Heidelberg University

Department of Physics and Astronomy

Book in Progress

An Introduction to Computational Physics

Leonard Storcks

January 12, 2024

Abstract

TODO: write abstract

Contents

1 Notes and Todo

TODO: create TOC for CSDA, finish summaries in word, determine general structure of book

- Simulation Methods: From Physical Model to Simulation
- Computational Statistics and Data Analysis: Gaining Insights from Data
- Scientific Machine Learning: Incorporating Physical Knowledge into Machine Learning and Machine Learning into Simulations

- Give a birds eye view of the field of computational physics

make sure to understand everything correctly (against Bullards-Dynamo-Situation)

2 Introduction

Add / refer to example applications

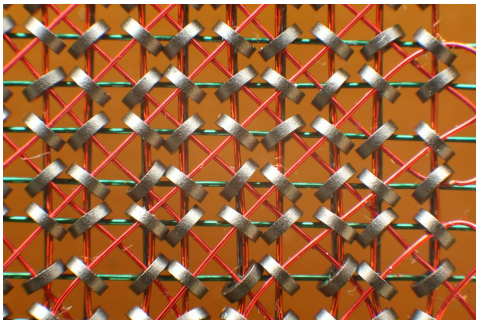
Computational physics encompasses

- simulation as a new paradigm to approach physical problems and validate theories by comparing simulated results to experiments
- statistics and data analysis to make sense of experimental or simulated data
- scientific machine learning to incorporate physical knowledge into Machine Learning and Machine Learning into simulations

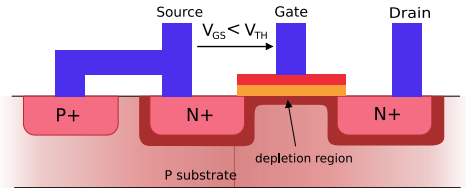
This books aim is to give an introduction to Computational Physics with the presented concepts often also being of great use in other fields (e.g. solving partial differential equations computationally in not only greatly important for physics but for instance also for solving the Black-Scholes equation in finance).

Content included mainly encompasses the lectures

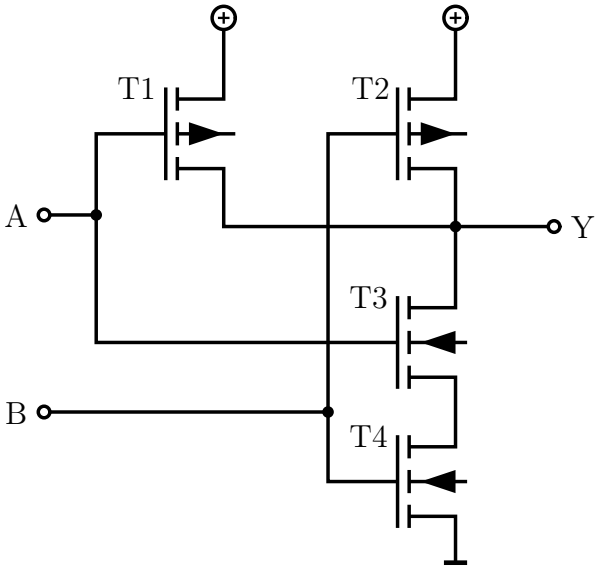
- Fundamentals of Simulation Methods (Ralf Klessen and Philip Girichidis, 2023)
- Computational Statistics and Data Analysis (Daniel Durstewitz, 2023)



(a) Historic Magnetic Core Storage. Bits are stored as the direction of magnetic flux.



(b) Field-effect transistor (more specifically MOSFET schematic). Power-efficiently switching currents is at the heart of modern computing. Based on a Floating-Gate or Charge-Trapping mechanism and the tunnel effect, storage can be realized via stored charges.



(c) NAND-Gate. Bit operations lie at the core of computations.

Figure 1: Basics of Computation.

Part I

Basics of Numerical Computation

How we write what algorithms is informed by how we represent data. While there exist exotic approaches like analog computers that can e.g. handle integration elegantly (Ulmann, 2020) or quantum computing which might e.g. at some point accelerate machine learning (Biamonte et al., 2017), standard binary data representation is vastly prevailing.

Binary data can be very efficiently and reliably stored and operated on (see figure 1), but the respective chosen representations might come with caveats.

3 Digital Representation of Numbers

In C, `int x = 100 * 200 * 300 * 400;` will surprisingly yield -1894967296 (for a 32-bit integer representation) (*overflow*), `float f1 = (2.3 + 1e20) - 1e20;` yields 0.0 (*round-off-error*) but `float f2 = 2.3 + (1e20 - 1e20);` correctly gives 2.3.

We want to understand and mitigate such caveats of arithmetic on computers, where we want quick calculations while also using as little storage as possible - simulations can quickly become big in storage (e.g. lots of particles).

Further details can be found in Higham, 2002 and Bryant and O'Hallaron, 2011.

3.1 Integer Arithmetic

In C, integers ($\in \mathbb{Z}$) are stored as fixed-size bit-sequences. The respective size dictates a range. C provides both unsigned and signed integer types, with negative numbers in the signed case represented using *two's-complements*.

3.1.1 Unsigned integers

For a bit-vector $\underline{x} = [x_{\omega-1}, x_{\omega-2}, \ldots, x_0]$ the unsigned conversion is

$$B2U_\omega := \sum_{i=0}^{\omega-1} x_i 2^i \tag{1}$$

(illustrated in figure 2) and the range is $0, \ldots, 2^\omega - 1$. In case of overflow in operations, the overflow is truncated in the most significant bits (see figure 3). As $B2U_\omega[x_{\omega-1}, x_{\omega-2}, \ldots, x_0] \mod 2^k = B2U_\omega[x_{k-1}, x_{k-2}, \ldots, x_0]$ we effectively store arithmetic result $\mod 2^\omega$.

$x_{\omega-1}$ is called most significant bit (MSB) and x_0 least significant bit (LSB).

Problem: When the result of an arithmetic operation exceeds the range of an integer type, unexpected results occur (overflow) (and also underflow in the signed case)

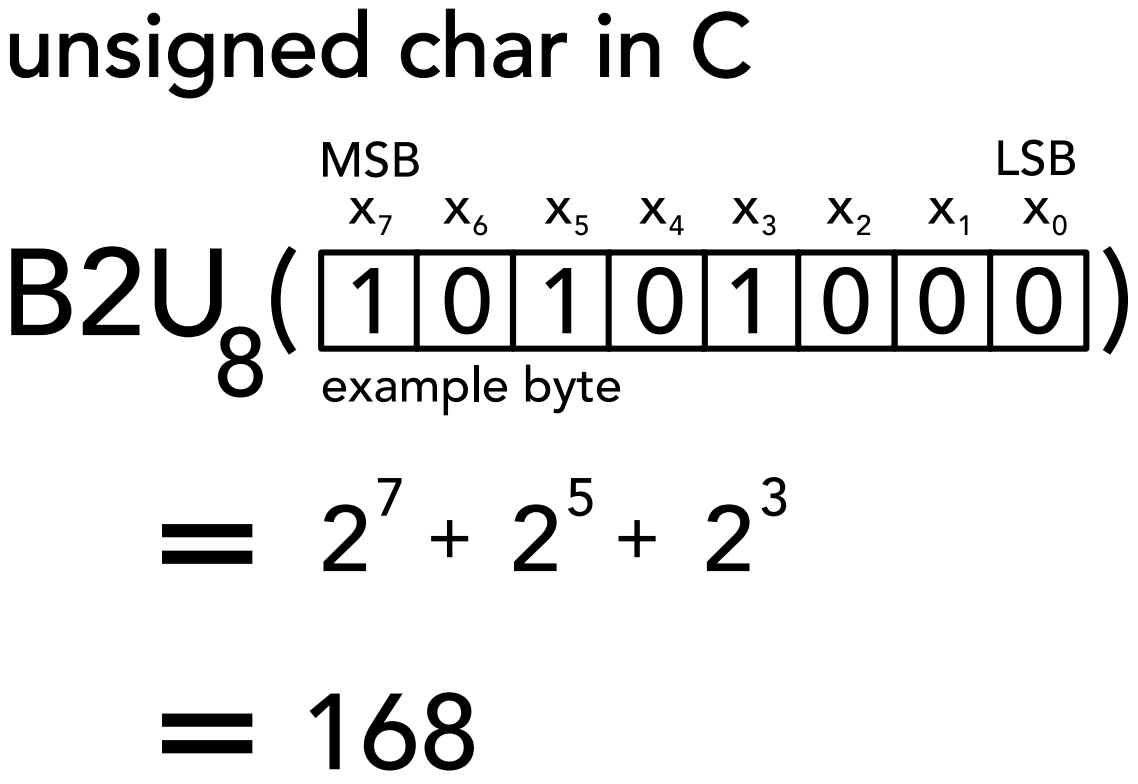


Figure 2: Example of an unsigned char in C.

