Floating Point Arithmetic
A Short Introduction

1. Introduction

We use floating point numbers to represent real numbers using the scientific notation on a computer. In the scientific notation, numbers are written as a normalized number multiplied by a power of 10. An example;

$$12345 = 1.2345 \cdot 10^4.$$

Of course as computers do not have infinite memory, these numbers are usually truncated and can only provide finite precision (i.e. we can not represent π, sad!). A *floating point system* is defined by three components;

- *Base* Base(or radix) of the number system,
- *Mantissa* represents the significant digits of that number,
- *Exponent* is magnitude of the number.

We represent a floating point system by

$$F[b\textit{(Base)}, m\textit{(size of mantissa)}, e\textit{(size of exponent)}],$$

alternatively sometimes it can be represented as,

$$F[b\textit{(Base)}, m\textit{(size of mantissa)}, e_{\min}, e_{\max}],$$

where $e_{\min}$ is *the min. value of exponent can take* and $e_{\max}$ is *the max. value of exponent can take*. Preferred one is the first one.

In computers, base is defined as 2 and it is not stored explicitly. By IEEE standarts a real number is stored in a binary computer in the following form;

`|s|exponent|mantissa|`

where `s` is sign of the number. One thing you will notice is we do not have a sign for exponent even though in real life, we deal with both positive and negative exponents. Because under IEEE standards, exponents are stored in *biased* form. Using n terms in exponent, allows numbers from 0 to $2^n - 1$ to be represented. If we subtract $2^{n-1} - 1$ from each of these numbers, we will get a range of numbers from $-2^{n-1} - 1$ to 2^{n-1}. So by assuming that every number in exponent is 2^{n-1} less than its original value(i.e. *biased*), IEEE removes necessity for the sign bit for exponents. Also if you check representation of 1 in IEEE standard, you will notice that mantissa is all zeros. As values are always normalized, computers do not store first 1 explicitly and just assume it is there. So let's consider an example in $F[2, 23, 8]$ which is the single precision.

Example
$(-69.125)_{10} = -1000101.001_2$
Normalizing $1000101.001_2 = 1.000101001_2 * 2^6$
We have the mantissa now 00010100100000000000000
True exponent is 6, to get biased exponent we should add $2^{8-1} = 128$ to it; $6 + 127 = 133_{10} = 10000101_2$.
Number is negative so the sign bit is 1. Therefore answer is;
`|1|10000101|00010100100000000000000|`

This is equivalent to what has been described in your textbook, but more correct version of how it is implemented in modern computers. Before IEEE standards every computer manufacturer had their ideas about how to do this and it was mess. So praise IEEE!

There are of course some limitations apply. Since e is finite, we can only represent numbers in a range -they cannot be arbitrarily large or small- and since m is finite, numbers cannot be arbitrarily close to each other -there are gaps between them-. Any number that does not satisfy these conditions must be approximated by those satisfy. Before going into further discussion, I should give some definitions.

2. Some Definitions

Let's first consider the range of numbers we can represent. By changing the sign bit, we can represent both positive and negative numbers. So our major concern is which is the smallest number greater than zero and which is the largest number in our floating point system. Take for example single precision $F[2, 23, 8]$;

Minimum value exponent can take is $-2^7 - 1 = -127$, (non-normalized) minimum value in mantissa 2^{-23}, so $x_{\min} = 2^{-150}$. The minimum value is normalized (run realmin in MATLAB, to see IEEE double precision minimum value) and for single precision it is `0|00000001|00000000000000000000000`$= 2^{-126}$.

Maximum value exponent can take is $2^7 = 128$. The maximum value is normalized (run realmax in MATLAB, to see IEEE double precision minimum value) and for single precision it is `0|11111110|11111111111111111111111`$= (1 - 2^{-24})2^{128}$.

One problem we can foresee is either going over maximum value or going under minimum value without changing sign. First case is *overflow* and second is *underflow*. Under IEEE standard if overflow happens, operation will return `Inf`-infinity positive or negative-. If underflow happens, operation will return 0, which may be catastrophic.

