Floating Point Arithmetic

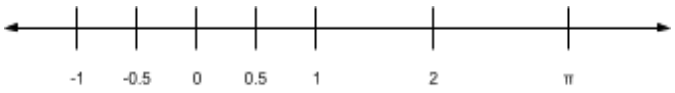
Introduction

Floating point is a number system used for pretty much all scientific computing. All modern CPUs/GPUs provide hardware accelerated support for floating point math, which ultimately ends up getting used in things like physics simulations, real-time computer graphics, artificial neural networks, etc…

These notes are mainly based off of the book Numerical Computing with IEEE Floating Point Arithmetic by Michael L. Overton (ISBN 978-0-8987-1571-2), Handbook of Floating Point Arithmetic (ISBN 978-0-8176-4704-9), Wikipedia, and also several other sources.

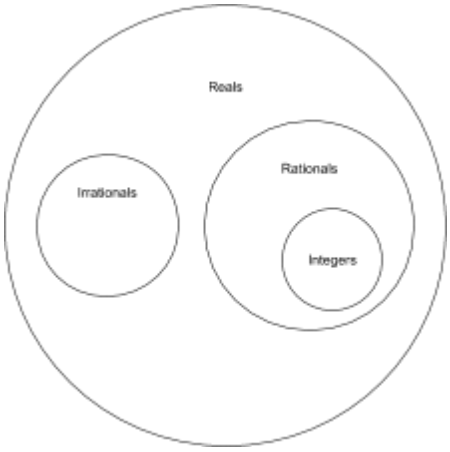
Real Numbers

Real numbers can be represented as points on a line. The line stretches infinitely in both directions and every point on the line corresponds to a real number (infinite number of points). The line diagram below shows an example line with just a few points listed.



> **NOTE**: According to the book: "∞(infinity) and -∞ (negative infinity) are not numbers in the conventional sense but are included in the extended real numbers."

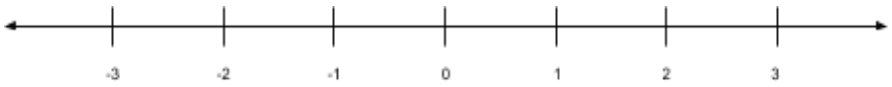
Real numbers are made up of integers, rational numbers (includes integers), and irrational numbers. The subsections below detail each.



Integer Numbers

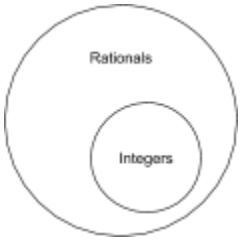
Integers are whole numbers, meaning that they don't contain a fractional portion. For example: 1,5,-10 are all integers, but -3.333 is not.

There are an infinite but countable number of integers. All this means is that even though there is no start/end limit (infinite), every integer would appear in the list given that we counted long enough. There are an infinite number of integers so in practice you will never be able to count them all, but they are countable nonetheless.



Rational Numbers

Rational numbers are numbers that can be expressed as a ratio (fraction) of two integers. For example: $\frac{1}{2}, \frac{3}{5}, \frac{10}{5}$ are rational numbers.



> **NOTE**: The set of rational numbers contains all integers. For example, $\frac{10}{5}$ evaluates to the integer 2.

NOTE: Even if the numerator and/or denominator in your fraction are rational numbers (instead of integers), remember that you can still rewrite them as integers and have them evaluate to the same result. For example, $\frac{1.2}{1}$ is equivalent to $\frac{12}{10}$.

Similar to integers, there are an infinite but countable number of rationals. There is no start/end limit to the rationals (infinite), and every rational number will appear at least once given that we counted long enough.

	1	2	3	4	...
0	±0/1	±0/2	±0/3	±0/4	...
1	±1/1	±1/2	±1/3	±1/4	...
2	±2/1	±2/2	±2/3	±2/4	...
3	±3/1	±3/2	±3/3	±3/4	...
...	...	...	...	...	...