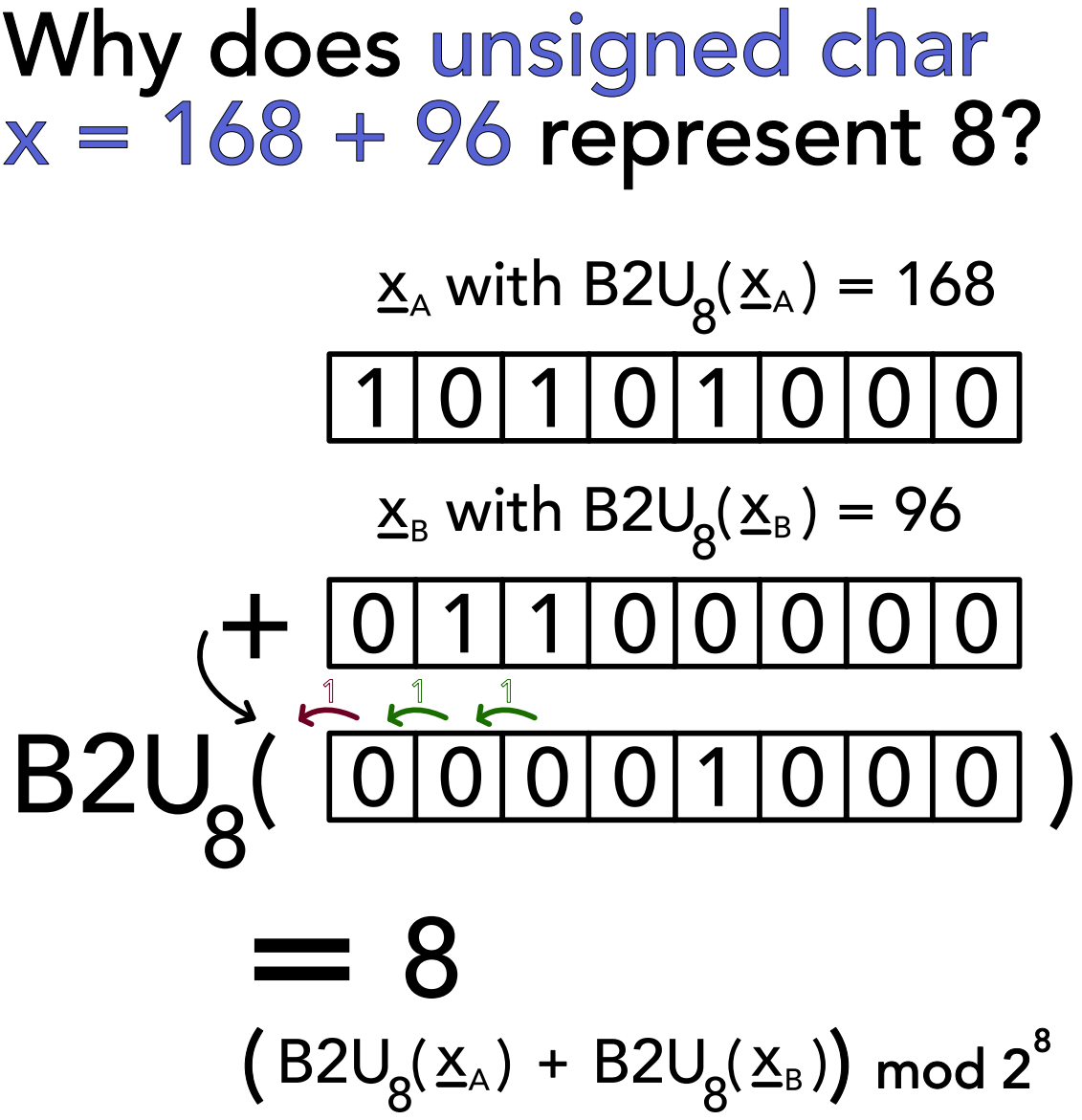


Figure 3: Example of unsigned char overflow.

3.1.2 Two's complement for negative numbers

Note that addition of integers by bitwise addition with carry-on is very fast.

Why can't we just let the MSB encode the sign?: A representation of the form

$$B2S_\omega(\underline{x}) := (-1)^{x_{\omega-1}} \cdot \sum_{i=0}^{\omega-2} x_i 2^i \tag{2}$$

(sign magnitude) has the disadvantage, that normal bitwise addition with carry-on does not work. Also, zero is encoded twice, as ± 0.

Idea of the two's-complement: The MSB flags the sign in $\underline{x} = [x_{\omega-1}, x_{\omega-2}, \ldots, x_0]$ by having a weighting factor $-2^{\omega-1}$

$$B2T_\omega(\underline{x}) := -x_{\omega-1} 2^{\omega-1} + \sum_{i=0}^{\omega-2} x_i 2^i \tag{3}$$

Carry-on to the ω-th bit in addition is again ignored. As we really just have to add positive numbers in bits x_0 to $x_{\omega-2}$ with correct sign-switch by carry-on, no special handling is necessary (see figure 4). The range is $-2^{\omega-1} \ldots 2^{\omega-1} - 1$.

From an unsigned int to the two's complement and vice versa we can get by

1. invert all bits
2. add +1 to the result[1]

If we want $\underline{x}$ with $B2T_\omega(\underline{x}) = -u$ then the bits following the MSB must encode k with $-u = -2^{\omega-1} + k$, so the encoded unsigned number must be $B2T_\omega(\underline{x}) = \underbrace{2^{\omega-1}}_{\text{sign bit}} + k = 2^\omega - u$ - a *two's complement.*

[1] These rules intuitively follow from the constraint, that bitwise addition with carry-on of $-u$ and u should result in an all-zero bitvector. Adding a bitvector to its inverted self results in an all-1 bitvector, adding one more then results in all zeros, as the last carry-on is discarded.

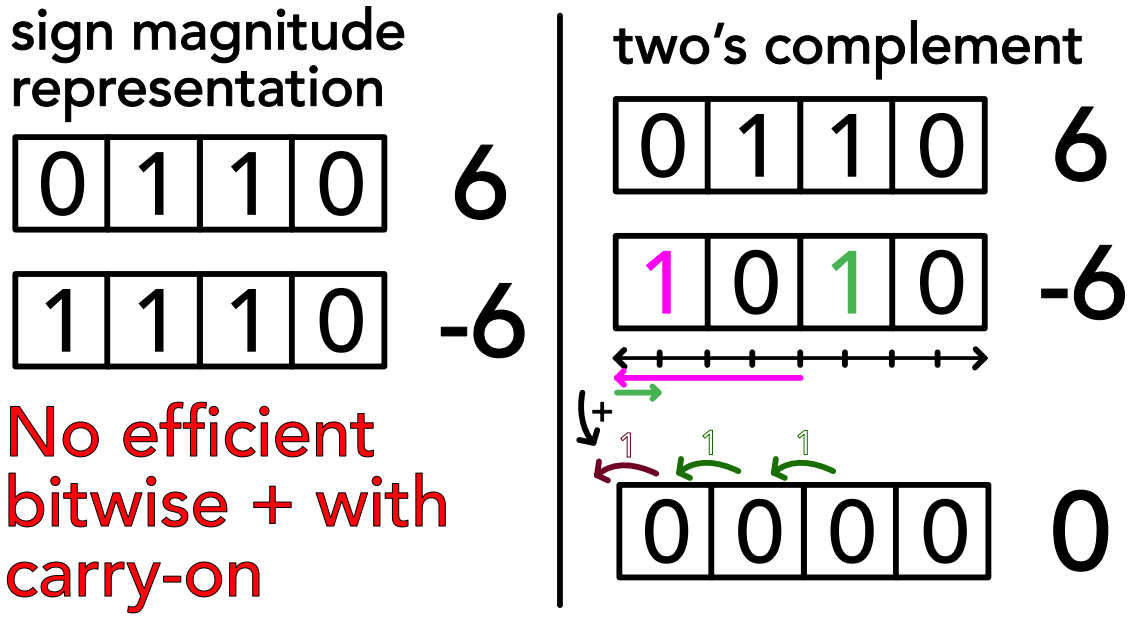


Figure 4: Illustration of the defining feature of the two's complement - addition is simple.

3.1.3 Integer types in C

Some common integer types with respective minimum sizes are given in table 1.

Type	Minimum Size ω
`char`	8 bits
`short`	16 bits
`int`	16 bits
`long`	32 bits
`long long`	64 bits

Table 1: Common integer types in C, signed ranges are $-2^{\omega-1} \dots 2^{\omega-1} - 1$, unsigned ranges are $0 \dots 2^{\omega} - 1$.

Problem: Using unsigned can come with unexpected results, when being cast from a negative number. E.g. `unsigned int x = -1;` will result in $2^{32} - 1$ in a 32-bit system (the two's complement is read as if an unsigned representation). More deviously, $-1 < 0U$ will evaluate to false, as all integers in a comparison are cast to unsigned, when one of them is unsigned.

3.1.4 Byte Ordering in Storage: Big and Little Endian

Bytes of e.g. a multi-byte integer can be stored from most significant byte to least significant byte (big endian) or vice versa (little endian), see figure 5.

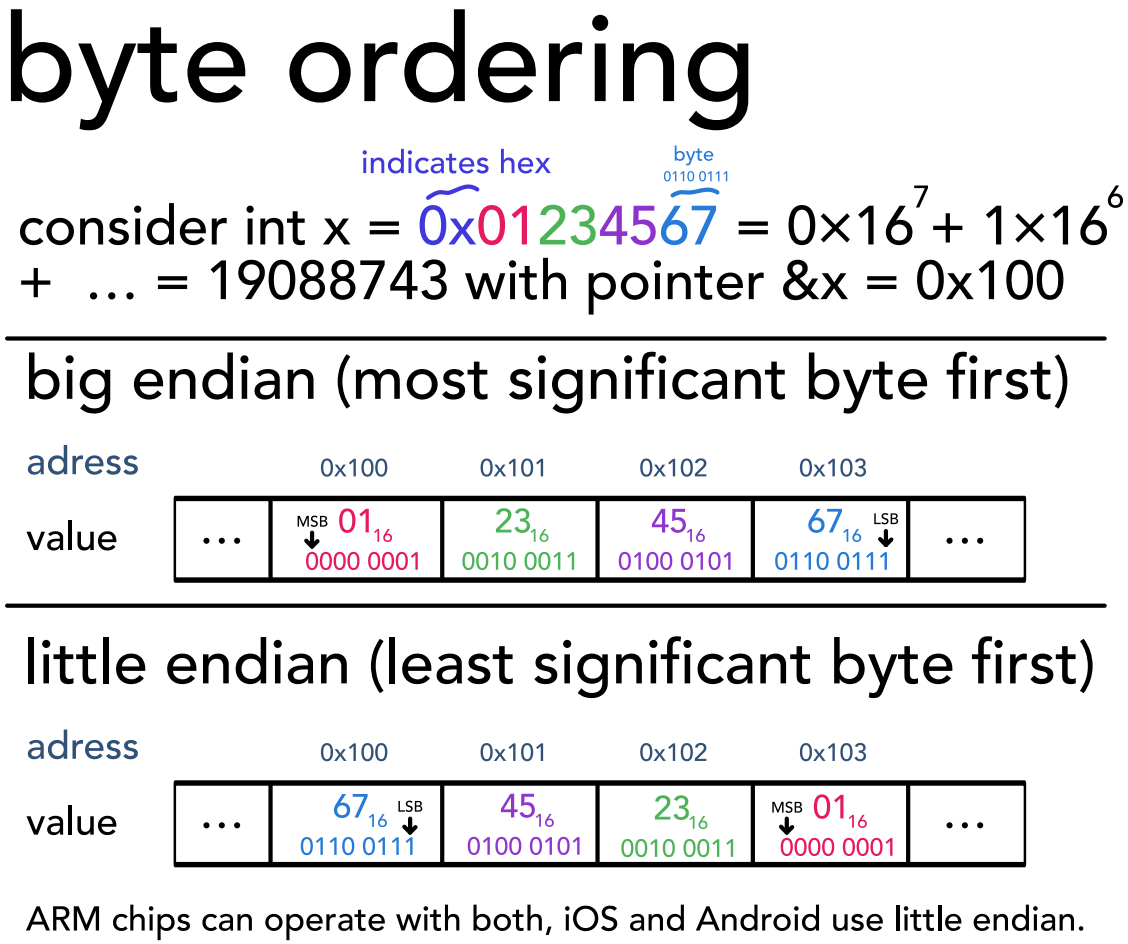


Figure 5: Big and Little Endian.

3.1.5 Properties and Caveats of Integer Arithmetic

Typical problems in integer arithmetic are overflow, integer division, the modulo operation and implicit type conversion.

Overflow: The respective integer ranges have finite ranges, overflow in arithmetic operations is cut-off. We must choose a type with sufficient range - mind that choosing types with too large of a footprint (e.g. always long) wastes storage and compute.

