spin. Compute $\Delta E = E' - E$: energy difference if the spin were flipped.
- If $\Delta E < 0$, flip the spin. If $\Delta E > 0$, flip it with probability

$$\frac{\frac{1}{Z}e^{-\frac{E'}{k_B T}}}{\frac{1}{Z}e^{-\frac{E}{k_B T}}} = e^{-\frac{\Delta E}{k_B T}}$$

- Repeat until a configuration typical for thermal equilibrium is reached (“burn-in”).
- Then, repeat again as often as needed to obtain a sample of the desired size, memorizing the values of E, E^2, M and M^2 at each step.
- Finally, calculate the mean values $\langle E \rangle$, $\langle E^2 \rangle$, $\langle M \rangle$, and $\langle M^2 \rangle$ on the sample.

We will measure temperatures in units of $T_0 \equiv \frac{J}{k_B}$, energies in units of J, and magnetizations in units of μ. Also, we will use specific energies E/N rather than E.

MCMC for the 2d Ising model

To create an $L \times L$ spin lattice, all spins in the state 1, with $J = 1$ and $L = 16$:

```
import numpy as np

L = 16                 # L x L spins
norm = 1 / L**2        # normalization
lat = np.ones((L, L), dtype=int)  # the lattice
coupling = 1.0         # coupling constant J
```

Initialize some constants:

```
burnin_steps = 10000   # number of burn-in steps
mcmc_steps = 100000    # Markov chain size

min_T = 0.1            # minimal temperature
max_T = 3              # maximal temperature
T_steps = 30           # different temperatures to scan over
filename = "ising.dat" # file for saving the data
```

MCMC for the 2d Ising model

Some functions to flip a single spin. It will be flipped with certainty if $\Delta E < 0$. Otherwise, it will be flipped with probability $e^{-\Delta E / k_B T}$.

```
# Flip the spin (i, j)
def flip(i, j):
        lat[i, j] *= -1

# Flip the spin (i, j) if energetically favoured or
# by thermal fluctuation; return energy difference
def flip_maybe(T, i, j):
   DeltaE = deltaE(i, j)
   if DeltaE < 0:         # energy gained? then flip
      flip(i, j)
      return DeltaE
   elif np.random.random() < np.exp(-DeltaE/T): # en. lost?
      flip(i, j)   # possibly flip anyway (fluctuation)
      return DeltaE
   else:           # change nothing, energy remains same
      return 0.0
```

MCMC for the 2d Ising model

To compute ΔE upon flipping the spin S_{ij}:

$$\begin{aligned} E &= -J\,(S_{i-1,j} + S_{i+1,j} + S_{i,j-1} + S_{i,j+1})\,S_{ij} + (\text{terms independent of } S_{ij}) \\ E' &= -J\,(S_{i-1,j} + S_{i+1,j} + S_{i,j-1} + S_{i,j+1})\,(-S_{ij}) + (\text{terms independent of } S_{ij}) \\ \Delta E = E' - E &= 2\,J\,(S_{i-1,j} + S_{i+1,j} + S_{i,j-1} + S_{i,j+1})\,S_{ij} \end{aligned}$$

```python
# energy difference if spin (i, j) were flipped
def deltaE(i, j):
    prev_i, next_i = (i - 1) % L, (i + 1) % L
    prev_j, next_j = (j - 1) % L, (j + 1) % L
    DeltaE = 2 * coupling * lat[i, j] *      \
                (lat[prev_i, j] + lat[next_i, j]
                    + lat[i, prev_j] + lat[i, next_j])
    return DeltaE
```

Note the “% L” to account for the periodic boundary conditions.