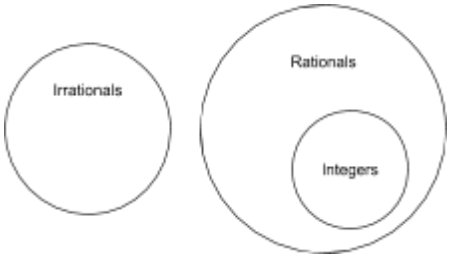
Unlike with integers, the way to visualize counting of rational numbers is to create a table with numerators (integer) and denominators (integer > 0). Note how a rational number can appear more than once. For example…

- ±0/1, ±0/2, ±0/3 ±0/4 all evaluate to 0
- ±1/1, ±2/2, ±3/3 all evaluate to ±1
- ±1/2, ±2/4 both evaluate to ±0.5

This is what was meant in the statement: every rational number will appear at least once given that we counted long enough. When counting, a rational number can appear more than once. You can get it unique representation by reducing it to its lowest form (canceling any common factors in the numerator and denominator): for example, $\frac{6}{9}$ has a factor of 3... $\frac{6}{9}=\frac{2\cdot3}{3\cdot3}=\frac{2}{3}$=0.6667

Irrational Numbers

Irrational numbers are real numbers that are not rational. That means that you cannot express irrational numbers as a fraction.



> **NOTE**: Remember that if you can't express it as a fraction (not rational), then it isn't an integer either. All integers can be expressed as fractions. For example, $\frac{10}{5}$ evaluates to the integer 2.

Common examples of irrational numbers are $\sqrt{2}$ and π.

According to the book: "Every irrational number can be defined as a limit of a sequence of rational numbers, but there is no way of listing all irrational numbers. The set of irrational numbers is said to be uncountable."

Scientific Notation

Scientific notation is just a way of writing really large or really small numbers in the format…
$\pm S * 10^{E}$

For example, the following numbers have be re-written in scientific notation…
$800 = 8*10^{2}$
$7100 = 71*10^{2}$
$-720000 = -7.2*10^{5}$
$0.000072 = 7.2*10^{-5}$

> **NOTE**: Remember that $10^{-5} = \frac{1}{10^5}$, so $7.2 \cdot 10^{-5} = 7.2 \cdot \frac{1}{10^5} = \frac{7.2}{100000}$.

In normalized scientific notation, the requirement $1 \leq S < 10$ is added. $1 \leq S < 10$ just means that the whole number portion of scientific notation has to be a single digit that's between 1 and 9 (in other words, a single digit that isn't 0). So for example…
$71*10^{2}$ <-- NOT normalized (whole number portion is 2 digits)
$0.1*10^{2}$ <-- NOT normalized (whole number portion is 0)
$7.1*10^{2}$ <-- normalized (whole number portion is 1 digit)

> **NOTE**: Having trouble thinking about why $1 \leq S < 10$ limits the whole number portion to a single digit? Write out some random numbers in scientific notation. Does $1 \leq S < 10$ evaluate to true when any of the numbers have more than 1 digit in the whole number portion?

The vocabulary for scientific notation (±S * 10^E) is:

- ± is the sign.
- S is the significand / mantissa (note S doesn't include the sign -- the sign is separate).
- E is the exponent.
- ±S * 10^E where $1 \leq S < 10$ is called the normalized representation.
    - normalization is the process of obtaining a normalized representation.

The multiplication by 10^E just shifts the decimal point either right or left by E. To the right if E is positive, to the left if E is negative. Given that $1 \leq S < 10$ (normalized representation), "we can imagine that the decimal point floats to the position immediately after the first non-zero digit in the expanded form of the number -- hence the name floating point." (quote from the book)

So, for example…

$0.00000125 = 1.25*10^{-6}$

You can visualize the decimal point floating to just after the first non-zero digit (1 in our example)...