Example I: In `char c = 100 * 4; // range -128 to 127` the result is -112 as 400 in bits is 000110010000 where a cut-off to one byte means 10010000 so $-2^7+2^4 = -128+16 = -112$.

Example II: For `int a = -1 * pow(2,31); int b = 10;` we have $a < b$ but not $a - b < 0$ (for char the behavior is a bit different as up to int there is implicit type conversion.)

Integer division: All decimal places are truncated, so

$$5/3 = 1; -5/3 = -1; 5/-3 = -1$$

Modulo operation: The modulus in C is defined as $n\%m = n - (n/m)m$ so

$$5\%3 = 2; -5\%3 = -2; 5\%-3 = -2$$

Implicit type conversion: In C, the type can implicitly be converted up to signed int, which can avoid overflow, as illustrated in table 2.

implicit type conversion up to int can be helpful

```
char a,b;
a = 100;
b = 4;
int c = a * b; //
  ↪ 400
```

mind its only up to int

```
int a = 2e9;
int b = 3;
long long c = a *
  ↪ b;
printf("%llu\n",
  ↪ c); //
  ↪ 1705032704
```

As $2^{32} - 1 \approx 4 \cdot 10^9 < 6 \cdot 10^9 < 2^{33} - 1$ (unsigned ranges) we overflow to the 33rd-bit, which is cut-off, giving the signed result of $6 \cdot 10^9 - 2^{32} = 1705032704$.

explicit type consversion

```
int a = 2e9;
int b = 3;
long long c =
  ↪ ((long long) a)
  ↪ * ((long long)
  ↪ b);
printf("%llu\n",
  ↪ c); //
  ↪ 6000000000
```

Table 2: Type conversion and its caveats

3.1.6 Can there be integer-overflow in python?

Note that in python3, integers are implemented as "long" integer objects of arbitrary size, overflows are impossible (at the cost of speed; mind that e.g. numpy is based on C code).

3.2 Floating Point Arithmetic

In the following, we will encode rational numbers in the form $V = x \cdot 2^y$, with $x, y \in \mathbb{Z}$. Similar to the decimal notation

$$d_m d_{m-1} \dots d_0 . d_{-1} d_{-2} \dots d_{-n} = \sum_{i=-n}^{m} d_i 10^i \tag{4}$$

we can write in binary notation

$$b_m b_{m-1} \dots b_0 . b_{-1} b_{-2} \dots b_{-n} = \sum_{i=-n}^{m} b_i 2^i, \quad 0.01_2 = 0.25_{10} \tag{5}$$

or generally in base β in scientific notation

$$\begin{gathered}(-1)^s \cdot \underbrace{b_0 . b_{-1} b_{-2} \dots b_{-n}}_{\text{mantissa M}} \cdot \beta^e = \beta^e \cdot \sum_{i=-n}^{0} b_i 2^i, \quad \text{exponent } e, \\ \text{precision (number of digits) } p = n + 1, \quad \text{sign-bit } s \in \{0, 1\}\end{gathered} \tag{6}$$

Note that in this notation (in the form $V = x \cdot 2^y$) we cannot exactly represent e.g. 0.1 or 0.2.

$$0.1_{10} = 1.10011[0011] \dots_2 \cdot 2^{-4} \tag{7}$$

Disastrous historic example: In the first Gulf War (more specifically on 25th February 1991), a Patriot missile defense battery failed to intercept an incoming Iraqi Scud missile, because it used an internal clock counting up in tenths of a second represented by a 23-bit sequence. Future missile positions are predicted by extrapolating from past position with constant velocity. The Patriot system mixed both this inaccurate internal clock and a more accurate one, leading to the failed interception and 28 deaths among soldiers.

3.2.1 IEEE 754 Floating Point Standard

Based on the representation in scientific notation, we can store a floating point number in a bit-vector with

- a sign bit s
- an exponent e stored as an unsigned integer $E = e + b$ with bias b
- a mantissa M

where for single precision (32-bit)

- the exponent is stored in 8 bits, $b = 127$, $e_min = -126$, $e_max = 127$ with $E = 255$ and $E = 0$ reserved for special cases
- the mantissa is stored in 23 bits, with the first bit being implicitly 1 (**normalization**), which can always be assumed by appropriately choosing the exponent (floating point representations are not unique), so we have $p = 23(+1)$ bits of precision encoding an integer M but need a special representation for 0

where based on the exponent we differentiate between

- **normalized values for** $1 \leq E \leq 254$ with value of the floating point number

$$V = (-1)^s \cdot \left(1 + \frac{M}{2^p}\right) \cdot 2^{E-b}, \quad \text{sign bit } s$$
$$\text{biased exponent } E = e + b, \quad \text{integer representation of mantissa } M \tag{8}$$

- **denormalized values for** $E = 0$ with value of the floating point number

$$V = (-1)^s \cdot \frac{M}{2^p} \cdot 2^{-b+1}, \quad \text{for } M = 0, V = \pm 0 \text{ depending on } s \tag{9}$$

  The denormalized numbers start just below the normalized ones (by the factor 2^{-b+1} with $+1$ as we do not have the implicit leading 1 here) and now no normalization (implicit starting 1-bit) is assumed. This allows to approach zero with gradually decreasing precision (and even spacing) (smaller numbers occupy fewer digits as of the leading zeros) and ensures that for $x \neq y$, $x - y$ is non-zero, so $\frac{1}{x-y}$ is safe for $x \neq y$.

- $\pm\infty$ **for** $E = 255$ **and** $M = 0$
- **NaN (Not a Number) for** $E = 255$ **and** $M \neq 0$

Why is the exponent stored in a biased way, not two's complement?: In a two's complement representation of the exponent to compare two numbers, we have to compare the exponents and mantissas separately and the exponent-comparison is a bit more complicated than comparing biased representations. In the biased exponent representation we can just compare the bitvectors of exponent followed by mantissa interpreted as integers (also mind the sign).

The cases are illustrated in figure 6, with a specific example in figure 7.

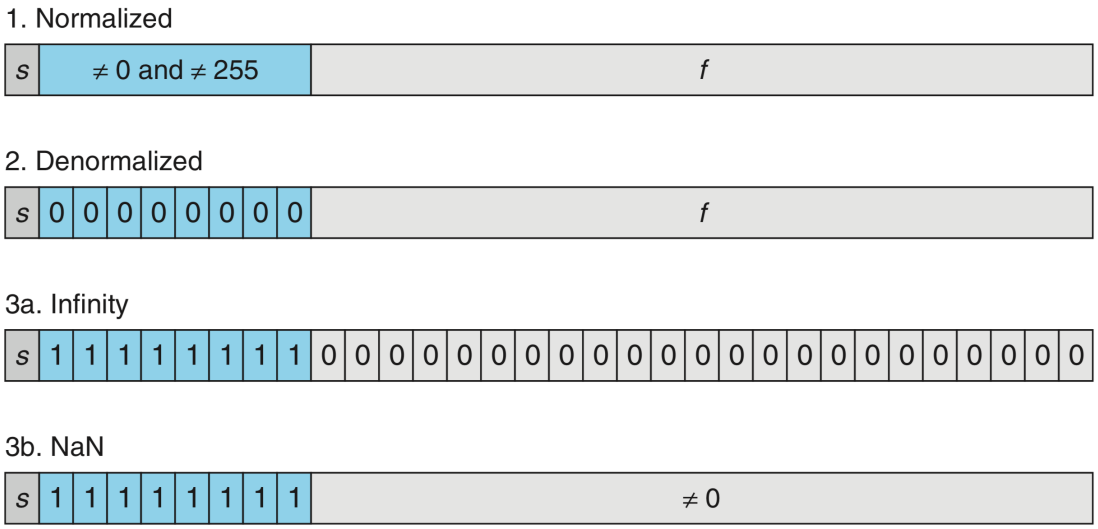


Figure 6: Cases of the IEEE 754 Floating Point Standard.

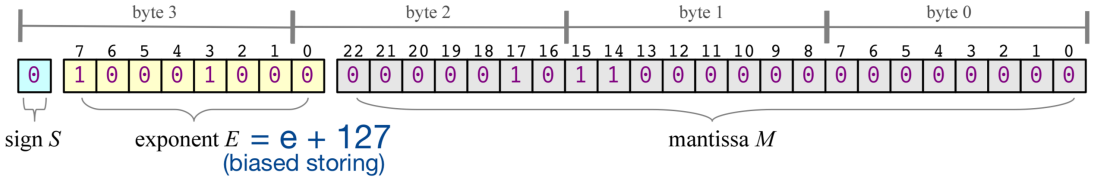


Figure 7: 523 can be written as $1.000001011_2 \cdot 2^9$ so $e = 9, E = e + 127 = 136$. As of the normalization, the leading 1 in the mantissa is implicit. The number is given as a single precision float in big endian.

3.2.2 Only a finite set of floating point numbers can be represented exactly

With 32 bits, 2^{32} states can be encoded (and mind that here e.g. NaN has multiple representations). In any case, the number of exactly representable numbers is finite, above 0 starting at

$$V_{\text{denorm,min}} = \frac{1}{2^p} \cdot 2^{-b+1} \underbrace{\approx}_{\substack{\text{single} \\ \text{p.}}} 1.4 \cdot 10^{-45} \tag{10}$$

with the smallest normalized number being

$$V_{\text{norm,min}} = (1+0) \cdot 2^{-b} \underbrace{\approx}_{\substack{\text{single} \\ \text{p.}}} 1.2 \cdot 10^{-38} \tag{11}$$

and the largest normalized number being

$$V_{\text{norm,max}} = (1 + \frac{2^p - 1}{2^p}) \cdot 2^{E_{\max} - b} \underbrace{\approx}_{\substack{\text{single} \\ \text{p.}}} 3.4 \cdot 10^{38} \tag{12}$$

3.2.3 Machine Precision is finite

The smallest increment in the mantissa, in $1 + \frac{M}{2^p}$, is the **machine precision**

$$\epsilon_{\text{mach}} = \frac{1}{2^p} \underbrace{\approx}_{\substack{\text{single} \\ \text{p.}}} 1.2 \cdot 10^{-7} \tag{13}$$

Consider two floating point numbers $V_1, V_2 > 0$ *next to each other*

$$V_1 = \left(1 + \frac{M}{2^p}\right) \cdot 2^{E-b}, \quad V_2 = \left(1 + \frac{M+1}{2^p}\right) \cdot 2^{E-b} \tag{14}$$

so their relative difference is bound by

$$\frac{V_2 - V_1}{V_2} = \frac{\frac{1}{2^p} \cdot 2^{E-b}}{\left(1 + \frac{M+1}{2^p}\right) \cdot 2^{E-b}} \leq \epsilon_{\text{mach}} \tag{15}$$

3.2.4 Rounding and Pitfalls of Floating Point Arithmetic

In the IEEE standard, results of addition, subtraction, multiplication and division must equal to one where the arithmetic operations are assumed to be exact and then there is rounding to the nearest representable number (computation is done at higher (typically double or higher) precision). Therefore (mind the section before)

$$\text{ralative error } \frac{|x - \hat{x}}{|x|} \leq \epsilon_{\text{mach}}, \quad \text{number } x, \text{ number on machine } \hat{x} \tag{16}$$

with the common pitfalls

- **Limitation of machine precision:** $a + b = a$ typically for $|b| < \epsilon_{\text{mach}}|a|$, i.e. when b cannot be resolved by the mantissa of a
- **Associativity is not guarantueed:** $(a+b)+c \underbrace{\neq}_{\text{i.A.}} a+(b+c)$, e.g. $(2.3+1e20)-1e20$ yields 0.0 but $2.3 + (1e20 - 1e20)$ yields 2.3
- **Problems of representability:** As e.g. 0.1 is not exactly representable in base $\beta = 2$,

$x/10 \neq 0.1 \cdot x$ in general while $x/2.0 = 0.5 \cdot x$ is exact. The compiler may automatically choose the multiplication variant as multiplication is faster than division.

