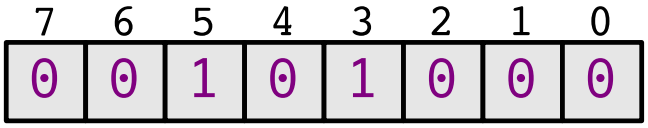
1 Issues of Floating Point Math

Numerical calculations on standard computers can sometimes lead to unexpected results due to complications such as overflow or round-off errors. It is important to be aware of the different things that can go wrong, and to adopt suitable precautions where possible. In this introductory section we recapitulate how standard computer languages handle arithemtic operations in order to get a more presice understanding of these issues.

1.1 Integer arithmetic

- Integers are whole numbers that computers represent in binary.
- The representable range depends on the available storage size of the chosen variable type.
- Negative numbers are represeted as so-called two's complement.

The basic unit is the byte with 8 bits. We can number the bits from 0 (the 'least significant bit', or LSB) to 7 (the most significant bit, MSB). Example:



This number has the value

$$2^5 + 2^3 = 32 + 8 = 40. \tag{1.1}$$

The C-types for single bytes are:

type	range of values
`unsigned char`	$\{0, \ldots, 255\}$
`(signed) char`	$\{-128, -127, \ldots, 0, 1, \ldots, 127\}$

1.1.1 Two's complement for negative numbers:

- The MSB bit flags negative numbers.
- To change the sign of an integer number, one needs two carry out the following steps:
  1. invert all bits
  2. add +1 to the number

Example: Find the binary representation of -9.
start with +9: `00001001`
invert all bits: `11110110`
add +1: `11110111` that's it!
With the two's complement, no special treatment is necessary when adding negative numbers to positive numbers.

1.1.2 Common integer types in C

C-name	bits	range		
`char`	8	-128	...	$+127$
`short`	16	-32768	...	$+32767$
`int`	32	-2147463648	...	$+2147463647$
`long long`	64	-9.2×10^{18}	...	$+9.2 \times 10^{18}$
	n	-2^{n-1}	...	$+2^{n-1}-1$

One can also use unsigned versions of these types (by adding `unsigned` in front), but it is recommened to do this only in exceptional cases because of the danger of nasty surprises in case a negative number is attempted to be stored in such a type.

1.1.3 Things to watch out for in integer arithmetic

The largest danger is **overflow**. Here is an example:

```
char a, b, c;
a = 100;
b = 5;
c = a * b;
```

When one tries to print out the variable c, one gets: -12 !
Why? `a * b = 500` has binary representation

```
....000111110100
```

This will now be truncted to just the 8 least significant bits, because the result is stored in a variable of type char. So one gets:

```
11110100
```

Because the most significant bit is set, this will be interpreted as a negative number. We can apply the two's complement to flip the sign and obtain the absolute value of the number. After the two's complement, we have

```
00001010
```

This corresponds to the value +12.

So when using integer variables, always be aware of their finite range. In the C-language, the type `int` is often sufficient as univeral counting and indexing variable. But increasingly often, its range of about 2 billion may not be enough any more for large simulations. In this case, the use of 64-bit integer variables should be considered. (But using this for everything will cost performance and typically result in a waste of storage space.)

Another issue where some caution is in order is **integer division**. In most languages, this is defined to always truncate all dezimal places, i.e.

```
 8 /  3 =  2

-8 /  3 = -2

 8 / -3 = -2
```

Also watch out for the defintion of the **modulus** operation in your language of choice. In C, the syntax for `8 modulus 3` is `8 % 3`, and the result is defined as `n modulus m = n - (n/m)*m`. Hence:

```
 8 %  3 =  2

-8 %  3 = -2

 8 % -3 =  2
```

Finally, another thing to watch out for are **implicit type conversions**. Suppose we have

```
char a = 85;
char b = 5;
int  c = a * b;
```

When we check the value of c we get the correct result 425, and not -87 that we would get due to overflow if the type of c would also be `char`. This happens because C does an automatic, implicit promotion of a and b to `signed int`, and only then carries out the arithmetic operation. But now contrast this with:

```
       int  a = 2000000000;
       int  b = 3;
long long  c = a * b;
```

When we now check the value of c, we get `1705032704`, which is equal to $6 \times 10^9 - 2^{32}$. Apparently, here our operation has overflown and was not rescued by implicit type conversion, because C will at most use `int` for this. To get the correct result, we have to force a type conversion ourselves, for example in the following form:

```
long long  c = a * ((long long)b);
```

This now yields the correct result of `6000000000`.

