MA5233 Computational Mathematics

Lecture 1: Machine Numbers

Simon Etter

Semester I, AY 2020/2021

Machine Numbers

Lecture outline

- What are computers?
- How do we do basic arithmetic on a computer?

Introduction: What are computers?

Computer are perhaps the most complicated piece of engineering in our everyday lives. Understanding how they work and why they work the way they do would require advanced degrees in physics, chemistry, electrical engineering, computer science and economics.

The purpose of this first part of this lecture is to establish some shared understanding of what computers are, and what they can and cannot do.

Machine Numbers

Computers as function evaluators

A good starting point for understanding computers is to clarify what we want them to do for us.

Arguably the most important application of computers these days is to send and display information (think WhatsApp, Facebook, YouTube, etc.).

However, this is strictly speaking an abuse of terminology; "messenger" or "displayer" would be more accurate to describe this aspect of computers.

What the term "computer" actually describes is a machine for mapping inputs to outputs, i.e. a tool for evaluating functions. Some examples:

- Word finding maps a keyword and a text to the position of the keyword in the text.
- A key component of an image viewer is a function which maps JPEG-compressed images to a matrix of RGB values which can then be displayed on the screen.

This function evaluation aspect of computers will be a recurring theme in this module, see next slide.

Machine Numbers

Functions in computational mathematics

Every topic discussed in this module will follow the same outline.

- Define a function $f(x)$.
  Examples:
  - Linear system solving: $(A, b) \mapsto A^{-1}b$.
  - Root finding: $f \mapsto x$ such that $f(x) = 0$.
  - PDE solvers: $f \mapsto u$ such that $-\Delta u = f$.
- Present algorithms for evaluating $f(x)$.
- Show that these algorithms are correct (i.e. show that they indeed evaluate $f(x)$), and discuss runtime and memory requirements.

A special feature of computational mathematics is that most functions studied in this field cannot be evaluated in finite time.

Example: $\sqrt{2} = 1.4142\ldots$ has infinitely many digits; hence no algorithm can ever compute a decimal representation of $\sqrt{2}$ in finite time (and memory).

In computational mathematics, correctness will therefore usually be replaced by convergence, which is the study of how much time we need to invest to get a result of a certain accuracy.

Machine Numbers

A mathematical model of computers

Now that we have a clearer picture of what we want from computers, we can move on to discussing how they work.

As mentioned, computers are very complex. However, for the purpose of correctness we can ignore virtually all of this complexity.

In fact, if we ignore performance then we can pretend that a computer consists of only two components.

- Memory: a mutable sequence of bits (i.e. `0` and `1`).
- Processor: a unit which can
  - read a sequence of bits I from memory,
  - compute another sequence of bits O based on I, and
  - write O back to memory.

Remark

Other computer designs (e.g. analogue, non-binary or quantum) have been proposed, but virtually all contemporary computers use the binary model presented above, and this will remain so for the foreseeable future.

Machine Numbers

Summary

- The purpose of computers is to evaluate functions, i.e. to map inputs to outputs.
- Virtually all computers are binary, i.e. they work in terms of `0` and `1`.
- Correctness and performance are key algorithm properties.

Introduction: Machine numbers

All functions of interest in computational mathematics involve numbers either as their input or their output.

To use computers for these functions, we hence need a way to represent arbitrary numbers in terms of `0` and `1`.

I will next present binary representations for

- unsigned integers,
- signed integers,
- real numbers.

Machine Numbers

Def: Binary representation of unsigned integer

Bit string $b_{n-1} \dots b_0$ represents the number $\sum_{k=0}^{n-1} b_k \times 2^k$.