- **Cancellation:** For $x = a - b$ subtractive cancellation causes relative errors already present in $\hat{a}$ and $\hat{b}$ to be (relatively) amplified, when a and b are of similar size. Significant digits are lost and the relative error explodes. Consider e.g. $a = 1.75682, b = 1.75471$ with $\hat{a} = 1.76$ and $\hat{b} = 1.75$. While $\hat{a}, \hat{b}$ have small relative errors (3 digits precision[2] Note that here 0.123 ans 0.127 agree in two significant digits, where one intuitively might say this should be rather 1.), $a - b = 0.00211$ and $\hat{a} - \hat{b} = 0.01$ has a large relative error (no precise digit).
- **NaN and Inf:** All calculations including NaN yield NaN, calculations with inf mostly inf (except of course $1/\text{inf} = 0$, ...)

where we should

- **Rewrite calculations so that errors do not amplify:** For $x = 10^8, y = 10^5, z = -1 - 10^5$ we have $xy + xz = -1.0066 \cdot 10^8$ (problem in resolving the -1 in xz) but $x(y + z) = -1.0 \cdot 10^8$.
- **Compare floats based on a maximum relative error:** Instead of $x == y$ we should use e.g. $|x - y| \leq \epsilon_{\text{mach}} \cdot \max(|x|, |y|)$.
- **As avoid overflow to inf and nan:** E.g. in `float x = 1e20; float y = x * x; float z = y / x` z will be inf.

3.2.5 Rewriting Expressions to Avoid Cancellation I

Consider $f(x) = \frac{1-\cos x}{x^2}$. For $x_e = 10^{-4}$, we have (when we represent 6 figures)

$$c := \cos x_e, \quad \hat{c} = 0.999999, \quad 1 - \hat{c} = 10^{-6}, \quad \text{while } 1 - c \approx 5 \cdot 10^{-9} \tag{18}$$

$1 - c$ has only one significant digit, the relative error is enormously amplified and we get

$$\frac{1 - \hat{c}}{x_e^2} = 100, \quad \text{while in reality for } x \neq 0 : 0 \leq f(x) \leq \frac{1}{2} \tag{19}$$

[2] $\hat{x}$ is said to approximate x to the r-th digit, if the absolute error is at most $\frac{1}{2}$ in the r-th digit, so

$$\text{largest integer s so that } 10^s < |x|, \quad |x - \hat{x}| < 1/2 \cdot 10^{s-r+1} \tag{17}$$

To avoid cancellation we can rewrite using $\cos x = \cos^2 \frac{x}{2} - \sin^2 \frac{x}{2} = 1 - 2\sin^2 \frac{x}{2}$ to

$$f(x) = \frac{1}{2}\left(\frac{\sin \frac{x}{2}}{\frac{x}{2}}\right)^2 \tag{20}$$

A comparison of both versions in python can be found in figure 8, at some point (roughly below 10^{-8}), $\cos x$ is too close to 1 to be resolved by the mantissa, so the total result is 0, above that the magnified error is visible.

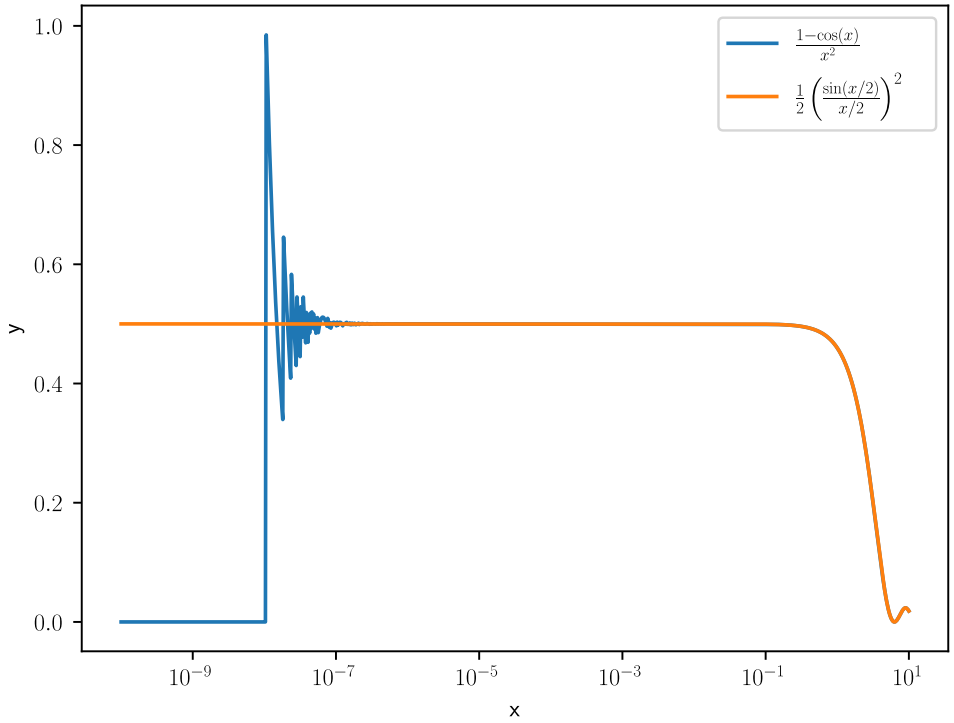


Figure 8: Comparison of the mathematically equivalent expressions $f(x) = \frac{1-\cos x}{x^2}$ and $f(x) = \frac{1}{2}\left(\frac{\sin \frac{x}{2}}{\frac{x}{2}}\right)^2$ in python.

3.2.6 Rewriting Expressions to Avoid Cancellation II

Consider the following expressions for the sample variance of $\{x_i\}_{i=1}^N$

$$\begin{aligned} \text{two-pass formula: } \bar{x} &= \frac{1}{N}\sum_{i=1}^{N-1} x_i, \quad \sigma_N^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 \\ \text{one-pass formula: } \sigma_N^2 &= \frac{1}{N-1}\left(\sum_{i=1}^{N} x_i^2 - \frac{1}{N}\left(\sum_{i=1}^{N} x_i\right)^2\right) \\ &\text{as } \sigma^2 = E[(x-\bar{x})^2] = E[x^2] - E[x]^2 \end{aligned} \tag{21}$$

While for the one-pass formula, we can calculate all necessary sums in one pass through the data, it suffers heavily from cancellation: For $\{10000, 10001, 10002\}$, the two-pass formula in single precision correctly gives 1.0 while the one-pass formula yields 0.0 (cancellation) (there are better one-pass formulas though).

3.2.7 Accumulation of Round-off Errors

Consider the sum

$$\sum_{k=1}^{\infty} k^{-2} = \frac{\pi^2}{6} \tag{22}$$

which we want to approximate by finitely many summands. If we sum the terms just as the formula suggests from large to small, at some point, the small changes will not be resolved anymore - so better sum up from small to large. Summing up in single precision for $N = 10^7$ terms, one gets

$$\begin{aligned} \text{big to small: } 1.644725323, \quad \text{small to big: } 1.644933939, \\ \text{exact (till 9th digit): } 1.644934058 \end{aligned} \tag{23}$$

As expected the big-to-small summation is too small.

3.2.8 Higher Precision

The above pitfalls are less severe in higher precision. For instance in double precision (64-bit)

$$\begin{aligned} p = 52(+1) \text{ mantissa bits}, \quad 11 \text{ exponent bits, with } e_{min} = -1022, e_{max} = 1023, \\ \text{smallest and largest repr. numbers } f_{min} \simeq 2.2 \cdot 10^{-308}, f_{max} \simeq 1.8 \cdot 10^{308}, \\ |e_{min}| < |e_{max}| \to \frac{1}{f_{min}} < f_{max} \end{aligned} \tag{24}$$

with machine precision $\epsilon_{\text{mach}} \approx 2.2 \cdot 10^{-16}$. There is even quad-double precision (128-bit) (not supported on hardware though and therefore relatively slow as it has to be emulated). Packages for nearly arbitrary precision also exist.

3.3 A more general view on sources of numerical error

In numerical computation, the typical error sources are

- rounding
- data uncertainty
- truncation (of terms in numerical schemata, e.g. in the approximation of a function by its Taylor series)

where we have now discussed rounding errors and their effects to some extent.

3.4 Backward error, forward error and condition number

Consider we approximate $y = f(x)$ as $\hat{y}$ in an arithmetic of limited precision, with $f : \mathbb{R} \to \mathbb{R}$.

- **Forward error:** The absolute or relative error between $\hat{y}$ and y is the forward error (living in the output space)
- **Backward error:** The backward error is the smallest Δx (in the input space) so that $f(x + \Delta x) = \hat{y}$, so the smallest perturbation where the exact function gives our approximate result.

If this Δx is sufficiently small, e.g. as small as the uncertainty in the data in the first place, we speak of backward stability. A weaker formulation is the mixed forward-backward error

$$\hat{y} + \Delta y = f(x + \Delta x), \quad |\Delta y| \leq \epsilon |y|, \quad |\Delta x| \leq \eta |x| \tag{25}$$

In the context of rounding errors, we call the algorithm numerically stable if $\hat{y}$ is almost the right answer for almost the right data (ϵ, η small).

3.4.1 Conditioning

Backward and forward error are connected by the conditioning of a problem, the sensitivity of the solution to perturbations in the data. Assuming $\hat{y} = f(x + \Delta x)$ and f differentiable, we have

$$\hat{y} - y = f(x + \Delta x) - f(x) = f'(x)\Delta x + \mathcal{O}((\Delta x)^2) \tag{26}$$

so the relative error is

$$\frac{\hat{y}-y}{y} = \frac{f'(x)\Delta x}{f(x)} + \mathcal{O}((\Delta x)^2) \tag{27}$$

leading to the relative condition number

$$\kappa(x) \underbrace{=}_{\text{i.A.}} \lim_{\epsilon \to 0^+} \sup_{||\Delta x|| \leq \epsilon} \frac{||y-\hat{y}||/||y||}{||\Delta x||/||x||} = \left| \frac{x f'(x)}{f(x)} \right| \tag{28}$$

for small Δx measuring the relative change in the output over a relative change in the input. As a rule of thumb

$$\text{forward error} \lesssim \text{condition number} \cdot \text{backward error} \tag{29}$$

so ill-conditioned problems can have large forward errors.