$0.00000125 = 0.00000125*10^{-0}$
$0.00000125 = 0.0000125*10^{-1}$
$0.00000125 = 0.000125*10^{-2}$
$0.00000125 = 0.00125*10^{-3}$
$0.00000125 = 0.0125*10^{-4}$
$0.00000125 = 0.125*10^{-5}$
$0.00000125 = 1.25*10^{-6}$

Exponent Rules

> **NOTE**: Sources are http://www.mesacc.edu/~scotz47781/mat120/notes/exponents/review/review.html and https://www.rapidtables.com/math/number/exponent.html.

There are 5 exponent rules…

1. Zero-exponent rule: $x^0 = 1$
   $5^0 = 1$
   $(x^{-1} \cdot y^{16} - 1)^0 = 1$
2. Power rule: $(x^m)^n = x^{m \cdot n}$
   $(x^3)^2 = x^{3 \cdot 2} = x^6$
   $(3x^3)^2 = (3^1 \cdot x^3)^2 = 9x^6$
   $(x^3 \cdot y^2)^2 = x^6 \cdot y^4$ // confused? expand it out by hand.
   $\sqrt{x^4} = (x^4)^{\frac{1}{2}} = x^{(4 \cdot \frac{1}{2})} = x^2$ // confused? sqrt(x) is x^(1/2)
   $\sqrt[3]{x^6} = (x^6)^{\frac{1}{3}} = x^{(6 \cdot \frac{1}{3})} = x^2$ // confused? cuberoot(x) is x^(1/3)

3. Negative exponent rule: $x^{-m} = \frac{1}{x^m}$
   $x^{-3} = \frac{1}{x^3}$
4. Product rule: $x^m \cdot x^n = x^{m+n}$
   $x^3 \cdot x^2 = x^{3+2}$
5. Quotient rule: $\frac{x^m}{x^n} = x^{m-n}$
   $\frac{x^3}{x^2} = x^{3-2}$

When in doubt, apply rules in the order they appear above. For example…
$(\frac{x^3 \cdot y^2}{y^{-2} \cdot x^0})^3$
$(\frac{x^3 \cdot y^2}{y^{-2} * x^0})^3 = (\frac{x^3 \cdot y^2}{y^{-2} \cdot 1})^3 = (\frac{x^3 \cdot y^2}{y^{-2}})^3$ ← applied rule 1 (zero-exp)
$(\frac{x^3 \cdot y^2}{y^{-2}})^3 = \frac{x^9 \cdot y^6}{y^{-6}}$ ← applied rule 2 (power)
$\frac{x^9 \cdot y^6}{y^{-6}} = \frac{x^9 \cdot y^6}{1} \cdot \frac{1}{y^{-6}} = x^9 \cdot y^6 \cdot y^6$ ← applied rule 3 (neg-exp)
$x^9 \cdot y^6 \cdot y^6 = x^9 \cdot y^{12}$ ← applied rule 4 (prod)
$x^9 \cdot y^{12}$

Binary (Base2)

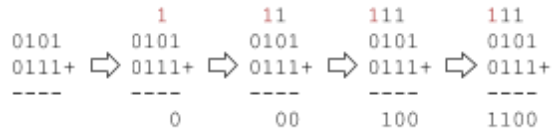
Binary numbers are numbers in base 2. Normal numbers are called decimal numbers, and they're in base 10. Base 2 is used in computing because it more closely aligns to the internals of computing hardware.

Binary (base 2) is pretty much the same as decimal (base 10), except that each digit/symbol in the number is ranges 0-1 instead of 0-9…

Base10	Base2
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

10	1010
11	1011
12	1100
...	...

Most if not all math operations are the same: add, subtract, multiply, etc. For example, you can add base 2 numbers the same way you would base 10 numbers, so long as you make sure that rollover happens after 1 instead of after 9…



Binary numbers are typically identified by either a b somewhere or a subscript of 2. For example…

- 0b1011
- 1011b
- $(1011)_2$

  NOTE: The last example (subscript) can be used for any base. For example, base 16… $(a0ff)_{16}$.