Examples

bit string	number and conversion
1	$1 = 1 \times 2^0$
10	$2 = 1 \times 2^1 + 0 \times 2^0$
11	$3 = 1 \times 2^1 + 1 \times 2^0$
100	$4 = 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
1011	$11 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

Machine Numbers

Binary arithmetic

Addition and multiplication work just like you learnt in school:

$$\begin{array}{rll} & 1\,1\,1\,1 & = 15 \\ + & 1\,0\,1\,1\,0 & = 22 \\ & {\scriptstyle 1\ 1\ 1\ 1\ 0} & \text{carried digits} \\ \hline & 1\,0\,0\,1\,0\,1 & = 37 \end{array}$$

$$\begin{array}{rll} & 1\,1\,0 & = 6 \\ \times & 1\,0\,1 & = 5 \\ \hline & 0\,0\,0 & \\ & 1\,0\,1 & \\ 1\,0\,1 & & \\ \hline & 1\,1\,1\,1\,0 & = 30 \end{array}$$

Machine Numbers

Fixed-size binary representation of unsigned integers

Problem: In writing, we distinguish "10, 11" from "1011" using comma and space, but computers don't know commas and spaces.

Solution: Fix the number of bits per number by adding zeros on the left.

Example: If we fix four bits per integer, then we write

$$0010 \text{ instead of } 10, \qquad \text{and} \qquad 0011 \text{ instead of } 11.$$

Fixed-size binary representation allows us to unambiguously merge a sequence of numbers into a single bit string, or conversely split a bit string into a sequence of numbers.

Example: $0010, 0011 \longleftrightarrow 00100011.$

Machine Numbers

Consequences of fixed-size binary representation

Fixing the number of bits n means that there is a largest representable number

$$I_{\max} = \underbrace{111\ldots111}_{n \text{ ones}} = 2^n - 1.$$

By convention, if integer arithmetic leads to numbers $> I_{\max}$, then the leading digits are discarded.

Example: $1111 + 1000 = 10111 \quad \rightarrow \quad 0111$

This behaviour is called *integer overflow* and a common source of errors.

Case study: Maiden launch of the Ariane 5 rocket

- Ariane 5: space rocket of the European Space Agency (ESA).
- Developed as a more powerful successor to Ariane 4.
- Maiden flight took place on 4 June 1996.
  Rocket and cargo were valued at $500 million.
  `https://youtu.be/gp_D8r-2hwk`

Machine Numbers

Case study: Maiden launch of the Ariane 5 rocket (continued)

Reason for failure:

- Ariane 5 used mostly the same software as Ariane 4.
- In some parts of the guidance software, the horizontal velocity v is stored as a 16-bit integer.
- For Ariane 4, v always remained safely within the range covered by 16-bit integers. For Ariane 5, v exceeded this range.
- This integer overflow broke the guidance system, which led to the rocket veering off its flight path and triggering the self-destruct mechanism.

Take-away:

It is tempting to assume that machine numbers behave exactly like their mathematical counterparts, but sometimes doing so can lead to disaster.

Reference:
`http://www-users.math.umn.edu/~arnold/disasters/ariane.html`.