Application on Matrices

Consider the linear system $\underline{\underline{A}}\underline{y} = \underline{x}, \underline{y} = \underline{\underline{A}}^{-1}\underline{x}, \underline{\hat{y}} = \underline{\underline{A}}^{-1}(\underline{x} + \underline{\Delta x})$. The condition number follows as

$$\begin{aligned}
\kappa(\underline{\underline{A}}) &= \max_{\underline{x}, \underline{\Delta x} \neq 0} \frac{||\underline{\underline{A}}^{-1}\underline{x} - \underline{\underline{A}}^{-1}(\underline{x} + \underline{\Delta x})||/||\underline{\underline{A}}^{-1}\underline{x}||}{||\underline{\Delta x}||/||\underline{x}||} \\
&= \max_{\underline{\Delta x} \neq 0} \frac{||\underline{\underline{A}}^{-1}\underline{\Delta x}||}{||\Delta x||} \max_{\underline{x} \neq 0} \frac{||\underline{x}||}{||\underline{\underline{A}}^{-1}\underline{x}||} \\
&= \max_{\underline{\Delta x} \neq 0} \frac{||\underline{\underline{A}}^{-1}\underline{\Delta x}||}{||\underline{\Delta x}||} \max_{\underline{y} \neq 0} \frac{||\underline{\underline{A}}\underline{y}||}{||\underline{y}||} \\
&= ||\underline{\underline{A}}^{-1}|| \cdot ||\underline{\underline{A}}||
\end{aligned} \tag{30}$$

where we used the definition of the matrix norm $||\underline{\underline{A}}|| = \max_{\underline{x} \neq 0} \frac{||\underline{\underline{A}}\underline{x}||}{||\underline{x}||}$. For large condition numbers, small perturbations in the input $\underline{x}$ lead to large changes in the solution $\underline{y}$.

Part II

Simulation Methods

The dynamical evolution of physical systems is described using differential equations. Numerical methods for solving differential equations and the rise of computers have allowed for accurate modeling of complex dynamical systems that could hardly be approached by analytical means even under the usage of perturbation theory (compare Moser, 1978).

Oftentimes, we face an initial value problem (IVP) where from initial values from the functions to solve and values for their derivatives as necessary, the evolution is sought to be calculated. In a boundary value problem a differential equation is given together with a set of additional constraints (e.g. Sturm-Liouville problems).

4 Integration of ordinary differential equations

Our aim is solving an ordinary differential equation (ODE) $\partial_t \underline{y} = \underline{f}\left(\underline{y}\right)$ with initial values $\underline{y}(t = t_0) = \underline{y}_0$. Notice that $\underline{f} = \underline{f}(\underline{y}, t)$ can be handled by augmenting $\tilde{\underline{y}} = \begin{pmatrix} \underline{y} \\ t \end{pmatrix}$ and $\tilde{\underline{f}}(\tilde{\underline{y}}) = \begin{pmatrix} \underline{f}(\tilde{\underline{y}}) \\ 1 \end{pmatrix}$.

4.1 Notes on ODEs

4.1.1 Converting to a first order system

Ordinary differential equations only contain derivatives with respect to one variable. Note, however, that higher order derivatives with respect to that variable can occur. We can get to the form $\partial_t \underline{y} = \underline{f}\left(\underline{y}\right)$ by converting to a coupled first order system.

Consider the n-th order ODE

$$\partial_t^n y(t) = f\left(y(t), \partial_t y(t), \ldots, \partial_t^{n-1} y(t), t\right), \quad f : U \subset \mathbb{R} \times \mathbb{K}^n \to \mathbb{K} \tag{31}$$

for instance a pendulum with damping

$$\partial_t^2 \phi = -\omega_0^2 \sin\phi - \gamma \partial_t \phi, \quad \gamma, \omega_0 \in \mathbb{R} \tag{32}$$

Now we define the variables

$$u_m = \partial_t^m y(t), \quad m \in \{0, \dots, n-1\} \tag{33}$$

leading to the coupled first order system

$$\partial_t \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n-2} \\ u_n \end{pmatrix} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ f\left(t, u_0, u_1, \dots, u_{n-1}\right) \end{pmatrix} \rightarrow \partial_t \underline{u} = \underline{f}\left(t, \underline{u}\right) \tag{34}$$

Using ϕ for the angle and $\omega = \partial_t \phi$ for the angular velocity, we can write the pendulum as

$$\partial_t \begin{pmatrix} \phi \\ \omega \end{pmatrix} = \begin{pmatrix} \omega \\ -\omega_0^2 \sin\phi - \gamma\omega \end{pmatrix} \tag{35}$$

4.1.2 Existence and uniqueness of an ODE solution for an initial value problem - Picard-Lindelöf and Lipschitz condition

For the initial value problem $\partial_t \underline{y} = \underline{f}\left(\underline{y}\right), \underline{y}(t_0) = \underline{y}_0$ to have a unique solution in the vicinity of $(\underline{y}_0, t_0)$, i.e. for the change around that point, to uniquely determine the development, this change must be *well-behaved*, $\underline{f}$ must be *Lipschitz-continuous*.

$$\forall(\underline{y}, t), (\underline{z}, t) \text{ in the vicinity of } (\underline{y}_0, t_0) : ||\underline{f}(\underline{y}, t) - \underline{f}(\underline{z}, t)|| \leq \lambda ||\underline{y} - \underline{z}|| \tag{36}$$

with $\lambda > 0$ and $||\cdot||$ being an arbitrary vector norm. The slope of the line connecting two close-by evaluations of $\underline{f}$ must be bounded by λ. This is guaranteed for $\underline{f}$ being continuous and sufficiently often differentiable with bounded derivatives and more so $\underline{f}$ analytic.

4.2 Introduction of Numerical Integration at the hand of the two-body problem

Our aim is computationally modelling the interaction of two-bodies. This lends itself well as an example, as stepping the system forward in time is easy to imagine visually, an analytic solution exists to which we might compare numerical solution and it guides us to the problem of conserved quantities and symplectic integrators.

4.2.1 The two-body problem

For the two-body problem (illustrated in figure 9) the equations of motion are

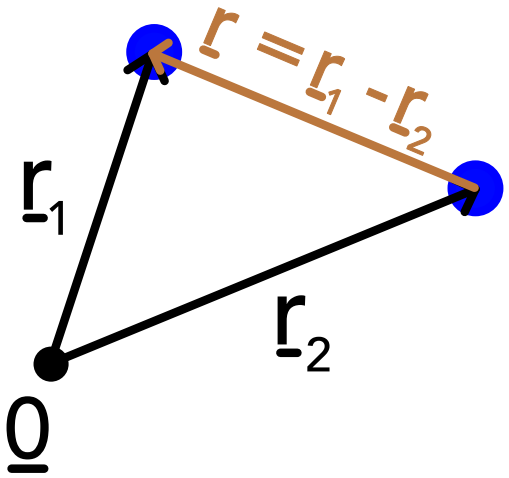


Figure 9: Illustration of the two-body problem.

$$\begin{aligned} m_1 \partial_t^2 \underline{r}_1 &= -G \frac{m_1 m_2}{|\underline{r}|^3} (\underline{r}) \\ m_2 \partial_t^2 \underline{r}_2 &= +G \frac{m_1 m_2}{|\underline{r}|^3} (\underline{r}) \end{aligned} \tag{37}$$

for $\underline{r} = \underline{r}_1 - \underline{r}_2$. Subtracting both yields

$$\partial_t^2 \underline{r} = -G \frac{M}{|\underline{r}|^3} \underline{r} \tag{38}$$

with $M = m_1 + m_2$. Which is equivalent to the equation of motion of a single body of mass $\mu = \frac{m_1 m_2}{M}$ in a potential $U(r) = -G\frac{m_1 m_2}{r} = -G\frac{M\mu}{r}$.

We can write this as the first order system

$$\partial_t \begin{pmatrix} \underline{r} \\ \underline{v} \end{pmatrix} = \begin{pmatrix} \underline{v} \\ -G\frac{M}{|\underline{r}|^3} \underline{r} \end{pmatrix} \tag{39}$$

4.2.2 Integrals of Motion

The following quantities are conserved along the trajectories of m_1 and m_2 and are therefore useful sanity checks for simulations.

- Total energy

$$E = T + U = \frac{\mu}{2} v^2 - \frac{GM}{r} \mu \tag{40}$$

- Angular momentum (perpendicular to the orbital plane)

$$\underline{L} = \underline{r} \times \underline{p} = \underline{r} \times \mu \underline{v} \tag{41}$$

- Laplace-Runge-Lenz vector (here in its dimensionless form, the eccentricity vector)

$$\underline{e} = \frac{\underline{v} \times \underline{j}}{GM} - \underline{\hat{e}}_r, \qquad \text{specific angular momentum } \underline{j} = \frac{\underline{L}}{\mu} \text{ eccentricity } e = ||\underline{e}|| \tag{42}$$

Note: The 1-body Kepler problem has 6 degrees of freedom (phase-space coordinates), of which one cannot be conserved, as nothing should be able to tell us the initial time of our motion. Therefore, only 5 quantities can be conserved and the Laplace-Runge-Lenz vector indeed only adds 1 more conserved degree of freedom (taking E and $\underline{L}$ as primary conserved quantities, $\underline{e}$ only has one degree of freedom).

Additional notes on the Laplace-Runge Lenz vector

The Lenz vector is conserved in all $\frac{1}{r}$-potentials, like the gravitational or Coulomb potential, for instance in the Hydrogen atom (but not for multi-electron atoms). Kepler-orbits are conic sections and the Laplace-Runge-Lenz vector is illustrated in figure 10.

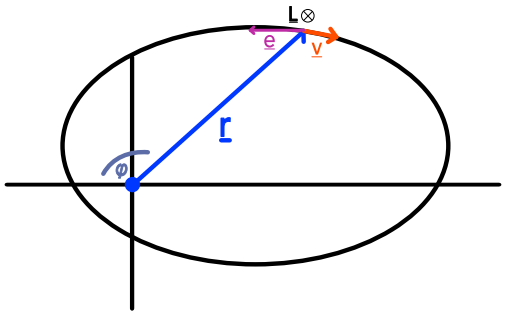


Figure 10: Illustration of the Laplace-Runge-Lenz vector.

From our pictorial evidence, we see that $\underline{e}$ is points along the semi-major axis. Note here we have drawn that $\underline{r} = \underline{r}_1 - \underline{r}_2$ follows a conic section. Likewise m_1 and m_2 move on conic sections with respect to the center of mass, $\underline{0} \underbrace{=}_{\text{o.b.d.A.}} \frac{1}{M}\left(m_1 \underline{r}_1 + m_2 \underline{r}_2\right)$ leading to $\underline{r}_1 = \frac{m_2}{M}\underline{r}$ and $\underline{r}_2 = -\frac{m_1}{M}\underline{r}$.

