Floating Point Representation

To introduce floating point representation, we first consider fixed point representation.

When representing integers we assume the decimal (or binary) point is to the right of the rightmost digit. Thus 12 is 12. (the decimal point is to the right of the 12). Using an unsigned 8 bit binary representation, 12 would be represented by 00001100.

If we shift the binary point to the left say by four positions, we could represent fractional values. For example 12.5 would be represented by 1100.1000 (note that digits below the binary point are powers of $\frac{1}{2} = 2^{-1}$). 5.75 would be represented by 0101.1100 (since the fractional part 0.75 equals $2^{-1}$ or $\frac{1}{2}$ plus $2^{-2}$ or $\frac{1}{4}$).

This is called fixed point representation. The binary point is assumed to be fixed between two of the digits or bits. In the above example we reduced the range of numbers that can be represented (the largest integer we can represent is 1111.0000 = 15 decimal) but allow fractions to be represented (0000.0001 = 0.0625 decimal is the smallest fraction that can be represented). The mechanics are not much different from the more familiar binary integer representations we’ve already considered (and neither is the underlying computer hardware used to perform arithmetic and logical operations).

A 6 decimal digit fixed point example:

Consider a six digit fixed point representation where an assumed decimal point is between the 3rd and 4th digit. For example +123.456 would be represented as

$$+123.456$$

where the assumed decimal point (indicated by ^) is between the 3 and 4. The largest number we can represent is 999.999 while the smallest non-zero value is 0.001.

Unfortunately this representation does not work well with very large numbers like Avogadro’s constant $6.025 \times 10^{23}$ or very smaller numbers like Planck’s constant $6.625 \times 10^{-34}$. However, there is a way around this problem

A 6 decimal digit floating point example:

Given a number like $6.025 \times 10^{23}$ take the same 6 digit representation above but reallocate the digits to store the exponent and the fractional part, i.e. 6.025 (also called the significant, the characteristic or the mantissa) in separate fields. The exponent offset by 50 is stored in the first two digits (the exponent field) and the fractional part in the remaining four digits (the fractional field). We store $23 + 50 = 73$ in the first two digits and 6025 in the remaining four.

$$+73.6025$$

Note that the assumed decimal point is still between the 3rd and 4th digit.
The *offset by 50* for the exponent is used to allow the representation of negative exponents. For example Planck’s constant, \(6.625 \times 10^{-34}\) would be represented by

\[
\begin{array}{cccccc}
+ & 1 & 6 & 6 & 6 & 2 & 5 \\
\end{array}
\]

since \(50 - 34 = 16\).

This is how floating point representation works. Note that with this representation we can represent a much larger *range* of numbers – essentially a range from \(10^{-50}\) to \(10^{+50}\) with a *precision* of 4 decimal digits, the size of the fraction.

Also notice that floating point representation is much different from fixed point or integer representation.

It should be noted that representing zero does cause a problem – so we reserve the special representation

\[
\begin{array}{cccccc}
+ & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

(i.e. all 0’s) to represent zero.

**An 8-bit Binary Example:**

Representing binary numbers using floating point representation requires introducing a couple of wrinkles to the method. To illustrate this let’s use as a very small example an 8 bit binary floating point representation. The left most bit is the sign bit (0 for plus, 1 for minus) followed by 3 bits for the exponent (offset by 011 or 3 decimal) followed by 4 bits for the fraction.

```
| |     |       |
+-+-+-+-+-+-+-+-+
7 6 5 4 3 2 1 0
```

5.75 in binary is 101.11 (check it for yourself) which could be expressed as \(101.11 \times 2^0\) or \(10.111 \times 2^1\) or \(1.0111 \times 2^2\) or even \(0.10111 \times 2^3\) depending on how much we wanted to shift the binary point and adjust the exponent accordingly.

**Normalization** is the convention used in floating point representation whereby we adjust shift the binary point and adjust the exponent accordingly so that *either* the binary point is to the immediate *left* of the leading non-zero bit (i.e. \(0.10111 \times 2^3\)) *or* the binary point is to the immediate *right* of the leading non-zero bit (i.e.\(1.0111 \times 2^2\)). Yes it comes in two flavors!

In (binary) floating point representation all numbers are normalized.
Since the IEEE 754 standard for floating point representation use the latter normalization method, this is what we’ll use. Thus \( 5.75 = 1.0111 \times 2^{10} \) is represented as

```
  0 1 0 1 1 0 1 1
```

Note that the exponent 2 or 10 in binary is offset by 011 – hence 101 is stored in the 3 bit exponent field. Unfortunately with only 4 bits for the fraction our five bit value 1.0111 gets truncated to four bits – 1.011

However, there is a way to obtain one more bit of precision

**The Hidden Bit**

Since (IEEE 754 standard) normalization shifts the binary point to the immediate right of the leading non-zero bit, the first bit in the fraction will always be a 1. Since we know this, we don’t have to store this bit in the fraction field. Thus instead of storing 1.011 we store .0111 which allows one extra bit of precision. So \( 5.75 = 1.0111 \times 2^{10} \) is stored as

```
  0 1 0 1 0 1 1 1
```

where we understand that the fractional part, the 0111 actually has a hidden 1 in front of it.

Again note that floating point representation (the actual IEEE 754 floating point standard is a 32-bit and 64-bit standard) is nothing like fixed point or integer representation which is why a computer cannot add, subtract, multiply and/or divide integers and floating point numbers together. It has to convert one representation to the other (usually integer to floating point) before it can perform any arithmetic operation.