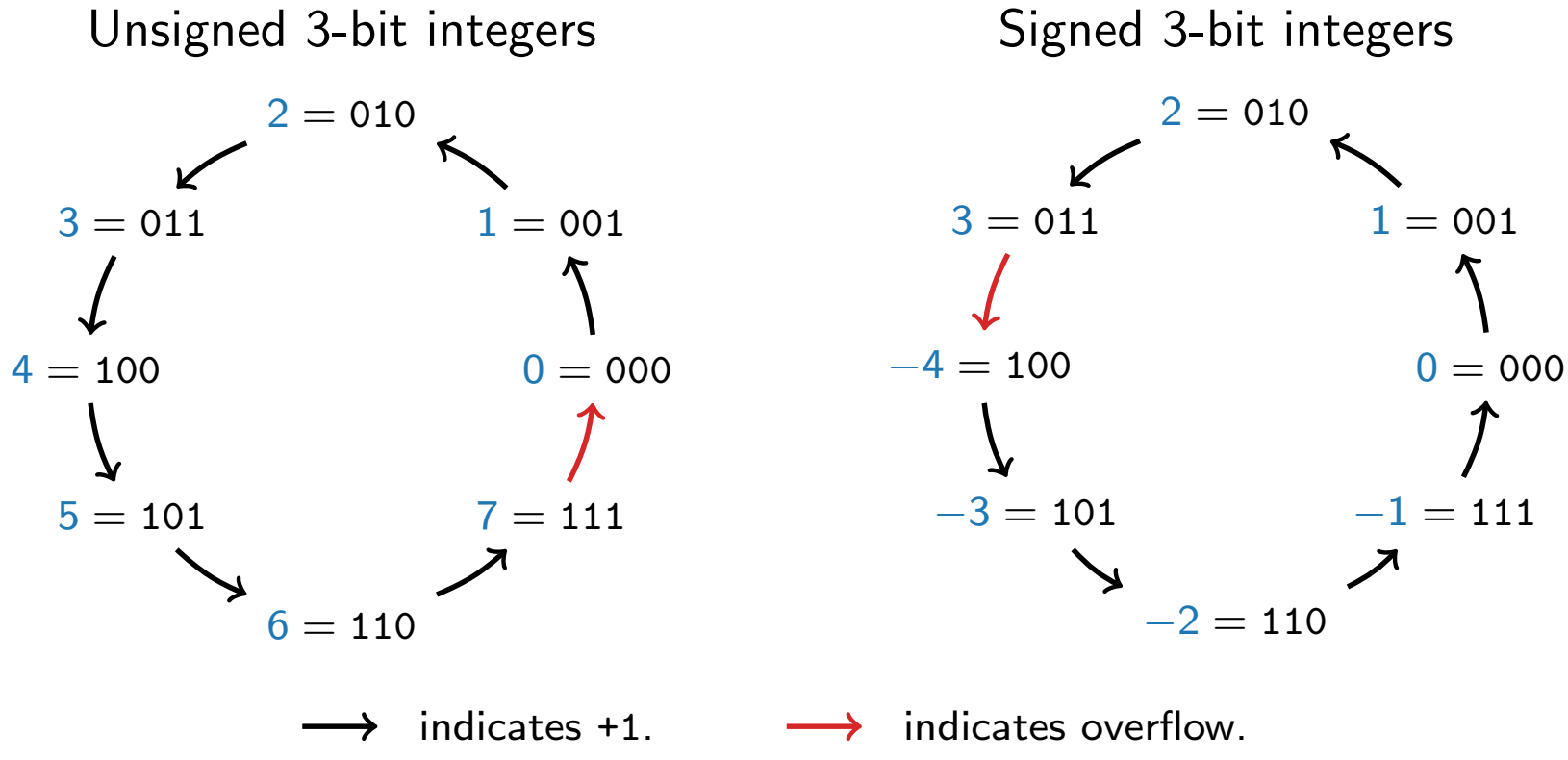
Machine Numbers

Binary representation of signed integers

Observation: Integer overflow turns integers into a circle.

Consequence: Signed integers can be implemented simply by reinterpreting the bit strings!

Example: `001 (1) + 111 (7) == 000 (0)`; hence "7 == -1".



Machine Numbers

Two's complement representation of signed integers

The signed integer representation introduced on the previous slide is known as *two's complement*.

In mathematical terms, the representation is

$$b_{n-1} \ldots b_0 \qquad \longleftrightarrow \qquad -b_k\, 2^{n-1} + \sum_{k=0}^{n-2} b_k\, 2^k .$$

The key advantage of two's complement is that signed and unsigned arithmetic are exactly the same. The only difference is the interpretation of the inputs and outputs.

In particular, signed arithmetic does not require special treatment for positive / negative numbers and zero, which would be required if we implemented signed integers using a sign bit.

Machine Numbers

Integer types in Julia

- Julia provides several `IntN` and `UIntN` types, where `U` indicates unsigned and `N` indicates the number of bits.

  Example: `Int64` = signed 64-bit integer.
- `Int` and `UInt` choose number of bits depending on your hardware.

  On most computers, `Int == Int64` and `UInt == UInt64`.
- `BigInt` is of variable width and hence never overflows.

  Downside: all operations are much slower.
- Integer literals (e.g. 42) are represented as `Int`s.
- You can convert an integer `x` to type `T` using `T(x)`.
- You can see the binary representation using `bitstring(x)`.
- Most functions preserve the type information.

See the code file of this lecture for illustrations.

Machine Numbers

Machine arithmetic for real numbers

An important new aspect of real arithmetic is that much of it cannot be modeled exactly on a computer.

Some examples:

- Multiplying two n-digits numbers yields a $(2n-1)$-digit number. Hence number of digits becomes unmanageable very quickly.