Common terminology for binary numbers:

- bit -- a binary digit, can represent either 0 (off) or 1 (on)
- bitstring -- a string of bits

Common boundaries for binary numbers:

- byte -- 8 bits, can represent 2^8 (256) unique values
- word -- 32 bits (4 bytes), can represent 2^{32} unique values
- double word -- 64 bits (8 bytes), can represent 2^{64} unique values
- kilobyte -- 1024 (2^{10}) bytes
- megabyte -- 1024 (2^{20}) kilobytes
- gigabyte -- 1024 (2^{30}) gigabytes
- terabyte -- 1024 (2^{40}) terabytes
- petabyte -- 1024 (2^{50}) petabytes

  NOTE: Word and double word definitions are old (the book is from 2001). They may no longer apply or are applied less frequently. Rather than saying word people more often now say 4 byte boundary?

  NOTE: Actual arithmetic is typical done on word and double word boundaries (sometimes byte as well). Everything else is used for strings and data storage.

> **NOTE**: kilo, mega, giga, tera, and peta amounts above are approximations of their true value…
> kilo = 10^3
> mega = 10^6
> giga = 10^{10}
> tera = 10^{12}
> peta = 10^{15}

Binary Representation of Real Numbers

Real number can be represented (expanded) in any base (binary, octal, decimal, etc..).

> **NOTE**: Expansion is just another way of saying representation. You're "expanding" by positioning/shifting each digit/symbol in the number. There's a whole blurb in the book about positional number systems but it isn't worth going over. It is important to know it when deriving the algorithm for how transition back-and-forth between binary and decimal.

For integers, expansion is simple…

$$(451)_{10} = (4 \cdot 100) + (5 \cdot 10) + (1)$$
$$(10101)_2 = (1 \cdot 10000) + (0 \cdot 1000) + (1 \cdot 100) + (0 \cdot 10) + (1)$$

For non-integers (non integrals?), the fractional portion may be tricky. A fractional portion that is non-terminating on one base (e.g. base2) may be terminating in another base (e.g. base10). For example…

An expansion that terminates in both decimal (base10) and binary (base2)...
$\frac{11}{4} = (2.75)_{10} = (2) + (7 \cdot \frac{1}{10}) + (5 \cdot \frac{1}{100})$
$\frac{11}{4} = (10.11)_2 = (1 \cdot 10) + (0) + (1 \cdot \frac{1}{10}) + (1 \cdot \frac{1}{100})$

An expansion that terminates in decimal (base10) but not binary (base2)...
$\frac{1}{10} = (0.1)_{10} = (0) + (1 \cdot \frac{1}{10})$
$\frac{1}{10} = (0.000110011\ldots)_2 = (0) + (0 \cdot \frac{1}{10}) + (0 \cdot \frac{1}{100}) + (0 \cdot \frac{1}{1000}) + (1 \cdot \frac{1}{10000}) + (1 \cdot \frac{1}{100000}) + (0 \cdot \frac{1}{1000000}) + (0 \cdot \frac{1}{10000000}) + (1 \cdot \frac{1}{100000000}) + (1 \cdot \frac{1}{1000000000}) + \ldots$

The key takeaways are …

- A fractional portion that is non-terminating on one base (e.g. base2) may be terminating in another base (e.g. base10).

- All rational numbers will either have a terminating (finite) expansion or non-terminating repeating expansion. If it's non-terminating, it will repeat.
- All irrational numbers will have non-terminating expansions that won't repeat.

Binary to Decimal Conversion

For the integer portion, you expand each binary digit/symbol to get its decimal representation, then sum it all up. For example, to convert the bit string 010111 to decimal (base10), begin by expanding each binary digit/symbol…

0b → 0b * 100000b → $0 * 2^5$ → 0

1b → 1b * 10000b → $1 * 2^4$ → 16

0b → 0b * 1000b → $0 * 2^3$ → 0

1b → 1b * 100b → $1 * 2^2$ → 4

1b → 1b * 10b → $1 * 2^1$ → 2

1b → 1b * 1b → $1 * 2^0$ → 1

Then, add up the expanded numbers 1+2+4+16=23.