It is also important to notice that floating point numbers are not equally spaced. The numbers will be more densely packed close to zero and there will be fewer numbers as we get further from zero. You can experiment and see this for yourself using floatgui.m(google it, or if you have Moler's book it is included there.).

Some things to notice:

- If you choose to represent number on a log scale they seem to be uniformly distributed, this is because as a result of our construction of floating point system, in each interval $2^{e-1} \leq x \leq 2^e$ numbers are equally spaced with increment 2^{e-m}.
- The distance between 1 and the next larger floating point is defined as *machine epsilon* ϵ_{mach} and $\epsilon_{mach} = 2^{1-m}$.
- Depending on the rounding we use, there is a number u which is less than ϵ_{mach} s.t. $1 + u = 1 + \epsilon_{mach}$. It is called *unit round-off*.

If base is different than 2, everything holds except that it will be base b rather 2 in formulas. Sometimes, and for the sake of this course following definition is made; $\epsilon_{mach} = u$.

An important thing with unit round-off is for every number x and its floating point representation `fl(x)`, $\frac{x - fl(x)}{x} \leq u$. Therefore u is sometimes called as max. rounding error.

For modern computers, under the IEEE standards, double precision spans great range of values. This implies that underflow and overflow are not common issues. But, the gaps between numbers can be a serious issue. Little errors we make in representing real numbers can accumulate to the extent that the errors dominate the number to be represented. This can be observed especially with problems with high condition numbers (equivalently unstable algorithms).

3. Floating Point Arithmetic

Up to this point we discussed specifically using IEEE standards. For arithmetic part we will move to a more general framework. Interested reader can find some relevant information on following websites;

- `https://www.doc.ic.ac.uk/~eedwards/compsys/float/`
- `http://www.cs.mcgill.ca/~cs573/fall2002/notes/lec273/lecture3/`
- Wikipedia: single precision, double precision

Let's start with considering two rounding functions `fl(x)`;

- Chopping `fl(x)`=$floor(x)$, ex. `fl(1.2)`=1,`fl(1.7)`=1,
- Rounding `fl(x)`=$round(x)$, ex. `fl(1.2)`=1,`fl(1.7)`=2.

There is an obvious advantage with using rounding rather than chopping as it introduces less amount of error, however chopping is far easier to implement.

Now with this we can define addition, subtraction, multiplication and division.

Definition 1 *For floating point numbers* `x` *and* `y`*, operations are defined as;*

- `x`$\oplus$`y=fl(fl(x)+fl(y))`
- `x`$\ominus$`y=fl(fl(x)-fl(y))`
- `x`$\otimes$`y=fl(fl(x)*fl(y))`
- `x`$\oslash$`y=fl(fl(x)/fl(y))`

Algorithm to make addition is as follows, with example;

Take $F[10, 2, 1]$, and perform $1.2 + 0.1$.
1. Start with normalizing numbers, $1.2 = 0.12 \times 10^1, 0.1 = 0.10 \times 10^0$.
2. Equalize exponents to larger (in absolute sense) number, $1.2 = 0.12 \times 10^1, 0.1 = 0.01 \times 10^1$.
3. Add the mantissas $0.12 + 0.01 = 0.13$.
4. Normalize and check for overflow/underflow $-9 \leq 1 \leq 9$, so no overflow/underflow.
5. Round the sum, in this case unnecessary as sum fits in two decimal places.

Result: $0.13 \times 10^1 = 1.3$, which is correct.

Subtraction is basically adding a positive and a negative number, just it is necessary to pay attention to sign as the result will have the sign of larger number in absolute value.

Similarly we can give an algorithm for multiplication. Take $F[2, 3, 2]$ and perform 2×0.5.

Let's start with converting the numbers into base 2 and normalizing them, $2_{10} = 10_2 = 0.100 \times 2^{10_2}$ and $0.5_{10} = 0.1_2 = 0.100 \times 2^{-00_2}$
1. Add exponents $10_2 + (-00_2) = 10_2$.
2. Multiply mantissas, in this case it is easy $0.100 \times 0.100 = 0.010$.
3. Normalize and check for overflow/underflow, $0.010 \times 2^{10_2} = 0.100 \times 2^{01_2}$ and $-3 \leq 1 \leq 3$, so neither.
4. round the sum, unnecessary for this case.