  Example 1: $1.1 \times 1.1 = 1.21$. We start with 2 digits and end with 3 digits.
  Example 2: Starting with $n = 2$ digits and doing 1000 multiplications would yield a $2^{1000} \approx 10^{300}$-digit result. Number of atoms in known universe is 10^{80}.

- $\sqrt{\cdot}$, exp, sin, ... immediately yield results with infinitely many digits.

This aspect makes real arithmetic fundamentally different from integer arithmetic (which is exact up to over-/underflow).

Key question for designing a binary representation of real numbers:
When and how should we truncate?

Truncate: drop digits to obtain an approximate but more manageable number.

Machine Numbers

Def: Rational representation for real numbers

A pair of integers x, y representing the number x/y.

Good: Allows for exact arithmetic.

Bad: How to truncate?

- Option 1: Find $\arg\min_{\{(\tilde{x},\tilde{y}) \mid \tilde{y} \leq y_{\max}\}} |\tilde{x}/\tilde{y} - x/y|$.
  This can be done, but it is expensive.
- Option 2: Fix $\tilde{y}$ and only optimise over $\tilde{x}$.
  This is essentially the fixed- and floating-point representations introduced below.

Conclusion. Rational numbers can be useful for short calculations where exactness is useful / important. Most languages provide a `Rational` number type, but it is only used in very special circumstances.

Machine Numbers

Def: Fixed-point representation for real numbers

An integer f and a fixed exponent E, representing the number $f \times 2^E$.

"Fixed" here means that we do not explicitly store E with the number but rather use it as meta-information for interpreting a given bit string. This is analogous to how the `N` works in the `IntN` types.

The Julia `FixedPointNumbers` package provides a `Fixed{T,E}` type, where `T` can be any signed integer type and `E` is the above exponent.

Example

```
bitstring(Fixed{Int8,2}(1    )) == "00000100"
bitstring(Fixed{Int8,2}(0.5  )) == "00000010"
bitstring(Fixed{Int8,2}(0.25 )) == "00000001"
bitstring(Fixed{Int8,2}(0.125)) == "00000000"
```

Note how `Fixed{Int8,2}(0.125)` leads to rounding since `0.125` is not representable in the `Fixed{Int8,2}` type.

Machine Numbers

Discussion

Fixed-point numbers are equally dense everywhere, i.e. the distance between any two consecutive fixed-point numbers x_-, x_+ is $|x_- - x_+| = 2^{-E}$ regardless of the magnitude of $x_\pm$.

This is almost always not what we want!

Example

Rounding the wealth of Bill Gates to the nearest billion is perfectly acceptable for most purposes.

Rounding the wealth of almost anyone else to the nearest billion would be absolutely meaningless.

The key difference between these two statements is that an error of 0.5 billion is small compared to the overall wealth of Bill Gates (about 0.5%), while it is very large compared to the wealth of most people.

This phenomenon turns out to be very common, see next slide.

Machine Numbers

Fact of life

For most real-world phenomena modeled using real numbers, we are interested in relative rather than absolute accuracy.

Put differently, if x is the number that we want to represent and $\tilde{x}$ is the closest number that we can represent, then we want $|\tilde{x} - x|/|x|$ to be small, not $|\tilde{x} - x|$.

Discussion

It is difficult to precisely pin down why relative accuracy is the right thing to aim for in most circumstances. I believe it is best to take this statement simply as experience accumulated by millions of mathematicians over centuries of numerical computations.

Machine Numbers

Def: Floating-point numbers

A triplet $s = \pm 1$ (sign), $e \in \mathbb{Z}$ (exponent) and $f \in [1, 2)$ (mantissa), representing the number $s \times f \times 2^e$.

Binary representation:

- s: a single bit which is `0` if $s = +1$ and `1` if $s = -1$.
- e: Shifted `UInt{E}` such that $e = 0$ is represented by `011...11`.

  Representing e as an `UInt` with shift rather than `Int` makes comparing floating-point numbers easier, see `https://en.wikipedia.org/wiki/Exponent_bias`.
- f: `Fixed{UInt{F},F-1}` where the first binary digit is always `1` and hence need not be stored.