1.2 Floating point arithmetic

In general, floating point representations have a

- base β
- precision p (the 'number of digits')
- exponent e

For example, if $\beta = 10$ and $p = 4$, the representations of the two numbers 0.1 and 523 are

number	representation ($\beta = 10$, $p = 4$)
0.1	1.000×10^{-1}
523	5.230×10^{-2}

Nowadays, floating point calculations on computers are most commonly done according to the IEEE-754 standard, using a binary base $\beta = 2$. In particular, single precision numbers use $\beta = 2$ and $p = 23(+1)$. This gives the representations

number	representation ($\beta = 2$)
0.1	$1.10011001100110011001101 \times 2^{-4}$
523	$1.00000101100000000000000 \times 2^{9}$

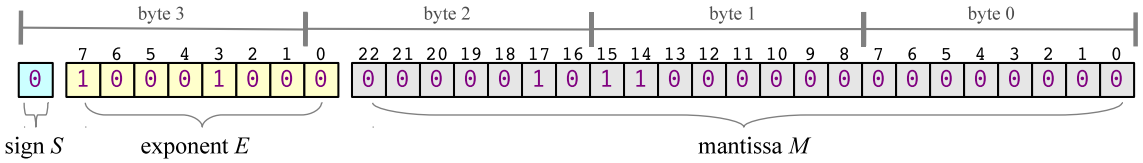
The first encoding has the value $(2^0 + 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + \ldots) \times 2^{-4} \simeq 0.0996$, which differs from 0.1. In fact, 0.1 cannot be represented exactly in the binary format.

Some further notes:

- The representation is not unique – by shifting the exponent, one can arrive at other representations. This is addressed by *normalization*. Every number is stored as convention in the standard such that the leading digit before the dot is 1.

- If this is the case, then the "1" doesn't have to be stored anymore (since it's always there anyway), and one gains a bit of significance if this is exploited. Hence p is really 24 for single precision IEEE-754 numbers. However, representing zero requires a special solution in this case.
- Some numbers cannot be represented exactly – such as the 0.1 from the example.
- The IEEE standard defines a smallest and largest exponent. For single precision numbers this is $e_{\min} = -126$ and $e_{\max} = +127$. This restricts the range of representable numbers, i.e. there is a minimum and a maximum value that is possible, and values beyond this will cause an over- or underflow.

The storage scheme for single-precision IEEE-754 numbers is as follows:



The exponent is stored here in the 8 bits reserved for it not as a two-complement, but rather in a *biased* way, defined through

$$E = e + 127, \tag{1.2}$$

where E is stored as an unsigned byte. Because the range for e is restricted for regular floating point numbers, we have for them $1 \leq E \leq 254$. The values $E = 0$ and $E = 255$ fulfill a special purpose, see below.

The bits are set for the example of encoding the number 523 as a single precision floating point number. In binary representation, 523 can be written in normalized form as

$$1.000001011 \times 2^9 \tag{1.3}$$

The exponent is hence $e = 9$, such that $E = 136$. Because one of the exponent bits is moved to byte 2 in the encoding, byte 3 will then have the value 68. Byte 2 has the value 2, byte 1 the value 192, and byte 0 the value 0.

On little Endian computers (Intel, AMD chips), these bytes will be stored in memory from right to left, with the least significant byte first, i.e. in the sequence 0, 192, 2, 68. On big Endian machines (e.g. PowerPC architecture), the sequence is reversed.

In general, the value f of a IEEE floating point number is

$$f = (-1)^s \cdot \left(1 + \frac{M}{2^p}\right) \times 2^{E-b}, \tag{1.4}$$

where here s is the sign bit, M the integer number representing the mantissa, and E is the exponent. For single precision, we have $p = 23$ and the bias is $b = 127$.

The exponent values $E = 0$ and $E = 255$ are used to represent special values:

1. $E = 0$, $M = 0$: This is zero. Actually, one can have ± 0, depending on the sign bit.

2. $E = 255$, $M = 0$: By convention, this represents $\pm\infty$, depending on the sign bit.