MCMC for the 2d Ising model

Total energy and magnetization: Recall

$$H = -J \sum_{i=1}^{L} \sum_{j=1}^{L} \left(S_{ij} S_{i+1,j} + S_{ij} S_{i,j+1} \right), \quad M = \mu \sum_{i} S_i$$

```python
# total energy
def energy():
    E = 0.0
    for i in range(L):
        next_i = (i + 1) % L # =0 if i=L-1, periodic BCs
        for j in range(L):
            next_j = (j + 1) % L # periodic BCs
            E -= coupling * lat[i, j] * lat[next_i, j]
            E -= coupling * lat[i, j] * lat[i, next_j]
    return E

# total magnetization
def magnetisation():
    return np.sum(lat)
```

MCMC for the 2d Ising model

Some auxiliary functions:

```python
# T -> infinity: all spins random
def heat():
    for i in range(L):
        for j in range(L):
            lat[i, j] = np.random.choice([-1, 1])

# T -> 0: all spins aligned, ground state
def freeze():
    spin = np.random.choice([-1, 1])
    for i in range(L):
        for j in range(L):
            lat[i, j] = spin

# generate a random index
def random_index():
    return np.random.randint(L)
```

MCMC for the 2d Ising model

“Burn-in” routine for attaining thermal equilibrium

```python
def burnin(T, burnin_steps):
    for n in range(burnin_steps):
        i, j = random_index(), random_index()
        flip_maybe(T, i, j)
```

Main program:

```python
Epoints = np.zeros(T_steps)  # <E> as a function of T
Mpoints = np.zeros(T_steps)  # <M>
E2points = np.zeros(T_steps) # <E^2>
M2points = np.zeros(T_steps) # <M^2>

T = min_T
dT = (max_T - min_T) / T_steps
Tpoints = np.arange(min_T, max_T, dT) # temperatures
```

MCMC for the 2d Ising model

Main loop:

```
for step in range(T_steps):
  freeze()                 # start in the ground state
  burnin(T, burnin_steps)  # burn-in, memorize nothing yet

  en = energy()          # compute energy after burn-in
  magn = magnetisation() # also compute magnetization
  E = M = E2 = M2 = 0.0  # <E>, <M>, <E^2>, <M^2>

  for n in range(mcmc_steps):   # loop to build up the chain
    i, j = random_index(), random_index() # choose a spin
    oldspin = lat[i, j]         # memorize its state
    en += flip_maybe(T, i, j)   # flip it (or not), update E
    E += en                     # add E to <E>
    E2 += en**2                 # add E^2 to <E^2>
    magn += lat[i, j] - oldspin # update M
    M += magn                   # add M to <M>
    M2 += magn**2               # add M^2 to <M^2>
  # finished constructing chain, must now normalize results
```

MCMC for the 2d Ising model

Normalize the results and save the data in a file

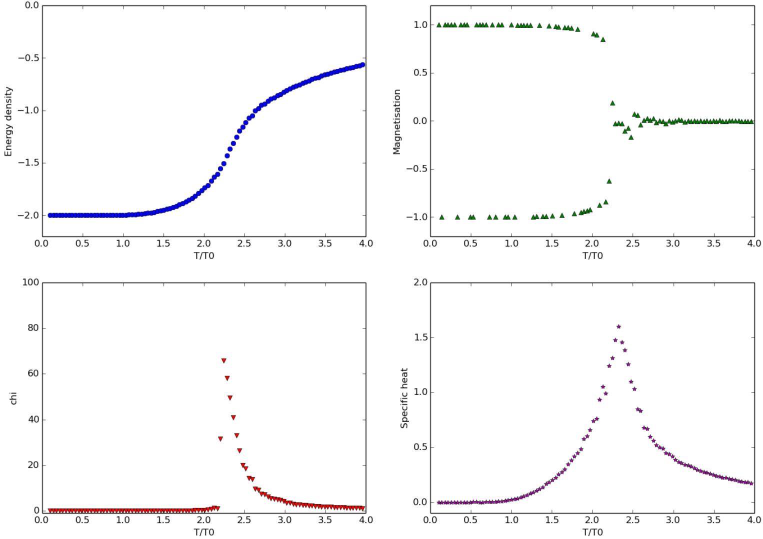
```
# Main loop continues
    E *= norm        # computing <E> per spin
    M *= norm
    M2 *= norm**2
    E2 *= norm**2
    E /= mcmc_steps  # <E> = sum(E) / (sample size)
    M /= mcmc_steps
    E2 /= mcmc_steps
    M2 /= mcmc_steps

    Epoints[n] = E    # add <E>(T) to the list
    E2points[n] = E2
    Mpoints[n] = M
    M2points[n] = M2

    T += dT # end of main loop

np.savetxt(filename, np.transpose([Epoints, E2points,
                              Mpoints, M2points, Tpoints]))
```