  If the first bit of f is not `1`, we can make it `1` by changing the exponent. Example: $0.11 \times 2^2 = 1.1 \times 2^1$.

`UInt{F}` indicates an `F`-bit unsigned integer here, but it is not actually a Julia type. Have a look at the examples on the next slide if the above is confusing to you.

Thm: Spacing of floating-point numbers

If $(x_i = s \times f_i \times 2^e)_{i \in \{1,2\}}$ are two consecutive floats, then

$$|x_1 - x_2| = 2^{e-F+1} \quad \text{which is proportional to} \quad 2^e \approx |x_i|.$$

This is the key feature of floating-point numbers.

Machine Numbers

Example (spaces added for readability)

```
# Sign bit
bitstring(Float16(  1.0 ))) == "0 01111 0000000000"
bitstring(Float16( -1.0 ))) == "1 01111 0000000000"

# Exponent
bitstring(Float16( 0.5 ))) == "0 01110 0000000000"
bitstring(Float16( 1.0 ))) == "0 01111 0000000000"
bitstring(Float16( 2.0 ))) == "0 10000 0000000000"
bitstring(Float16( 4.0 ))) == "0 10001 0000000000"

# Mantissa
bitstring(Float16( 1.0  ))) == "0 01111 0000000000"
bitstring(Float16( 1.25 ))) == "0 01111 0100000000"
bitstring(Float16( 1.5  ))) == "0 01111 1000000000"
bitstring(Float16( 1.75 ))) == "0 01111 1100000000"

# Other
bitstring(Float16( 4.0 ))) == "0 10001 0000000000"
bitstring(Float16( 5.0 ))) == "0 10001 0100000000"
bitstring(Float16( 6.0 ))) == "0 10001 1000000000"
bitstring(Float16( 7.0 ))) == "0 10001 1100000000"
bitstring(Float16( 8.0 ))) == "0 10010 0000000000"
bitstring(Float16( 9.0 ))) == "0 10010 0010000000"
bitstring(Float16(10.0 ))) == "0 10010 0100000000"
```

Machine Numbers

Floating-point types in Julia

Like for integers, Julia provides several floating-point types.

	`Float16`	`Float32`	`Float64`
# exponent bits	5	8	11
# mantissa bits	11	24	53

`Float64` is by far the most commonly used type and the default in Julia.

Example: `typeof(1.0) == Float64`

`Float32` and `Float16` are sometimes used for graphics and machine learning where some accuracy can be traded for better performance.

These `FloatN` types are actually not specific to Julia but are defined in the IEEE Standard for Floating-Point arithmetic (IEEE 754).

IEEE = Institute of Electrical and Electronics Engineers

See `https://en.wikipedia.org/wiki/IEEE_754` for more details.

Julia also provides a `BigFloat` type which has a very large exponent and an arbitrary number of mantissa bits (default `F = 256`).

`BigFloat` is much slower than its `FloatN` counterparts but can sometimes be useful to test your code.

Machine Numbers

Special floating-point values

Floats with $e = 000\ldots000$ and $f = 000\ldots000$ represent ±0.0.

```
bitstring(Float16(  0.0 ))) == "0 00000 0000000000"
bitstring(Float16( -0.0 ))) == "1 00000 0000000000"
```

Floats with $e = 111\ldots111$ and $f = 000\ldots000$ represent ±Inf.

```
bitstring(Float16(  Inf ))) == "0 11111 0000000000"
bitstring(Float16( -Inf ))) == "1 11111 0000000000"
```

±0.0 and ±Inf represent limits and behave correspondingly.

```
1.0/±0.0 == ±Inf    1.0/±Inf == ±0.0
```

Floats with $e = 111\ldots111$ and $f \neq 000\ldots000$ represent ambiguous limits and are called NaN (Not a Number).

```
bitstring(Float16( NaN ))) == "0 11111 1000000000"
```

```
0.0/0.0 == NaN    0.0*Inf == NaN    Inf/Inf == NaN    Inf-Inf == NaN
        0.0*NaN == NaN    1.0*NaN == NaN    Inf*NaN == NaN
                (NaN == NaN) == false (use isnan())
```

Machine Numbers

Remark: Why a special representation for 0?