3. $E = 255$, $M \neq 0$: This signals "not a number" (`NaN`), which can result as part of an invalid mathematical operation (division by 0, or square root of a negative number).

4. $E = 0$, $M \neq 0$: These are so-called *denormalized numbers*, because here no leading 1 in front of the dot can be assumed. The value of these numbers is given by

$$f = (-1)^s \cdot \frac{M}{2^p} \times 2^{-b+1} \tag{1.5}$$

Machine precision

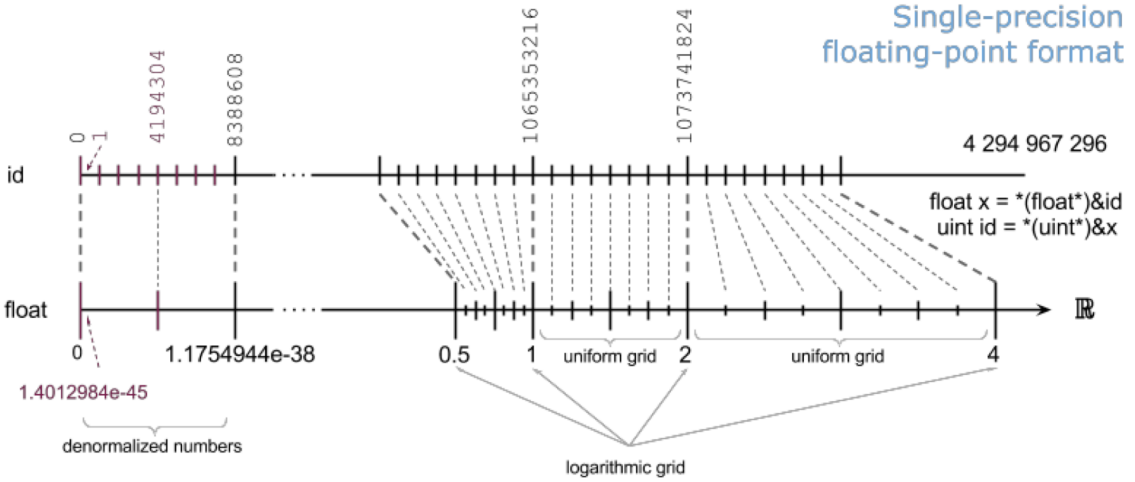
The smallest increment in the mantissa,

$$\epsilon_m = \frac{1}{2^p}. \tag{1.6}$$

is called machine precision. This can be coarsely interpreted as the smallest relative spacing between two floating point numbers that can still be distinguished by the representation scheme. For single precision this is $\epsilon_m \sim 1.19 \times 10^{-7}$.

1.2.1 The set of representable numbers

The set of floating point numbers is finite and subdivides the reals in a special kind of logarithmic grid. (figure below by Denis Yurin)



The largest representable single precision number is

$$f_{\max} = \left(1 + \frac{2^p - 1}{2^p}\right) \times 2^{127} \simeq 3.403 \times 10^{38}, \tag{1.7}$$

while the smallest normalized number is

$$f_{\min} = (1 + 0) \times 2^{-126} \simeq 1.175 \times 10^{-38}. \tag{1.8}$$

But actually, thanks to denormalized numbers, the smallest possible single precision value is

$$f_{\text{smallest}} = \frac{1}{2^p} \times 2^{-126} \simeq 1.4 \times 10^{-45}. \tag{1.9}$$

Due to the introduction of denormalized numbers, IEEE-754 guarantees that for $x \neq y$ one always has $x - y \neq 0$. This is meant to protect, for example, against the common bug-prone idom:

```
if  x != y  then z = 1.0 / (x-y)
```

Notes on floating point arithmetic

- Any arithmetic operation's result is mapped to one of the numbers in the *finite* set of representable floating point numbers. This is called *rounding*.
- The IEEE standard requires that the result of addition, subtraction, multiplication, and division is rounded exactly. This means that the result is as if the calculation was done exactly, and is then rounded to the *nearest* representable number.
- Any operation involving NaN will again yield NaN.