The fractional portion is done similarly, expand it out and sum it up. For example, to convert the fractional portion of 0.0101 to decimal (base10), begin by expanding it…

1b → 1b * (1b/10000b) → $1 * (1/2^4)$ → 1/16

0b → 0b * (1b/1000b) → $0 * (1/2^3)$ → 0

1b → 1b * (1b/100b) → $1 * (1/2^2)$ → 1/4

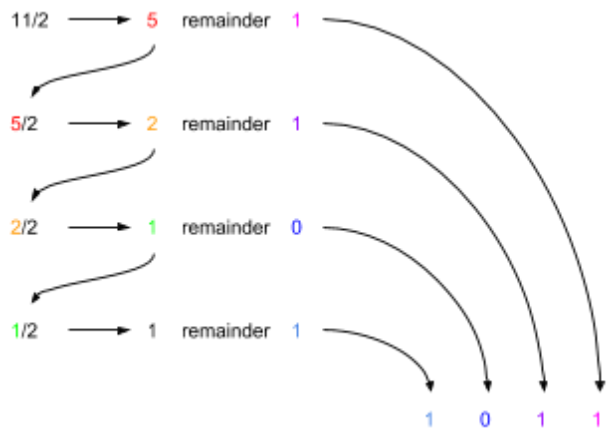
0b → 0b * (1b/10b) → $0 * (1/2^1)$ → 0

Then, add up the expanded numbers $\frac{1}{4} + \frac{1}{16} = \frac{4}{16} + \frac{1}{16} = \frac{5}{16} = 0.3125$

> **NOTE**: It's trivial to convert this process to any other base.

Decimal to Binary Conversion

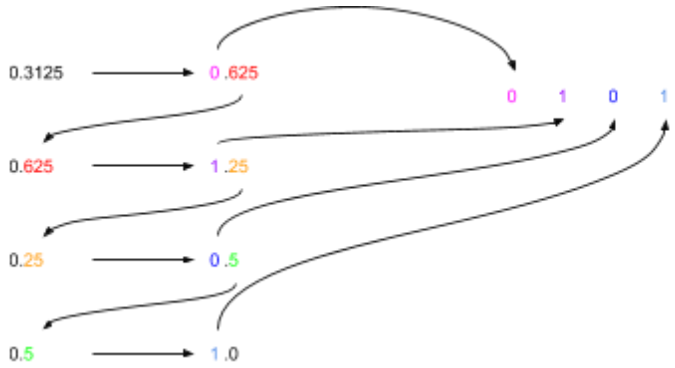
For the integer portion, you have to iteratively divide the output by 2. At each division, the remainder is the binary digit/symbol. For example, to convert the number 11 to binary…



In the example diagram above, it shows how $(11)_{10} = (1011)_2$

> **NOTE**: If you want to see why this works, convert the numerator and denominator both to base 2 and do longhand division. For example. instead of 11/2 do 1011/10. It boils down to you just popping bits off the tailend.

The fractional portion is done similarly. Instead of iteratively dividing the output by 2, you iteratively multiply it by 2. The integer portion is the bit while the fractional portion becomes the new numerator. For example, to convert the fractional portion of 0.3125 to binary…



In the example diagram above, it shows how $(0.3125)_{10} = (0.0101)_2$

> **NOTE**: The section of the book this is fram has a whole blurb about positional number systems and their expansion/representation. This blurb is what gives you the “hints” to derive the conversion algorithms detailed above. The algorithms aren’t given to you outright.

Other Useful Bases

Often times, instead of working directly in base2, programmers will chose to work in base8 (octal) or base16 (hexadecimal). So long as the base can be represented as 2^b, each digit/symbol of that base can be represented as a bitstring of size b.