MCMC for the 2d Ising model

The separate program `plot_ising.py` will plot the data:

```
import numpy as np
import matplotlib.pyplot as plt

filename = "ising.dat"

data = np.transpose(np.loadtxt(filename))
Epoints = data[0]
E2points = data[1]
Mpoints = data[2]
M2points = data[3]
Tpoints = data[4]

# Compute the susceptibility and the heat capacity
chipoints =  256 * (M2points - Mpoints**2) / Tpoints
CVpoints = 256 * (E2points - Epoints**2) / Tpoints**2

plt.plot(Tpoints, CVpoints, ’m*’) # plot CV(T)
plt.xlabel(’T/T0’)
plt.ylabel(’Specific heat’)
plt.show()
```

Results for 100 temperatures, 3M MCMC iterations per temperature (∼2h on a standard laptop):



MCMC for the 2d Ising model, results

- At low temperatures: spontaneous magnetization
- At high temperatures: mean magnetization is zero
- Second-order phase transition at the Curie temperature $T_c = 2.27\,T_0$
- At $T > T_c$: Curie-Weiss law, $\chi \sim \frac{1}{T-T_c}$ (approximately)
- Simulation less accurate close to T_c

MCMC

Exercises

Create a MC simulation of an ideal quantum gas in a box. The gas consists of 1000 atoms, each characterized by three quantum numbers n_x, n_y, $n_z = 1, 2, 3 \ldots \infty$. The kinetic energy per atom is

$$E = \frac{\pi^2 \hbar^2}{2mL^2} \left(n_x^2 + n_y^2 + n_z^2 \right)$$

where L is the box size and m the mass. The atoms don't interact, hence their total energy is the sum of their kinetic energies.

- Show that, when passing from n_x to $n_x \pm 1$, the energy changes by

$$\Delta E = \frac{\pi^2 \hbar^2}{2mL^2} (\pm 2n_x + 1) \, .$$

- Use the Metropolis-Hastings method to simulate this system. The probability distribution is the Boltzmann distribution,

$$P(\phi) = \frac{1}{Z} e^{-E(\phi)/k_B T} \, .$$

  Plot the energy per particle for a sample of $N = 200\,000$ points at fixed T.
- Plot the mean energy as a function of T.

MC optimization: Simulated annealing

Simulated annealing is a numerical optimization method inspired by a physical process, namely the progressive cooling of a heated crystal lattice of a solid.

- If the cooling proceeds sufficiently slowly, the solid will relax to a state of minimal energy (perfect lattice).
- If the process is too rapid, it will end up in a local minimum with crystallographic defects.

In a more general context, the goal of the method is to find an approximate global minimum of some complicated function. If there are many local minima, deterministic optimization methods tend to be inefficient, while a MC method may be able to find a good approximation of the global minimum.

Simulated annealing

To find the minimum-energy state of a system by simulated annealing:

- Explore the configuration space with a random walk following the Metropolis-Hastings algorithm for the Boltzmann distribution.
- Progressively lower the temperature.
- As $T \to 0$, thermal fluctuations become less and less likely. Ultimately the system will settle to a state of low energy (not always the true global minimum, but often a good approximation).

In practical applications, the function to be minimized need not correspond to the energy of a physical system; it can be any function depending on the system's parameters.

Simulated annealing

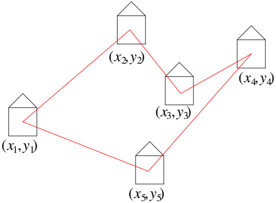
Example: The travelling salesman

A travelling salesman plans to visit N towns, whose coordinates in the plane are given. He would like to take the shortest possible path which passes by every town exactly once. In which order should he visit them?