Some consequences and pitfalls

- It is possible that the result of $a + b$ is identical to a for $b \neq 0$. (In fact, this will typically happen when $|b| < \epsilon_m \cdot |a|$.)
- The law of associativity is not guaranteed (due to rounding errors). This means that $(a + b) + c$ does not necessarily evaluate to the same number as $a + (b + c)$.
- Even though $x/2.0$ and $0.5 * x$ are always the same, this is not true for $x/10.0$ and $0.1 * x$. This is because 0.1 is not exactly representable for base $\beta = 2$. Also, beware that optimizing compilers may change $x/10.0$ to $0.1 * x$ (since multiplication is much faster than division), but this can then change the result of the calculation and the semantics of the program.
- When numbers nearly cancel, a loss of precision (reduced number of significant digits) results. For example, consider $x = 10^8$, $y = 10^5$ and $z = -1 - 10^5$. Then:

```
x * y - x * z =  -1.0066 x 10^8

x * (y + z)   =  -1.0 x 10^8
```

Here the second result is correct, but the first one is 0.6% off.

Closer look at loss of precision

Let us more formally define the number of significant digits. Assume $x^\star$ is our floating point approximation to a number x, and likewise $y^\star$ for y. We would then call

$$\frac{x^\star - x}{x} \tag{1.10}$$

the relative error of the representation. One says that $x^\star$ approximates x to r significant digits if

$$|x^\star - x| < \frac{1}{2} 10^{s-r+1}, \tag{1.11}$$

where s is the largest integer such that $10^s < |x|$ (i.e. the absolute error is at most 0.5 in the r-th significant digit of x). For example, $x^\star = 22/7 = 3.1428\ldots$ approximates π to three significant digits.

Cancellation of large numbers typically leads to a loss of significant digits and an explosion of the relative error. Example:

$$x^\star = 0.76545421 \tag{1.12}$$

$$y^\star = 0.76544200 \tag{1.13}$$

Both numbers are stored with 7 significant digits. But the difference

$$z^\star = x^\star - y^\star = 0.12210000 \times 10^{-4} \tag{1.14}$$

has only 3 significant digits in its approximation of z. Consequently, the relative error has become larger by a factor of 1000 or so!

The lesson is, if cancellation can be foreseen *avoid it if possible*. For example, the evaluation of

$$f(x) = 1 - \cos(x) \tag{1.15}$$

involves a cancellation for x near zero. Calculating instead

$$f(x) = \frac{\sin^2(x)}{1 + \cos(x)} \tag{1.16}$$

will yield a more accurate (but potentially more costly) result in this case.

1.2.2 Double and higher precision

Standard complient double precision numbers use $p = 52(+1)$ bits for the mantissa, and 11 bits for the exponent which has the allowed range $\epsilon_{\max} = +1023$ and $\epsilon_{\min} = -1022$. (Btw: $|\epsilon_{\min}| < |\epsilon_{\max}|$ is adopted such that the inverse of the smallest representable doesn't overflow.) The storage footprint of a double is hence 64 bits, or 8 bytes. In total, there are about 4 billion different double precision numbers. The largest and smallest representable numbers are

$$f_{\max} \simeq 1.8 \times 10^{308}, \tag{1.17}$$

$$f_{\min} \simeq 2.2 \times 10^{-308}. \tag{1.18}$$

And the machine precision is

$$\epsilon_m \simeq 2^{-52} = 2.2 \times 10^{-16}. \tag{1.19}$$

- Double precision has all the same principle pitfalls as single precision – but "much less".
- Recommendation: Use always double precision unless memory constraints force the use of single precision for some reason.
- But: Don't believe the use of double precision protects against inaccurate floating point results in all situations!

Quad-double precision

- This is a 128-bit floating point format in the IEEE standard based on a 16 byte presentation.
- It offers twice as many significant digits as plain double precision ($\sim$ 34 dezimal places) and an extended exponent range.
- Unfortunately, this is typically not supported in hardware by current processors, but it can be emulated in software by some compilers, in which case every floating point operation is between 2-10 times slower than a corresponding double precision operation.
- Note that `long double` will be accepted by C-compilers, but in many cases this will either be simply identical to 64-bit double precision values, or it will yield a 96-bit format. Sometimes, the software emulation of 128-bit accuracy can however be enabled through compiler flags.

Arbitrary precision

There are good floating point libraries that can be used for calculations in (nearly) arbitrary, user-defined precision. One very capable package is GMP, the 'GNU big number library' (`https://gmplib.org`).