For example, the digits/symbols for base8 can be viewed as grouping of 3 bits ($2^3 = 8$)…

Base8	Base2
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Non-Floating Point Number Systems

Sign-and-Magnitude (Integer)

Sign-and-magnitude simply means that the first bit of the bitstring (most significant bit) represents the sign of the number. The remaining bits are used as the magnitude. For example, for a 3-bit integer space…

```
0 = 000    100 = -0
1 = 001    101 = -1
2 = 010    110 = -2
3 = 011    111 = -3
```

There's at least 1 edgecase here: there are 2 representations of 0s. Hardware implementations of basic math functions (e.g. add) need to take this into account. Programmers also need to watch out when using equality/relational operators. For example, will 0 == -0? or will 0 > -0?

One's Complement (Integer)

One's complement is a integer number system where the negative numbers are a bitwise negation of the positive numbers. For example, for a 3 bit integer space…

```
0 = 000    111 = -0
1 = 001    110 = -1
2 = 010    101 = -2
3 = 011    100 = -3
```

Just like sign-and-magnitude, there's an issue with there being 2 representations of 0. But, unlike sign-and-magnitude, certain math operations become much easier to implement in hardware (as if you were doing them by hand with normal decimal/base10 numbers). For example, addition becomes simpler…

```
(2)   010
(-2)  101 +
      ---
(-0)  111
```

One's complement requires that, if an overflow occurs during addition, that overflow bit must be added back to the right-most bit (least significant bit). This is called an "end-around carry"...

```
(2)   010
(-1)  110 +
      ---
     1000 --> 000   // perform the "end-around carry"
              001 +
              ---
           (1) 001

(-1)  110
(-2)  101 +
      ---
     1011 --> 011   // perform the "end-around carry"
              001 +
              ---
          (-3) 100
```

> **NOTE**: Check out the Wikipedia page for One's complement for more information. There are other things to watch out for.

Two's Complement (Integer)

Two's complement is the integer number system used by most hardware today. Unlike one's complement, it doesn't have redundant zero value (-0) and the operations of addition/subtraction/multiplication (maybe others?) are performed exactly the same as they would be for unsigned integers.

For example, for a 3 bit integer space…

```
0 = 000
```

```
1 = 001     111 = -1
2 = 010     110 = -2
3 = 011     101 = -3
            100 = -4
```

> **NOTE**: The range of negatives are shifted down by 1. Two's complement doesn't have a negative 0 value like sign-and-magnitude/one's complement. You can abstract the range out to for two's complement as $[-2^{(b-1)}, 2^{(b-1)} - 1]$.

Like the introductory paragraph says, the hardware implementations for add/subtract/multiply is identical to the unsigned version. There's no "end-around carry/borrow" phase like there is in one's complement -- the overflow just gets discarded.

Addition...

```
(2)    010
(-2)   110 +
       ---
(0)   1000 <-- overflow bit gets discarded so answer is correct: 000
```

Subtraction...

```
(-2)   110
(1)    001 -
       ---
(-3)   101
```

> **NOTE**: If you're too retarded to remember how to subtract by hand, go to https://www.youtube.com/watch?v=gcAAZvFe_X8.

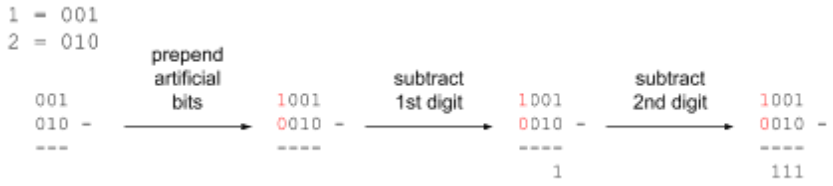
Negation (changing the sign of the number)...

```
(1)    001
(-2)   NOT(001) = 110
(-1)   110 + 1 = 111
```

> **NOTE**: Negation works both ways. Doesn't matter if you're starting with negative or positive. It seems to work as long as the number being negated is not 0 and not the minimum (-4 in the 3 bit integer space we're using).

The book mentions that no special hardware is needed for two's complement subtraction. So long as you can negate a number, you can use add… e.g. x-y = x+(-y). The book walks through corner cases with overflows and stuff such it that ultimately prove that it all works the way it should (pages 10-11).

