Example 8. $4.5 = (100.1)_2 = (1.001)_2 \cdot 2^2 = +\underbrace{1.001}_{\text{binary}} \cdot 2^2$

Next, we will see exactly how IEEE 754 would store this as bits (32 bits in the case of "single precision").

IEEE 754 offers several choices but the two most common are:

- **single precision**: 32 bit (1 bit for sign, 23 bit for significand, 8 bit for exponent)
- **double precision**: 64 bit (1 bit for sign, 52 bit for significand, 11 bit for exponent)

A floating-point number: $\pm 1.x \cdot 2^y$ is then stored as follows:

- **Sign:** 1 bit is used for the sign: 0 for $+$ and 1 for $-$.
- **Significand:** Recall that the significand is preceded by an implied bit equal to 1.

In other words, if the significand is $\pm 1.x$ then only the bits for x are stored.

- **Exponent:** In IEEE 754, a constant, called a **bias**, is added to the exponent so that all exponents are positive (this avoids using a sign bit for the exponent).

If the exponent is y, then one stores $y + \text{bias}$ where $\text{bias} = 2^7 - 1 = 127$ for single precision ($\text{bias} = 2^{10} - 1$ for double precision).

Comment. IEEE 754 also offers half precision as well as higher precisions but single and double are the most commonly used because this is what older and current CPUs use. Moreover, the base (also called radix) can also be 10 instead of 2.

Example 9. Represent 4.5 as a single precision floating-point number according to IEEE 754.

Solution. $4.5 = (100.1)_2 = (1.001)_2 \cdot 2^2 = \boxed{+}\underbrace{1.\boxed{001}}_{\text{binary}} \cdot 2^2$

The exponent 2 gets stored as $2 + 127 = \boxed{1000,0001}$.

Overall, 4.5 is stored as $\boxed{0}\,\boxed{1000,0001}\,\boxed{0010,0000,0000,0000,0000,000}$.

Example 10. Represent -0.1 as a single precision floating-point number according to IEEE 754.

Solution. In Example 6, we computed that $0.1 = (0.0001100110011\cdots)_2$.

Hence: $-0.1 = \boxed{-}\underbrace{1.\boxed{1001,1001...}}_{\text{binary}} \cdot 2^{-4}$

The exponent -4 gets stored as $-4 + 127 = \boxed{0111,1011}$.

Overall, -0.1 is stored as $\boxed{1}\,\boxed{0111,1011}\,\boxed{1001,1001,1001,...}$.

Caution. Note that we are not able to store -0.1 exactly. Therefore we need to be careful about how to choose the final bit to best approximate -0.1. According to IEEE 754, the final bit should be 1 (rather than 0 which we would get if we simply truncated) so that -0.1 gets stored as $\boxed{1}\,\boxed{0111,1011}\,\boxed{1001,1001,1001,1001,1001,101}$.

Note that it certainly makes sense to round up the final 0 to 1 because it is followed by $1...$ (this is similar to us rounding up a final 0 in decimal to 1 if it is followed by $5...$).

Example 11. Give an example where one should not use floats.

Solution. One should not use floats when dealing with money. That is because, as we just saw, an amount such as 0.10 dollars cannot be represented exactly using a float (when using base 2, as is the default in most programming languages such as Python) and thus will get rounded. This is problematic when working with money.

Comment. For most purposes, the easiest way to avoid these issues is to store dollar amounts as cents. For the latter we can then simply use integers and work with exact numbers (no rounding).

Armin Straub
straub@southalabama.edu

Example 12. (reasons for floats) Almost universally, major programming languages use floating-point numbers for representing real numbers. Why not fixed-point numbers?

Solution. Fixed-point numbers have some serious issues for scientific computation. Most notably:

- Scaling a number typically results in a loss of precision.
  For instance, dividing a number by 2^r and then multiplying it with 2^r loses r digits of precision (in particular, this means that it is computationally dangerous to change units). Make sure that you see that this does not happen for floating-point numbers.
- The range of numbers is limited.
  For instance, the largest number is on the order of 2^N where N is the number of bits used for the integer part. On the other hand, a floating-point number can be of the order of $2^{2^{M-1}}$ where M is the number of bits used for the exponent. (Make sure you see how enormous of a difference this is! See the previous example for the case of double precision.)

