Interlude: Representing negative integers

In our discussion of IEEE 754, we have seen two ways of storing signed numbers using r bits:

- **sign-magnitude:** One bit is used for the sign, the other bits for the absolute value.

  Typically, the most significant bit is set to 0 for positive numbers and 1 for negative numbers.

  This is what happens in IEEE 754 for the most significant bit.

- **offset binary (or biased representation):** Instead of storing the signed number n, we store $n+b$ with b called the bias.

  Often the bias is chosen to be $b=2^{r-1}$.

  This is what happens in IEEE 754 for the exponent (however, the bias is chosen as $b=2^{r-1}-1$).

(Perhaps) surprisingly, neither of these is how signed integers are most commonly stored:

- **two's complement:** The most significant bit is counted as -2^{r-1} instead of as 2^{r-1}.

  Two's complement is used by nearly all modern CPUs.

  For instance, using 4 bits, the number 3 is stored as 0011, while -3 is stored as 1101. Similarly, 5 is stored as 0101, while -5 is stored as 1011.

  To negate a number n in this representation, we invert all its bits and then add 1.

  As a result, adding a number to its negative produces all ones plus 1 (using r bits in the usual way, all ones corresponds to 2^r-1 so that adding 1 results in 2^r; which is where the name comes from).

  Important. At the level of bits, addition in the two's complement representation works exactly as addition of unsigned integers. For instance, consider the addition $0010+1011=1101$: interpreted as unsigned integers, this is $2+11=13$; alternatively, interpreted as signed integers (using two's complement), this is $2+(-5)=-3$. Likewise, the same is true for multiplication.

  Advanced comment. Two's complement makes particular sense when we interpret it in terms of modular arithmetic (ignore this comment if you are not familiar with this). Namely, if using r bits, all numbers are interpreted as residues modulo 2^r (recall that -1 and 2^r-1 represent the same residue; similarly, 2^{r-1} and -2^{r-1} are the same modulo 2^r). Instead of representing the residues by $0,1,2,\ldots,2^r-1$, we then make the choice to represent them by $0,1,2,\ldots,-3,-2,-1$. Since we can compute with residues as with ordinary numbers, this explains why, using two's complement, addition and multiplication work the same as if the numbers are interpreted as unsigned (assuming that no overflow occurs).

  Comment. Two's complement can also be interpreted as follows: instead of storing the signed number n, we store $n+2^r$, where we only use the r least significant bits (thus throwing away the bit with value 2^r). Apart from this last bit (pun intended!), this is like offset binary.

There are yet further possibilities that are used in practice, most notably:

- **ones' complement:** A positive number n is stored as usual with the most significant bit set to 0, while its negative $-n$ is stored by inverting all bits.

  For instance, using 4 bits, the number 5 is stored as 0101, while -5 is stored as 1010.

  As a result, adding a number to its negative produces all ones (hence the name).

https://en.wikipedia.org/wiki/Signed_number_representations

Example 20. Which of the above four representations has more than one representation of 0?

Solution. In the sign-magnitude as well as in the ones' complement representation, we have a $+0$ and a -0.

Example 21. Express -25 in binary using the two's complement representation with 6 bits.

Solution. Since $25=(011001)_2$, -25 is represented by 100111 (invert all bits, then add 1).

Alternatively, note that $-25=-2^5+7$ and $7=(111)_2$ to arrive at the same representation.

As another alternative, we store $-25+2^6=7$, resulting in the same representation.

Armin Straub
straub@southalabama.edu

Another floating-point issue

Example 22. Explain the following floating-point issue about mixing large and small numbers:

```
>>> 10.**9 + 10.**-9

    1000000000.0

>>> 10.**9 + 10.**-9 == 10.**9

    True

>>> 10.**9

    1000000000.0

>>> 10.**-9

    1e-09
```

Solution. Recall that double precision floats (which is what Python currently uses) use 52 bits for the significand which, together with the initial 1 (which is not stored), means that we are able to store numbers with 53 binary digits of precision. This translates to about $53/\log_2 10 = 15.95... \approx 16$ decimal digits. However, storing $10^9 + 10^{-9}$ exactly requires 19 decimal digits.

[Recall that, if $2^{53} = 10^r$, then $r = \log_{10}(2^{53}) = 53\log_{10}(2) = 53/\log_2 10$.]

Errors: absolute and relative

Suppose that the true value is x and that we approximate it with y.

- The **absolute error** is $|y - x|$.
- The **relative error** is $\left|\frac{y - x}{x}\right|$.

For many applications, the relative error is much more important. Note, for instance, that it does not change if we scale both x and y (in other words, it doesn't change if we change units from, say, meters to millimeters). Speaking of units, note that the relative error is dimensionless (it has no units even if x and y do).