I assume you are not surprised that `±Inf` and `NaN` have a special floating-point representation; after all, these are clearly very special numbers which require special rules for addition, multiplication and division.

However, you may be surprised that `0` has a special floating-point representation as well; except for division, `0` behaves like any other number (other than `±Inf` and `NaN`), so it may feel wrong that this number is treated specially.

However, `0` is in fact special because in relative terms, it is infinitely far away from any other floating-point number: $\frac{|x-0|}{|0|} = \infty$.

$\frac{c}{0}$ with $c > 0$ is to be interpreted as $\lim_{x \to 0} \frac{c}{x} = \infty$ here.

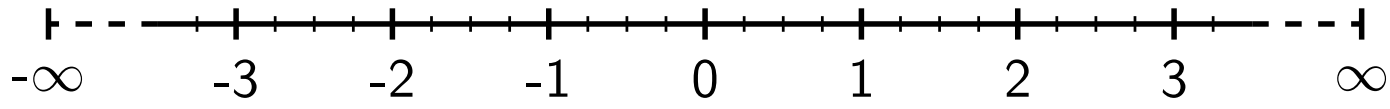
In this sense, `0` is thus indeed more similar to `±Inf` than it is to numbers like `1.0` or `pi = 3.1415...`

Machine Numbers

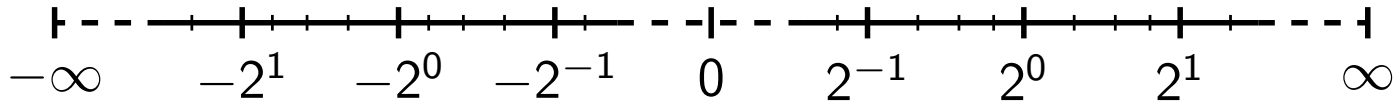
Remark: Pictorial representation of floating-point numbers

The above discussion can be summarised in pictorial form as follows.

Floating-point numbers should not be thought of as



but rather as



Machine Numbers

Properties of floating-point types

Julia provides the following functions for floating-point types `T`.

- `floatmin(T)`: smallest nonzero `T` ($e = 00\ldots01$, $f = 00\ldots00$).
- `floatmax(T)`: largest finite `T` ($e = 11\ldots10$, $f = 11\ldots11$).
- `eps(T)` (machine precision): difference between `1.0` ($f = 00\ldots00$) and the next larger floating-point number ($f = 00\ldots01$).

	`Float16`	`Float32`	`Float64`
`floatmin(T)`	6.1×10^{-5}	1.2×10^{-38}	2.2×10^{-308}
`floatmax(T)`	6.5×10^{4}	3.4×10^{38}	1.8×10^{308}
`eps(T)`	9.8×10^{-4}	1.2×10^{-7}	2.2×10^{-16}

Machine Numbers

Floating-point rounding

We have seen that `IntN(x)` converts an arbitrary number `x` to type `IntN` and throws an error if `IntN` cannot represent `x` exactly.

Mirroring this behaviour for floating-point types would not be practical. For example, 0.1 has infinitely many binary digits, so `FloatN(0.1)` would have to throw an error for any `N`.

Instead, `FloatN(x)` returns the `FloatN` number which is closest to `x`. We refer to this operation as *rounding*.

Thm: Rounding error

Let `T` be a floating-point type. For all $x \in [\texttt{floatmin(T)}, \texttt{floatmax(T)}]$, we have that

$$\texttt{T}(x) = x\,(1+\varepsilon) \quad \text{for some} \quad |\varepsilon| \leq \texttt{eps(T)}/2.$$

Proof. This is essentially a corollary of the spacing theorem from slide 20, but there are some edge cases which would require a more detailed discussion (e.g. for numbers like $x = 1.0$ which sits at the boundary between numbers with exponent $e = -1$ and $e = 0$). I skip these details.

Machine Numbers

Floating-point arithmetic

Observation: Most operations applied to `FloatN`s lead to results which cannot be represented exactly as `FloatN`s.

Examples: $1.1 + 0.11 = 1.21$, $1.1 \times 1.1 = 1.21$

In both cases, we start with two significant digits but end with three significant digits.