4.2.3 Kepler Orbits are Conic Sections

4.2.4 Connection of the Runge-Lenz vector to the eccentricity of a conic section

4.2.5 Dimensionless variables

4.2.6 Solving the two-body problem using explicit (aka forward) Euler

4.2.7 Probing the accuracy of an integration scheme - energy error of explicit Euler

4.3 Explicit Euler and it's shortcomings

The simplest method for solving an ODE is the Explicit Euler method

$$\underline{y}^{(n+1)} = \underline{y}^{(n)} + \underline{f}\left(\underline{y}^{(n)}\right)\Delta t, \quad \text{where} \quad \underline{y}^{(0)} = \underline{y}_0$$

which is explicit as the computation of $\underline{y}^{(n+1)}$ only depends on already known states.

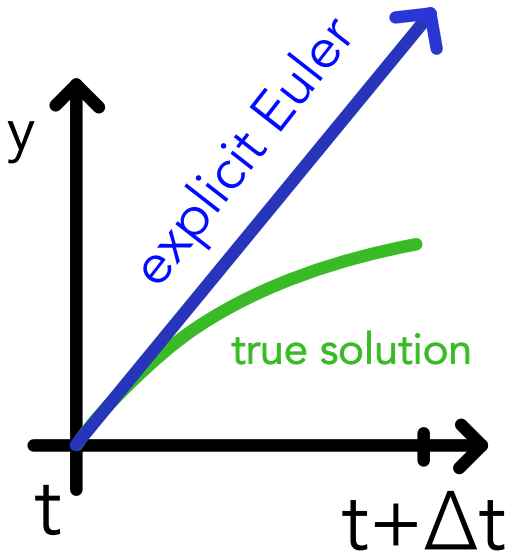


Figure 11: Illustration of one time step in the Explicit Euler scheme.

As illustrated in Figure 11 in every step we step forward along the current derivative $\underline{f}\left(\underline{y}^{(n)}\right)$.

4.3.1 Explicit Euler is only first order accurate | truncation error

A simple error approximation follows from Taylor expansion

$$\underline{y}(t + \Delta t) = \underline{y}(t) + \Delta t \underline{f}(t) + \mathcal{O}_s(\Delta t^2)$$

In each step we make an error $\mathcal{O}_s(\Delta t^2)$ so over some timespan T where we need $N_S = \frac{T}{\Delta t}$ steps we accumulate the error $N_S \mathcal{O}_S(\Delta t^2) = \mathcal{O}_T(\Delta t)$. We therefore call Explicit Euler first

order accurate.

Note: For a global error scaling with $\mathcal{O}_T(\Delta t^n)$ (n-th order accurate scheme), the local truncation error (of the Taylor expansion) must be $\mathcal{O}_s(\Delta t^{n+1})$.

4.3.2 Explicit Euler has stability issues

Stability analysis is a broad field, and the interested reader can find details in chapter IV.3 of Hairer and Wanner, 1996. For now, consider the ODE $\partial_t y = \alpha y, Re(\alpha) < 0, y(0) = y_0$ with the solution $y(t) = y_0 e^{\alpha y}$.

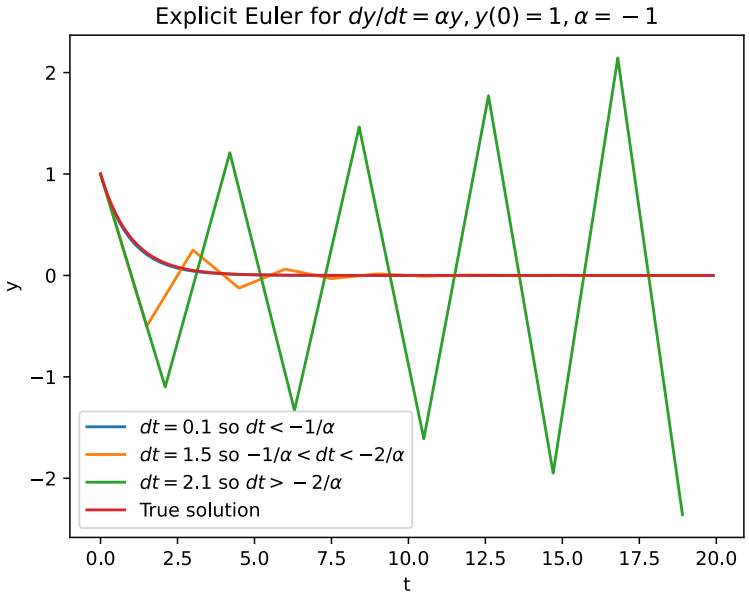


Figure 12: Linear stability of the Explicit Euler scheme.

The results of applying Explicit Euler for different step sizes Δt are shown in figure 12. At a small step size the correct solution is obtained, for a larger step size the numerical solution becomes oscillatory and for even larger step sizes it diverges. We can quantitatively explain this behavior by looking at the Euler steps

$$\begin{aligned} y^{(n+1)} &= y^{(n)} + \alpha y^{(n)} \Delta t \\ &= y^{(n)}(1 + \alpha \Delta t) \\ &= y^{(0)}(1 + \alpha \Delta t)^{n+1} \end{aligned}$$

- $\Delta t < -\frac{1}{\alpha} \rightarrow$ we observe monotonous decrease (ok)
- $-\frac{1}{\alpha} < \Delta t < -\frac{2}{\alpha} \rightarrow$ oscillation (regarding the sign) but still decrease in the absolute value (problematic)

- $-\frac{2}{\alpha} < \Delta t \rightarrow$ an increasing, oscillating solution (very bad)

The growth factor $R(\alpha\Delta t) = 1 + \alpha\Delta t$ in $y^{(n+1)} = R(\alpha\Delta t)y^{(n)}$ is called stability function and

$$\mathcal{D} := \{z \in \mathcal{C} : |R(z)| \leq 1\} \quad \text{so} \quad D_{Euler} = \{z = \alpha\Delta t \in \mathcal{C} : |1 + z| \leq 1\}$$

is called region of absolute stability or linear stability domain. D_{Euler} is a finite region of absolute stability in form of a circle on the left of the complex plane (see figure 13).

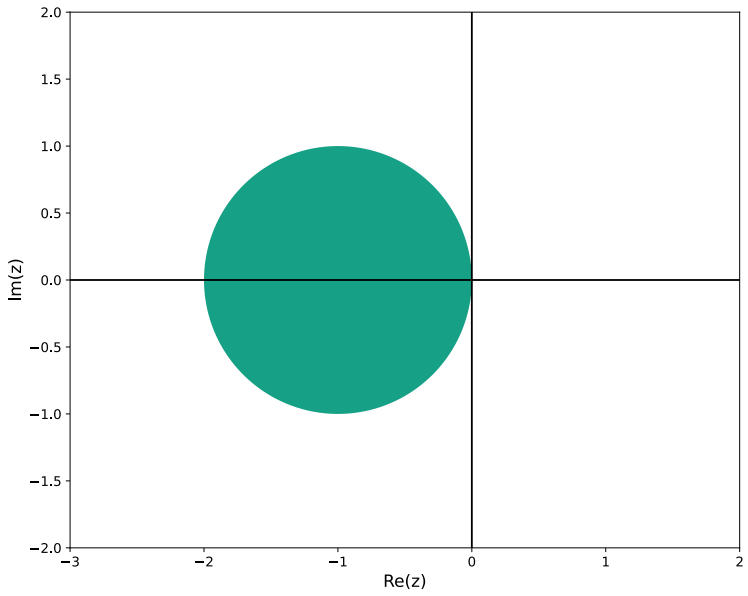


Figure 13: Region of absolute stability of the Explicit Euler method.

Problem: While in this example the stability constraint is easy to fulfill (we get a good solution for a reasonably large step-size), in problems with different timescales, with explicit Euler we must resolve the fastest one, even if its completely negligible (*stiff problems*).

4.4 Introduction of the Problem of Stiffness and Implicit Euler to the help

4.4.1 Introducing stiffness at the hand of a simple example

Consider the following ODE system (following Press et al., 2007, chapter 17.5)

$$\partial_t y_1 = 998y_1 + 1998y_2$$
$$\partial_t y_2 = -999y_1 - 1999y_2$$

with initial conditions $y_1(0) = 1$ and $y_2(0) = 0$. The system can be represented in matrix form as

$$\partial_t \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \underline{\underline{A}} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \quad \underline{\underline{A}} = \begin{pmatrix} 998 & 1998 \\ -999 & -1999 \end{pmatrix}$$

The eigenvalues of $\underline{\underline{A}}$ are $\lambda_1 = -1$ and $\lambda_2 = -1000$. The eigenvectors are $\underline{e}_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$ and $\underline{e}_2 = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$. The solution of the system is then

$$\begin{pmatrix} y_1(t) \\ y_2(t) \end{pmatrix} = \exp\left(\underline{\underline{A}}t\right) \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \end{pmatrix} \exp(-1t) + \begin{pmatrix} -1 \\ 1 \end{pmatrix} \exp(-1000t)$$

Let us now apply the Explicit Euler method to this system for different time-steps Δt. The result is shown in figure 14.

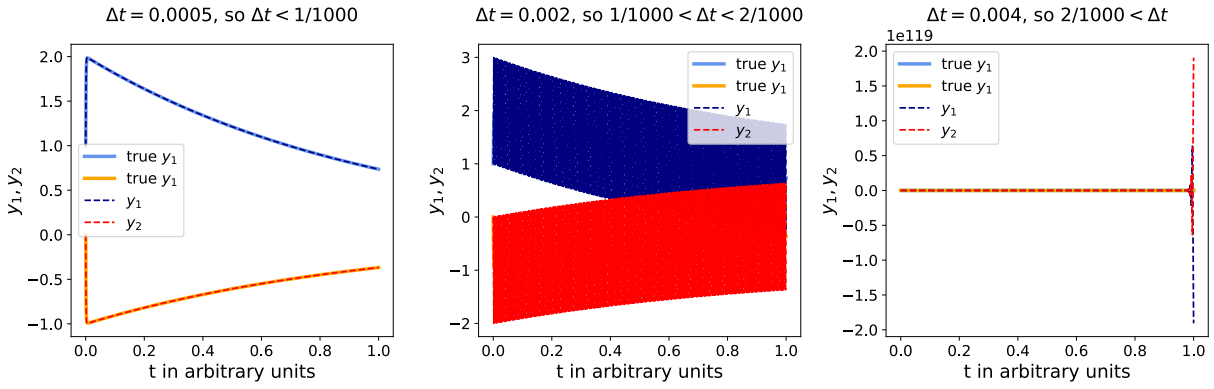


Figure 14: Numerical solution to the linear system $\partial_t \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \underline{\underline{A}} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ with $\underline{\underline{A}} = \begin{pmatrix} 998 & 1998 \\ -999 & -1999 \end{pmatrix}$ and $y_1(0) = 1$, $y_2(0) = 0$ using the Explicit Euler method for different time-steps Δt. The left panel shows the solution for $\Delta t = 0.0005$, the central one for $\Delta t = 0.002$ and the right one for $\Delta t = 0.004$.