Result: $1.00_2 \times 2^{00_2} = 1.00_2 = 1_{10}$.

As the last algorithm let's consider division, before going into more complicated examples. Take $F[3, 2, 1]$ and perform $3/(1/3)$.

Let's start with converting the numbers into base 3 and normalizing them, $3_{10} = 10_3 = 0.10 \times 3^{2_3}$ and $(1/3)_{10} = 0.10_3 = 0.10 \times 3^{-0_3}$.
1. Subtract exponents $2_3 - (-0_3) = 2_3$.
2. Divide mantissas, in this case it is easy $0.10 \div 0.10 = 1.00$.
3. Normalize and check for overflow/underflow, $1.0 \times 3^{2_3} = 0.10 \times 3^{10_3}$ and $3 \geq 2$, so overflows.
4. Round the sum, unnecessary for this case as we have overflow already.

Result: overflow, not good.

4. Examples

For all examples, we will use floating point system $F[4, 3, 2]$ and everything will be written in base 4, except base itself.
Example, Addition with Overflow Add 0.213×4^{33} to 0.213×4^{33}.
1. Normalize the numbers; they are already normal.
2. Equalize the exponents the larger number; they are already equal.
3. Add the mantissas;

```
  0.213
+ 0.213
-------
  1.032
```

4. Normalize and check for overflow/underflow; 0.1032×4^{100} and $100 > 33$, so there is overflow.
5. Round the sum; unnecessary as we have overflow already.

Example, Multiplication Multiply 0.213×4^{10} by 0.213×4^{13}.
1. Normalize the numbers; they are already normal.
2. Add exponents; $10 + 13 = 23$.
3. Multiply mantissas;

```
      0.213
    x 0.213
----------
     1311
    0213
+   1032
----------
  0.113301
```

4. Normalize and check for overflow/underflow; number is normal already and $-33 \leq 23 \leq 33$ so neither.
5. Round the sum; we have 3 places in mantissa and fourth value is 3 and it is greater than round threshold ($base/2 = 4/2 = 2$) so we will round it up: 0.120×4^{23}
Result: 0.120×4^{23}

Example, Division Divide 0.120×4^{23} by 0.213×4^{13}.
1. Normalize the numbers; they are already normal.
2. Subtract exponents; $23 - 13 = 10$.
3. Divide mantissas;

```
        0.2131
      -----------
0.213 ) 0.120
      - 0.1032
      -----------
        0.0102
      - 0.00213
      -----------
        0.00201
      - 0.001311
      -----------
        0.000033
      - 0.0000213
      -----------
        0.0000111
```

We could continue but quotient has already more than allowed mantissa and rest of the operation we will make will not matter as they will be rounded off.
4. Normalize and check for overflow/underflow; number is normal already and $-33 \leq 10 \leq 33$ so neither.
5. Round the sum; we have 3 places in mantissa and fourth value is 1 and it is less than round threshold ($base/2 = 4/2 = 2$) so we will round it down: 0.213×4^{10}
Result: 0.213×4^{10}

5. Some Exercises

Take any system except decimal system as your floating point system, choose size of mantissa and exponent.
Exercise 1 Represent -0.75_{10} in floating point format.
Exercise 2 Represent 4.9_{10} in floating point format.
Exercise 3 Multiply 4.9_{10} and -0.75_{10} in floating point format.
Exercise 4 Divide 4.9_{10} by -0.75_{10} in floating point format.
Exercise 5 Divide -0.75_{10} by 4.9_{10} in floating point format.
Exercise 6 Multiply results of exercises 4 and 5, compare to the expected result.
Exercise 7 Create two numbers x, y in your floating point system, such that $x \oplus y = x$.
Exercise 8 Google "loss of significance" and create an example in floating point system.
Exercise 8 Create two numbers x, y in your floating point system, such that $x \oslash y = 0$ but $x/y \neq 0$.
Exercise 9 Create two numbers x, y in your floating point system, such that $x \otimes y =$`Inf` but $x \times y < \infty$.