Consequence: Floating-point arithmetic cannot be the same as standard arithmetic. I will write $\oplus$, $\ominus$, $\otimes$ and $\oslash$ to distinguish floating-point operations from their standard counterparts.

IEEE 754 mandates that all of these operations must satisfy

$$\boxed{x \circledast y = \mathtt{T}(x * y)}$$

i.e. the result of floating-point arithmetic should be the exact result rounded to the nearest representable number. An analogous rule is also imposed on `sqrt`.

This rigorously defines floating-point arithmetic, and combined with the rounding error theorem it guarantees that

$$\boxed{x \circledast y = (x * y)(1 + \varepsilon) \quad \text{for some} \quad |\varepsilon| \leq \mathtt{eps(T)}/2.}$$

Machine Numbers

Floating-point arithmetic (continued)

Floating-point arithmetic violates many mathematical identities.

- Associativity: With a 2-bit mantissa, we have

```
(-1.0 + 1.0) + 0.01 ==  0.0 + 0.01 == 0.01
-1.0 + (1.0 + 0.01) == -1.0 + 1.01 == 0.0
```

- Distributivity: With a 2-bit mantissa, we have

```
(1.1 - 1.0) * 1.1 == 0.1 * 1.1 == 0.11
1.1 * 1.1 - 1.0 * 1.1 == 10.01 - 1.1 == 0.1
```

- Multiplicative identity: With a 3-bit mantissa, we have

```
1.01 / 1.10 == 0.111
1.01 * (1 / 1.10) == 1.01 * 0.101 = 0.11001
```

Details for first equation: $1.01_2/1.10_2 = 1.25_{10}/1.5_{10} \approx 0.833_{10} \approx 0.111_2$
Details for second equation: $1/1.10_2 = 1/1.5_{10} \approx 0.666_{10} \approx 0.101_2$.

Consequence: The compiler is not allowed to do some seemingly trivial optimisations for us. See `div_performance()` for illustration.

Machine Numbers

Conclusion: Integer vs floating-point arithmetic

Integer arithmetic:

- Good: Exact up to over-/underflow.
- Bad: When overflow happens, it will most likely break your code.

Floating-point arithmetic:

- Bad: Most operations are inaccurate due to rounding.
- Good: Rounding errors are usually harmless.

It seems plausible that Ariane 5 would have been fine if it had used floating-point numbers instead of integers. On the other hand, there are examples where integer arithmetic is more appropriate, see next slide.

Machine Numbers

Conclusion: Integer vs floating-point arithmetic (continued)

Examples where integer / fixed-point arithmetic is more appropriate than floating-point arithmetic:

- If banks used floats, then every transaction would involve a small rounding error. These errors can add up and cost the bank a fortune.
- The American Patriot missile defense system had two clocks, one counting time in integers and one in floats. These clocks diverged over time due to floating-point rounding errors. During the gulf war of 1991, the discrepancy between the clocks led to the system missing an incoming missile which killed 28 US soldiers.

  Source: `http://www-users.math.umn.edu/~arnold/disasters/patriot.html`

Machine Numbers

Conclusion

Most computer scientists and mathematicians assume that machine numbers behave as expected and only start investigating if something goes wrong. This is perfectly acceptable if the stakes are low.

Correspondingly, the main purpose of this lecture is to provide you with the tools needed to investigate if needed, not to make you distrust computers for the rest of your lives.

Machine Numbers

Summary

- Integers: binary representation, two's complement, overflow.
- Floats: binary representation, `eps()`, `floatmin()`, `floatmax()`.
- For both of these topics, it is enough to have some basic understanding. There is no need to e.g. practice binary multiplication for speed. You may ignore rational and fixed-point numbers.
- Rounding theorem: $\mathtt{T}(x) = x\,(1+\varepsilon)$ for some $|\varepsilon| \leq \mathtt{eps(T)}/2$.
- IEEE 754 arithmetic: $x \circledast y = \mathtt{T}(x * y)$ for all $* \in \{+, -, \times, \div\}$, and $\mathtt{sqrt(x)} = \mathtt{T}(\sqrt{x})$.

Recommended exercises

- Have a look at `bitstring(x)` for a couple of integers and floating-point numbers `x` and try to make sense of the output.