Let us think back to the linear stability analysis of the Explicit Euler scheme for $\partial_t y = \alpha y, Re(\alpha) < 0, y(0) = y_0$. We had obtained

- $\Delta t < -\frac{1}{\alpha} \rightarrow$ we observe monotonous decrease (ok)
- $-\frac{1}{\alpha} < \Delta t < -\frac{2}{\alpha} \rightarrow$ oscillation (regarding the sign) but still decrease in the absolute value (problematic)
- $-\frac{2}{\alpha} < \Delta t \rightarrow$ an increasing, oscillating solution (very bad)

The same result holds in principle for our linear system - but with α replaced by the eigenvalue of largest magnitude of $\underline{\underline{A}}$, here $\lambda_2 = -1000$ (for the proof see Press et al., 2007, chapter 17.5).

As we move away from the origin, the fastest decreasing term $\propto \exp(-\lambda_2 t)$ in the true solution is completely negligible. However, in the explicit scheme it still sets the timescale that has to be resolved for a stable solution.

In the setting of $\partial_t y = \alpha y, Re(\alpha) < 0$ the stability constraint for Δt is not too problematic because the resulting step-size is reasonable compared to the timescale of the problem. In the case of an ODE with different timescales in the solution, however, we are often interested in the timescale of the slowest processes but in the explicit scheme we still need to resolve the fastest timescale which quickly becomes infeasable. This is the problem of stiffness and can - in such a linear setting with all negative eigenvalues of $\underline{\underline{A}}$ - be characterized by the stiffness ratio

$$\text{stiffness ratio} := \frac{\max_{\text{eigenvalues } \lambda_i \text{ of } \underline{\underline{A}}} |\text{Re } \lambda_i|}{\min_{\text{eigenvalues } \lambda_i \text{ of } \underline{\underline{A}}} |\text{Re } \lambda_i|} = \frac{\lambda_2}{\lambda_1} = 1000$$

A large stiffness ratio indicates that an explicit scheme like the Explicit Euler method would be very inefficient for following the slowest process.

4.4.2 A *definition* of stiffness

As discussed in Lambert, 1991 a hard mathematical definition of stiffness is difficult and we therefore resort to the broad practical definition (Lambert, 1991, chapter 6)

> »If a numerical method with a finite region of absolute stability, applied to a system with any initial conditions, is forced to use in a certain interval of integration a step length which is excessively small in relation to the smoothness of the exact solution in that interval, then the system is said to be stiff in that interval.«

An example for a numerical method with a finite region of absolute stability is the Explicit Euler method (see figure 13). In the example above, in spite of the fact that the solution is very smooth and the term $\propto \exp(-\lambda_2 t)$ is quickly negligible, we have to use excessively small steps.

4.4.3 Implicit Euler to the help

At the core of dealing with stiffness are implicit methods, the simplest representative being Implicit Euler.

An Implicit Euler step for solving $\partial_t \underline{y} = \underline{f}\left(\underline{y}\right)$ is given by

$$\underline{y}^{(n+1)} = \underline{y}^{(n)} + \underline{f}\left(\underline{y}^{(n+1)}\right)\Delta t \quad \text{where} \quad \underline{y}^{(0)} = \underline{y}_0$$

which is an implicit equation as $\underline{f}$ is evaluated at the new time step $\underline{y}^{(n+1)}$.

Intuition behind implicit Euler: We can write the implicit Euler step as $\underline{y}^{(n+1)} - \underline{f}\left(\underline{y}^{(n+1)}\right)\Delta t = \underline{y}^{(n)}$, so which is the point where when I sit on it and shoot back with the corresponding slope, I get back to where I am coming from. This is illustrated in figure 15.

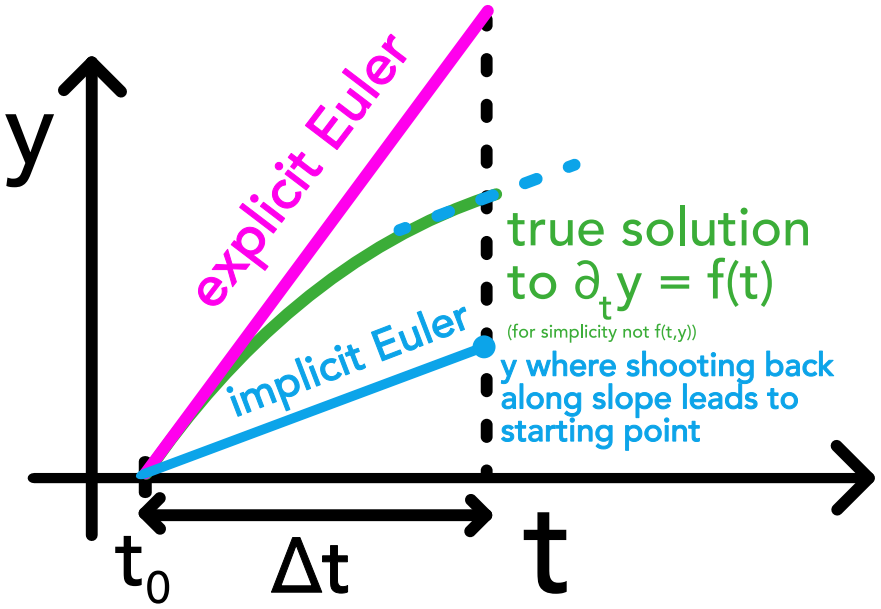


Figure 15: Illustration of the implicit Euler step.

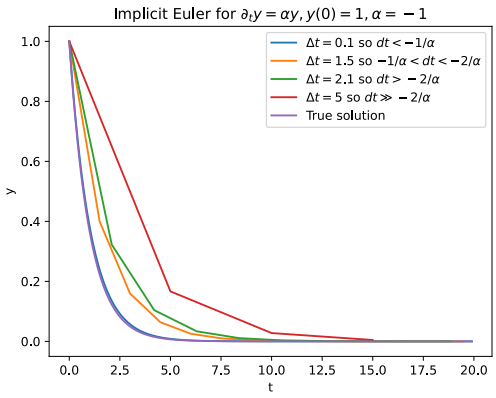
Note: Implicit Euler is often referred to as backward Euler and the explicit Euler as forward Euler.

Problem: Note that implicit Euler is also a first order accurate scheme.

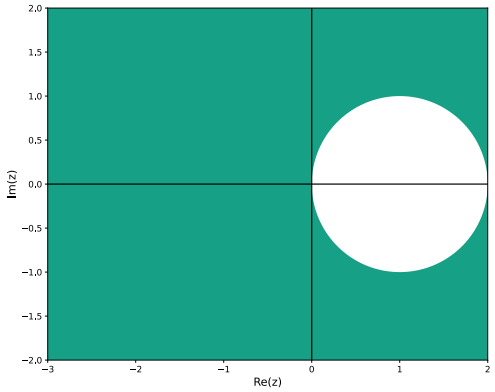
Region of absolute stability of the Implicit Euler method

As for the Explicit Euler method, we perform a linear stability analysis of the Implicit Euler method for $\partial_t y = \alpha y, Re(\alpha) < 0, y(0) = y_0$. We obtain

$$y^{(n+1)} = y^{(n)} + \alpha y^{(n+1)}\Delta t \quad \Rightarrow \quad y^{(n+1)} = \frac{1}{1-\alpha\Delta t} y^{(n)}$$



(a) Linear stability of Implicit Euler.



(b) Region of absolute stability of the Implicit Euler method (shaded in green).

Figure 16: Stability of the Implicit Euler scheme.

which decreases for any $\Delta t > 0$ (illustrated in figure 16a). For large time-steps, the result is inaccurate (Implicit Euler is a first order scheme) but the solution remains stable. As of the stability function $R(z) = \frac{1}{1-z}$ the region of absolute stability is given by

$$\mathcal{D}_{\text{implicit euler}} = \{z \in \mathbb{C} \mid |R(z)| < 1\} = \{z \in \mathbb{C} \mid |1 - z| > 1\}$$

which is illustrated in figure 16b. The whole left half plane is included in the region of absolute stability and the method is therefore unconditionally stable.

Implicit Euler for stiff linear ODEs

As Implicit Euler is unconditionally stable, the fast oscillating terms resulting from

$$\partial_t \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \underline{\underline{A}} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \quad \underline{\underline{A}} = \begin{pmatrix} 998 & 1998 \\ -999 & -1999 \end{pmatrix}$$

with initial conditions $y_1(0) = 1$ and $y_2(0) = 0$ are no problem as illustrated in figure 17, where in spite of the relatively large time-step a good approximation of the solution is obtained.

The implicit step for such a linear system $\partial_t \underline{y} = \underline{\underline{A}}\underline{y}$ is

$$\underline{y}^{(n+1)} = \underline{y}^{(n)} + \underline{\underline{A}}\underline{y}^{(n+1)}\Delta t \quad \Rightarrow \quad \left(\underline{\underline{1}} - \underline{\underline{A}}\Delta t\right) \underline{y}^{(n+1)} = \underline{y}^{(n)}$$

which means that to make a step we have to solve a linear system which is usually done by