Moreover, as noted in the box below, fixed-point numbers do not really offer anything that isn't already provided by integers. This is the reason why most programming languages don't even offer built-in fixed-point numbers.

Fixed-point numbers are essentially like integers.

For instance, instead of 21.013 (say, seconds) we just work with 21013 (which now is in milliseconds).

Using Python as a basic calculator

Example 13. `Python` We can use Python as a basic calculator. Addition, subtraction, multiplication and division work as we would probably expect:

```
>>> 16*3+1

    49

>>> 3/2

    1.5
```

To compute powers like 2^{64}, we can use ** (two asterisks).

```
>>> 2**64

    18446744073709551616
```

Division with remainder of, say, 49 by 3 results in $49 = 16 \cdot 3 + 1$. In Python, we can use the operators // and % to compute the result of the division as well as the remainder:

```
>>> 49 // 3

    16

>>> 49 % 3

    1
```

% is called the **modulo** operator. For instance, we say that 49 modulo 3 equals 1 (and this is often written as $49 \equiv 1 \pmod 3$).

IMPORTANT. The commands here are entered into an interactive Python interpreter (this is indicated by the >>>). When running a Python script, we need to use `print(16*3+1)` or `print(3/2)` to receive the first two outputs above.

Example 14. `Python` Python automatically uses (double precision) floats when we enter numbers with a decimal point or as the result of divisions.

```
>>> 1.1

    1.1

>>> 1/3

    0.3333333333333333
```

Comment. Note how we can see (roughly) the 52 bit precision of the double precision floats (there are 16 decimal digits after the decimal point, which translates to about $16 \cdot \log_2 10 \approx 53.15$ binary digits; this corresponds to about 52 bit after the first implicit 1).

IMPORTANT. As noted earlier, the commands here are entered into an interactive Python interpreter (this is indicated by the >>>). When running a Python script, we need to use `print(1.1)` or `print(1/3)` to receive the above outputs.

For very large (or very small) numbers, scientific notation is often used:

```
>>> 2.0 * 10**80

    2e+80

>>> (1/2)**100

    7.888609052210118e-31
```

The following are two things that are (somewhat) special to Python and are often handled differently in other programming languages. First, integers are not limited in size (often, integers are limited to 64 bits, which can cause issues like overflow when one exceeds the 2^{64} possibilities). This is illustrated by the following (this explains why we wrote 2.0 above):

```
>>> 2 * 10**80

    200000000000000000000000000000000000000000000000000000000000000000000000000000
```

Second, Python likes to throw errors when a computation runs into an issue (there are nice ways to "catch" these errors in a program and to react accordingly, but that is probably beyond what we will use Python for).

```
>>> 1 / 0

    ZeroDivisionError: division by zero
```

Some other programming languages would instead (silently, without error messages) return special floats representing $+\infty$, $-\infty$ or `NaN` (not-a-number).

Example 15. `Python` Explain the following floating-point rounding issue:

```
>>> 1 + 1 + 1 == 3
```

```
True
```

```
>>> 0.1 + 0.1 + 0.1 == 0.3
```

```
False
```

```
>>> 0.1 + 0.1 + 0.1
```

```
0.30000000000000004
```

Solution. As we saw in Example 6, 0.1 cannot be stored exactly as a floating-point number (when using base 2). Instead, it gets rounded up slightly. After adding three copies of this number, the error has increased to the point that it becomes visible as in the above output.

IMPORTANT. In the Python code above, we used the operator == (two equal signs) to compare two quantities. Note that we cannot use = (single equal sign) because that operator is used for assignment (`x = y` assigns the value of `y` to `x`, whereas `x == y` checks whether `x` and `y` are equal).

Comment. As the above issue shows, we should never test two floats x and y for equality. Instead, one typically tests whether the difference $|x - y|$ is less than a certain appropriate threshold. An alternative practical way is to round the floats before comparison (below, we round to 8 decimal digits):

```
>>> round(0.1 + 0.1 + 0.1, 8) == round(0.3, 8)
```

```
True
```