Let (x_i, y_i) $(i = 1 \ldots N)$ be the towns' Cartesian coordinates. The salesman should end up at his starting point at the end of his trip. We therefore need to minimize the overall path length given by

$$d = \sum_i \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2}$$

where $(x_{N+1}, y_{N+1}) \simeq (x_1, y_1)$ and we are looking for the minimum on the set of all possible paths = all possible permutations of towns modulo overcounting.



$\frac{1}{2}(N-1)!$ inequivalent paths $\Rightarrow$ cannot test them all if N is large. Better deterministic algorithms to compute the exact solution still have exponential time complexity.

Simulated annealing

Simulated annealing for the travelling salesman problem:

- Start with an arbitrary path = an arbitrary order.
- At each iteration, propose a new path with two randomly chosen towns exchanged. Compute the "energy" = the length of the proposed new path.
- Accept or reject the new path according to the Metropolis-Hastings algorithm for a Boltzmann weight with "temperature" T.
- Progressively lower the "temperature" to gradually suppress "thermal" fluctuations.

Simulated annealing

Represent a path between N towns by a NumPy array p containing the $2N$ coordinates (x_i, y_i), in the order that the salesman visits them

```
import numpy as np

N = 20         # number of towns
Tstart = 10.0  # starting temperature
Tend = 1.E-2   # minimal temperature
delta = 1.E-4  # cooling rate

def length(p): # compute the length of a path p
    L = 0.0
    x = p[:, 0] # x coordinates
    y = p[:, 1] # y coordinates
    for n in range(-1, N-1): # (index -1 = last point)
        dx = x[n] - x[n + 1]
        dy = y[n] - y[n + 1]
        L += np.sqrt(dx**2 + dy**2)
    return L
```

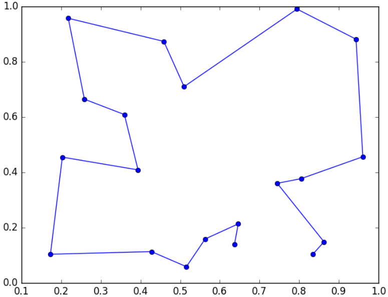
Simulated annealing

```python
path = np.random.random([N, 2])  # N random coordinate pairs
d = length(path)    # path length
T = Tstart          # initialize temperature

while T > Tend:     # main loop
    town1, town2 = np.random.randint(N), np.random.randint(N)
    path[[town1, town2],:] = path[[town2, town1],:]
    newd = length(path)  # length of proposed new path
    if np.exp(-(newd - d)/T) > np.random.random():
        d = newd
    else:  # undo the change
        path[[town1, town2],:] = path[[town2, town1],:]
    T *= (1 - delta) # exponential cooling
```

Here the cooling schedule is defined by $T \rightarrow (1-\delta)T$ at each step. Depending on the problem, other choices may be more efficient.

Simulated annealing

Example for $N = 20$ towns:



Simulated annealing

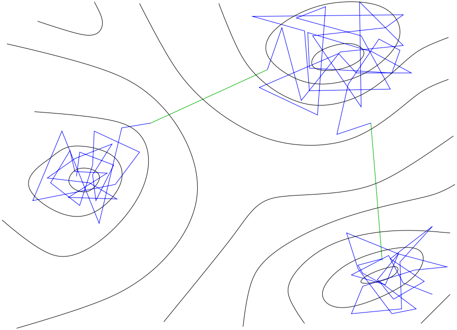
Exercises

Write a program which computes the maximal number of dimers that can be placed on a square $N \times N$ tiling. Each dimer will occupy two adjacent tiles. (The result is of course $N^2/2$ or $(N^2-1)/2$ depending on the parity of N, but let's pretend we didn't know that.) Use the simulated annealing method.

Kinetic Monte Carlo

MC methods can also be used to simulate the time evolution of a system out of equilibrium. For example, actual annealing of a real crystal lattice.

Consider a system of atoms or molecules with many local potential energy minima. At low temperatures the system will fluctuate around one of the local minima for most of the time. Occasionally a thermal fluctuation will be large enough to cross the energy barrier into a different state.



To simulate the long-term evolution of this system, we only need to know the transition rates between the different minima: r_{ij} = rate to jump between state i and state j.