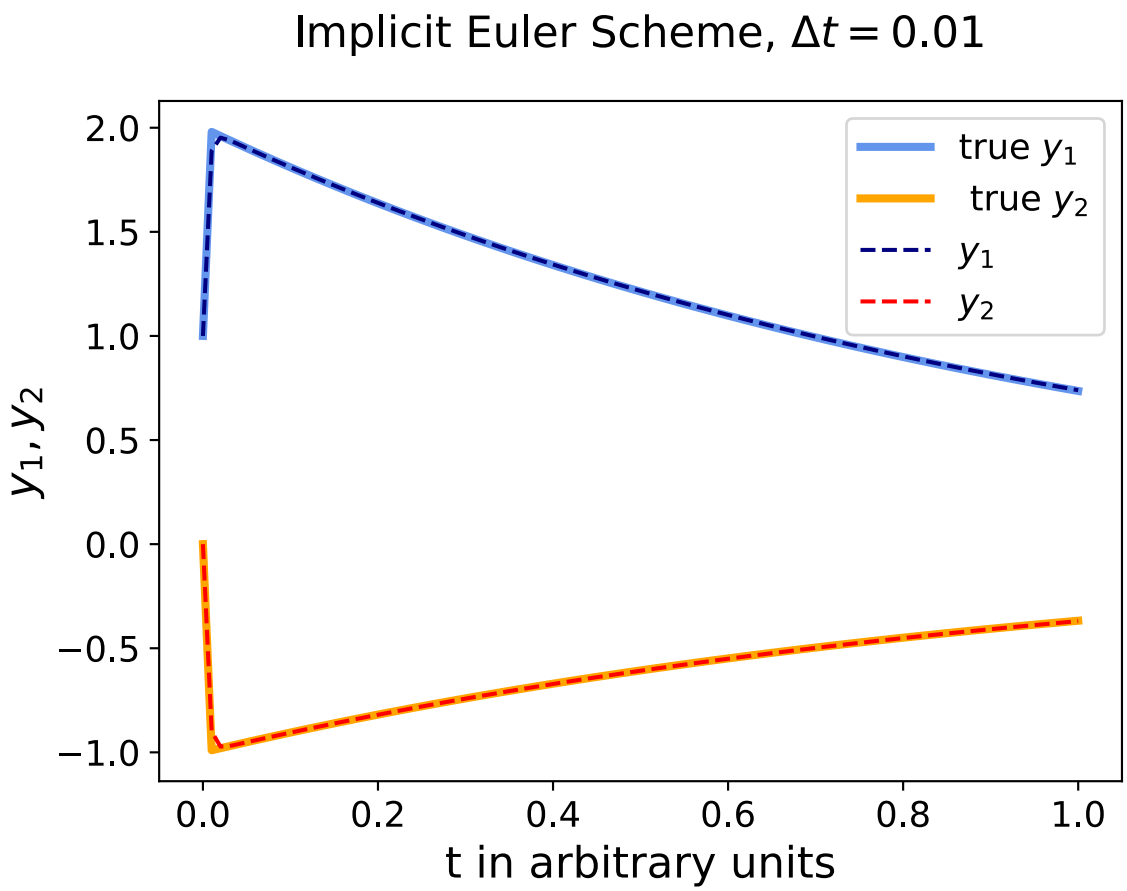


Figure 17: The same problem as in figure 14 is now approached using the Implicit Euler method and a relatively large time-step of $\Delta t = 0.01$.

matrix decomposition (like LU decomposition).

But how can we approach non-linear ODEs using the Implicit Euler method?

To perform an implicit step

$$\underline{y}^{(n+1)} = \underline{y}^{(n)} + \underline{f}\left(\underline{y}^{(n+1)}\right)\Delta t$$

for a non-linear system $\partial_t \underline{y} = \underline{f}\left(\underline{y}\right)$ like the Davis-Skodje equation

$$\begin{aligned}\dot{y}_1(t) &= -y_1(t)\\ \dot{y}_2(t) &= -\gamma y_2(t) + \frac{(\gamma-1)y_1(t) + \gamma y_1^2(t)}{\left(1+y_1(t)\right)^2}\end{aligned}$$

where γ is a measure for the stiffness (see Heiter, 2012, chapter 2.4) we reformulate the implicit step as a root-finding problem

$$\underline{0} = \underline{y}^{(n+1)} - \underline{y}^{(n)} - \Delta t \underline{f}\left(\underline{y}^{(n+1)}\right), \quad \underline{g}\left(\underline{\xi}\right) := \underline{\xi} - \underline{y}^{(n)} - \Delta t \underline{f}\left(\underline{\xi}\right)$$
$$\rightarrow \underline{0} = \underline{g}\left(\underline{\xi}\right) \Leftrightarrow \underline{\xi} = \underline{y}^{(n+1)}$$

where each of those time-steps is solved using Newton's method (or quasi-Newton)

$$\underline{\xi}_{k+1} = \underline{\xi}_k - \underline{\underline{J}}_{\underline{g}}^{-1}\left(\underline{\xi}_k\right)\underline{g}\left(\underline{\xi}_k\right), \quad \underline{\underline{J}}_{\underline{g}} = \underline{\underline{1}} - \Delta t\gamma\left(\underline{\xi}_k\right)\underline{\underline{J}}_{\underline{f}}$$

$$\underline{\xi}_0 = \underline{y}^{(n)}, \quad \underline{\xi}_m \rightarrow \underline{y}^{(n+1)} \quad \text{for} \quad m \rightarrow \infty$$

where $\underline{\underline{J}}_{\underline{f}}$ is the Jacobian of $\underline{f}$. In Quasi-Newton the Jacobian is only recalculated once per time-step in the Euler method

$$\underline{\xi}_{k+1} = \underline{\xi}_k - \underline{\underline{J}}_{\underline{g}}^{-1}\left(\underline{\xi}_0\right)\underline{g}\left(\underline{\xi}_k\right)$$

For the Davis-Skodje problem mentioned above some Implicit Euler steps are drawn into the stream plot of the equation in figure 18. Here, one can also see the intuition behind Implicit Euler steps: One searches a point where the derivative is such that shooting back with this slope leads back to the point we are coming from, as

$$\underline{y}^{(n)} = \underline{y}^{(n+1)} - \underline{f}\left(\underline{y}^{(n+1)}\right)\Delta t$$

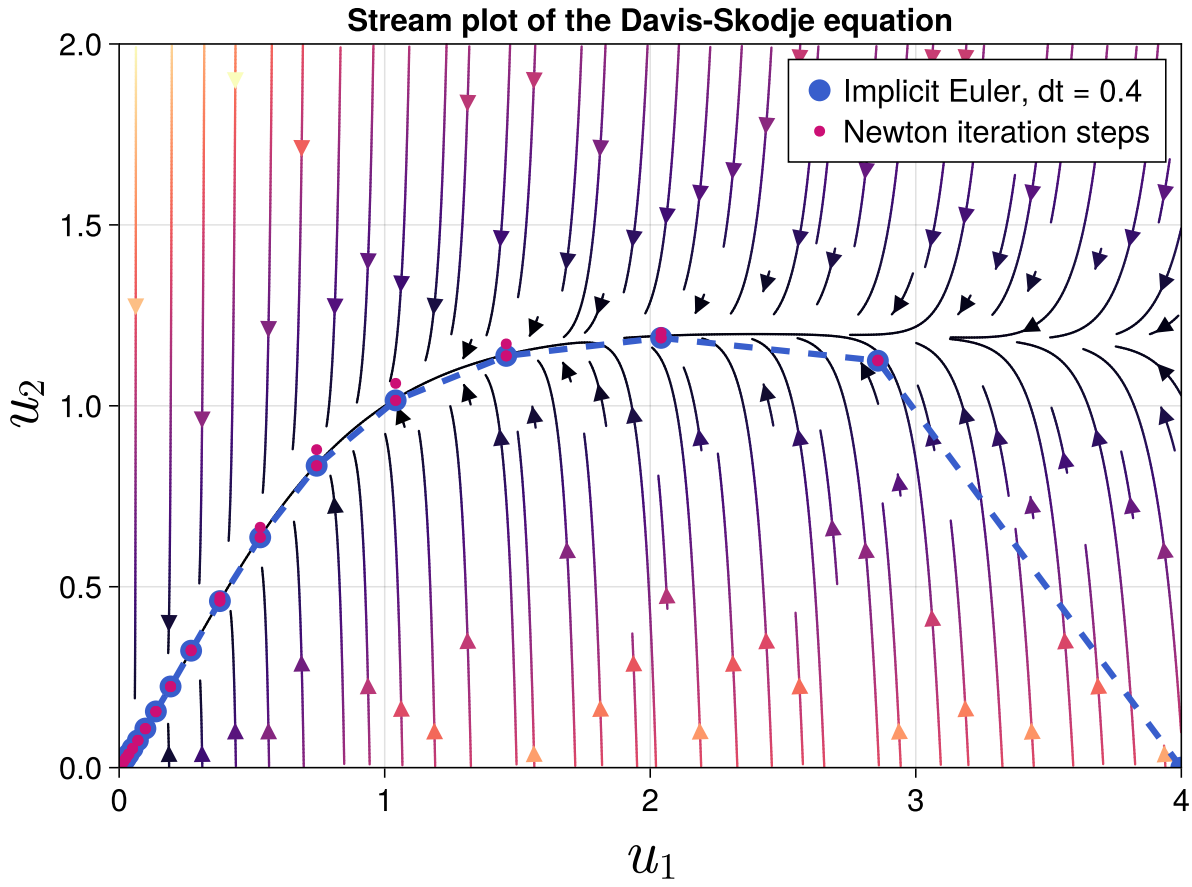


Figure 18: Stream plot of the Davis-Skodje equation with some Implicit Euler steps also drawn. The direction of the Implicit Euler steps is from right (starting at (4, 0)) to left.

The steps of the Newton iteration done for each Implicit Euler step can most intuitively be understood in the formulation as the linear equation

$$\underline{b} := \underline{g}\left(\underline{\xi}_k\right) = \underline{\underline{J}}_{\underline{g}}\left(\underline{\xi}_k - \underline{\xi}_{k+1}\right) = \underline{\underline{J}}_{\underline{g}}\,\underline{a}, \quad \underline{a} := \underline{\xi}_k - \underline{\xi}_{k+1}$$

which is also the equation solved on the computer using matrix decomposition. $\underline{\underline{J}}_{\underline{g}}\,\underline{a}$ is the directional derivative of $\underline{g}$ in the direction of $\underline{a}$ and in a step of the Newton iteration we search for a step $\underline{a}$ that gets us from $\underline{0}$ to $\underline{b}$ in other words $\underline{\xi}_{k+1} = \underline{\xi}_k - \underline{a}$.

Problem: While the Implicit Euler method is unconditionally stable, performing the implicit step for non-linear ODEs requires solving a non-linear equation with some root-finding algorithm, which can be even more costly than doing small explicit steps if no proper care (e.g. smart forward differentiation in the root finding) is taken.

References

Biamonte, Jacob et al. (2017). »Quantum machine learning«. In: *Nature* 549.7671, pages 195–202.

Bryant, Randal E and David Richard O'Hallaron (2011). *Computer systems: a programmer's perspective*. Prentice Hall.

Hairer, Ernst and Gerhard Wanner (1996). *Solving Ordinary Differential Equations II*. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-05221-7. URL: https://doi.org/10.1007/978-3-642-05221-7.

Heiter, Pascal Frederik (2012). »On Numerical Methods for Stiff Ordinary Differential Equation Systems«. Master's thesis. Ulm University. URL: https://www.uni-ulm.de/fileadmin/website_uni_ulm/mawi.inst.070/abschlussarbeiten/masterthesis_pfh.pdf.

Higham, Nicholas J. (2002). *Accuracy and Stability of Numerical Algorithms*. Second. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898718027. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898718027.

Lambert, J.D. (1991). *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem*. Wiley. URL: https://books.google.de/books?id=P0vPnQEACAAJ.

Moser, Jürgen (1978). »Is the solar system stable?« In: *The Mathematical Intelligencer* 1.2, pages 65–71. DOI: 10.1007/BF03023062. URL: https://doi.org/10.1007/BF03023062.

Press, William H. et al. (2007). *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd edition. USA: Cambridge University Press.

Ulmann, Bernd (May 2020). *Analog and Hybrid Computer Programming*. De Gruyter. DOI: 10.1515/9783110662207. URL: http://dx.doi.org/10.1515/9783110662207.