Example 23. There are lots of interesting approximations of π. In each of the following cases, determine both the absolute and the relative error.

(a) $\pi \approx \frac{22}{7}$ $(22/7 \approx 3.14286)$

(b) $\pi \approx \sqrt[4]{9^2 + 19^2/22}$ (This approximation is featured in https://xkcd.com/217/.)

Solution.

(a) The absolute error is $\left|\frac{22}{7} - \pi\right| \approx 0.0013 = 1.3 \cdot 10^{-3}$.

The relative error is $\left|\frac{\frac{22}{7} - \pi}{\pi}\right| \approx 0.00040 = 4.0 \cdot 10^{-4}$.

Comment. Sometimes the relative error is quoted as a "percentage error". Here, this is 0.04%.

(b) The absolute error is $\left|\sqrt[4]{9^2 + 19^2/22} - \pi\right| \approx 1.0 \cdot 10^{-9}$.

The relative error is $\left|\frac{\sqrt[4]{9^2 + 19^2/22}}{\pi}\right| \approx 3.2 \cdot 10^{-10}$.

Example 24. (homework) π^{10} is rounded to the closest integer. Determine both the absolute and the relative error (to three significant digits).

Solution. $\pi^{10} \approx 93{,}648.0475$

The absolute error is $|93{,}648 - \pi^{10}| \approx 0.0475$.

The relative error is $\left|\frac{93{,}648 - \pi^{10}}{\pi^{10}}\right| \approx 5.07 \cdot 10^{-7}$.

Coding in Python: binary digits of 0.1

In the next several examples, we will gradually get more professional in using Python for writing our first serious code.

Example 25. Python Recall that, to express, say, 0.1 in binary, we compute:

- $2 \cdot 0.1 = \boxed{0}.2$
- $2 \cdot 0.2 = \boxed{0}.4$
- $2 \cdot 0.4 = \boxed{0}.8$
- $2 \cdot 0.8 = \boxed{1}.6$
- $2 \cdot 0.6 = \boxed{1}.2$
- and so on...

The above 5 multiplications with 2 reveal 5 digits after the "decimal" point: $0.1 = (0.00011\cdots)_2$. (We can further see that the last four digits repeat; but we will ignore that fact here.)

Let us use Python to do this computation for us. We will start with very basic and naive code, and then upgrade it next time.

```
>>> x = 0.1   # or any value < 1
```

Comment. Everything after the # symbol is considered a comment. This is useful for reminding ourselves of things related to the surrounding code. Comments are usually on a separate line but can be used as above (here, we remind ourselves that the code that follows is not going to handle a number like 2.1 correctly).

To have Python do the above computation for us, we plan to multiply x by 2 (call the result x again), collect a digit (we get that digit as the integer part of x), then subtract that digit from x and repeat. Python has a function called `trunc` which "truncates" a float to its integer part but we need to import it from a package called `math` to make it available.

```
>>> from math import trunc
```

Advanced comment. We can also use `*` in place of `trunc` to import all the functions from the `math` package. However, it is good practice to be explicit about what we need from a package. Note that the function `trunc` is very close to the function `floor` (which computes $\lfloor x \rfloor$, the floor of x, which is the closest integer when rounding down) which also seems appropriate here; however, `floor` returns a float rather than an integer, and we prefer the latter. Also note that we could use `int` (this is a general function that converts an input to an integer) instead of `trunc`. We chose `trunc` because it is more explicitly what we want, and because it gives us a chance to see how to import functions from a package.

We are now ready to compute the first digit:

```
>>> x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
```

```
0
```

By copying-and-pasting these four lines four more times, we can produce the next four digits:

```
>>> x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
    x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
    x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
    x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit

   0
   0
   1
   1
```

Clearly, we should not have to copy-and-paste repeated code like this. We will fix this issue in the next example, as well as discuss several other important improvements.

Example 26. `Python` Let us return to the task of using Python to express, say, 0.1 in binary. Last time, we copied four lines of code 5 times to produce 5 digits. Instead, to repeat something a certain number of times, we should use a **for loop**. For instance:

```
>>> for i in range(3):
        print('Hello')

   Hello
   Hello
   Hello
```

Homework. Replace `print('Hello')` with `print(i)`.

Important comment. The indentation in the second line serves an important purpose in Python. All lines (after the first) that are indented by the same amount will be repeated. Test this by adding a non-indented line containing `print('Bye!')` at the end.

With this in mind, we can upgrade our previous code as follows:

```
>>> x = 0.1  # or any value < 1
    nr_digits = 5  # we want this many digits of x
    from math import trunc
    for i in range(nr_digits):
        x = 2*x
        digit = trunc(x)
        print(digit)
        x = x-digit

   0
   0
   0
   1
   1
```