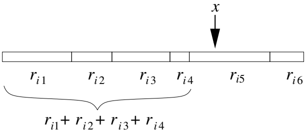
Kinetic Monte Carlo

To simulate the time evolution on large scales:

- No interest in simulating small fluctuations around any given state
- Transition rates r_{ij} must be inferred from experiment and/or small-scale simulations (e.g. molecular dynamics) and/or theory

Algorithm: Kinetic Monte Carlo

- At some given time the system is in state i
- Determine transition rates r_{ij} to other states j, compute total rate $R = \sum_j r_{ij}$
- Randomly select a transition to state k by weighting with rates r_{ij}:
  - Draw random number x uniformly distributed between 0 and R
  - Find k such that $\sum_{j=1}^{k-1} r_{ij} < x < \sum_{j=1}^{k} r_{ij}$



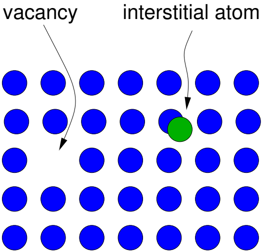
- Change state $i \leftarrow k$. Increase time by $\Delta t = -\log(u)/R$ where u is randomly drawn from $[0,1]$ (inverse transform sampling for the PDF $P(\tau) = \frac{1}{R}e^{-R\tau}$). Iterate.

Kinetic Monte Carlo

Example: Defect migration in a crystal

An impurity in an otherwise perfect crystal undergoes radiactive decay, which leads to two kinds of lattice defects:

- interstitial atoms at sites where there should be no extra atom in a perfect lattice,
- vacancies, i.e. empty lattice sites where the atom has been knocked out.



At finite temperature, three kinds of processes can change the system's configuration:

- migration of interstitials to an adjacent site,
- migration of vacancies to an adjacent site,
- recombination if an interstitial and a vacancy become neighbours → both disappear.

Kinetic Monte Carlo

Crude model for defect migration:

- 3D primitive cubic lattice: a defect can migrate to 6 adjacent sites, will recombine if a defect of the other type is on one of the 26 closest sites
- Migration rates:

  $$r = w \exp\left(-\frac{E}{k_B T}\right)$$

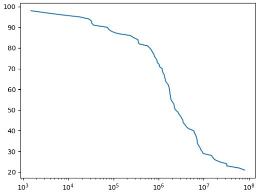
  where w is a prefactor, E is the activation energy and T is the temperature.
- We take $w_i = 1$, $E_i / k_B T = 10$ (interstitials), $w_v = 10^{-3}$ and $E_v / k_B T = 1$ (vacancies). (These are realistic orders of magnitude in units of fs^{-1} for a Si crystal at $T = 1000$ K.) Vacancies are more mobile.
- Recombination happens on much shorter time scales ⇒ instantaneous if an interstitial gets close to a vacancy.

Starting configuration:

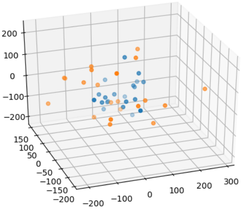
- $N = 100$ defects of either type
- Gaussian distribution with width $\sigma_i = 20$ and $\sigma_v = 10$ (interstitials are more widely scattered initially).

Kinetic Monte Carlo

Number of defect pairs as a function of time:



Defects after 1M simulation steps ($\sim$ few minutes computing time, $t \approx 10^8$):



Vacancies have mostly annihilated or moved outward from interstitials due to their greater mobility $\Rightarrow$ annihilation proceeds much slower at late times.

Kinetic MC

Exercises

Write a program to simulate this system.

Hints:

- To sample from the Gaussian normal distribution with mean `x0` and standard deviation `sigma`, use `numpy.random.normal(x0, sigma)`.
- Represent the position of a defect in the lattice by a triple of integers (x, y, z). In each simulation step, one of the defects changes one of its coordinates by ± 1. An interstitial recombines with a vacancy if $|x_i - x_v| \leq 1$ and $|y_i - y_v| \leq 1$ and $|z_i - z_v| \leq 1$.
- To create a linear-log plot, such as the one on the previous slide, use `matplotlib.pyplot.semilogx(x, y)`.