Actually trying to come up with logic to perform a subtraction will end up being more convoluted than simply negating + adding. For example, try doing 1-2 vs 1+(-2)...



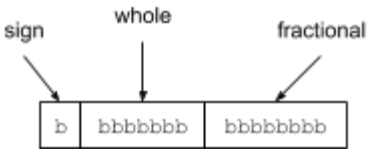
vs

> **NOTE**: I don't know if the subtraction logic I came up with is entirely correct. There are likely other cases for subtraction that need handling.

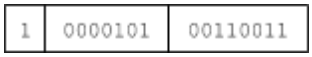
Both results produce -1 (111 in two's complement), but the addition doesn't require any special cases. However, subtraction still requires handling of special cases (probably?).

Fixed Point (Real)

Fixed point breaks up a number into 3 fields: sign, whole number portion, fractional number portion. For example, a 16-bit bitstring may use 1 bit for the sign, 7 bits for the whole number, and 8 bits for the fractional portion…



For example, -5.1 would come out to…



> **NOTE**: If you don't understand why the fractional portion comes out the way it does, go to the Binary Representation of Real Numbers section of this document.

> **NOTE**: The book also mentions "symbolic numbers", meaning that your number is actually represented by 2 numbers: numerator and denominator. The book says that this format is more accurate but becomes "inconvenient" when it comes to arithmetic. AFAIK I've never seen a platform use this.

NOTE: The book mentions that the range supported by fixed point numbers make them inadequate for scientific applications. However, I remember using fixed point for some things on J2ME CLDC1.0 (did not support floats). It was piggybacking off of two's complement integer arithmetic. As long as never bleed into the negatives, basic add/subtract worked. I think multiply and divide was workable also (I can't remember). I remember using fixed point to make a progress bar. Things like sqrt and log could have probably been approximated using lookup tables.

NOTE: Later on the book mentions a fixed point system called block floating point, where a group of fixed point numbers have an exponent. This system seems to have also failed in favour of normal floating point.

Floating Point Basics

Floating point numbers are based on normalized scientific notation. Recall that normalized representation scientific notation is defined as $\pm S * 10^E$ where $1 \leq S < 10$.

This is pretty much how floating point numbers are represented in a computer, except that instead of decimal (base10) they use binary (base2): $\pm S * 2^E$ where $1 \leq S < 2$. So for example, the binary number -10000.1101 would first be normalized…

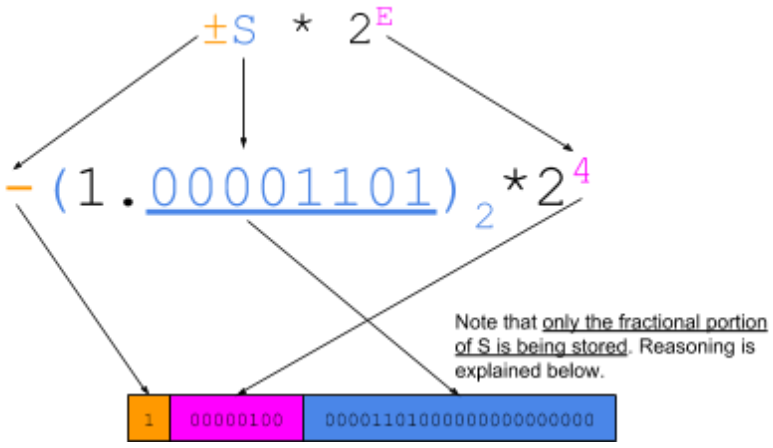
$-(10000.1101)_2 = -(10000.1101)_2 * 2^0$
$-(10000.1101)_2 = -(1000.01101)_2 * 2^1$
$-(10000.1101)_2 = -(100.001101)_2 * 2^2$
$-(10000.1101)_2 = -(10.0001101)_2 * 2^3$
$-(10000.1101)_2 = -(1.00001101)_2 * 2^4$

NOTE: Remember that the S (significand/mantissa) is in base2. The full base10 representation here is $-1.05078125 * 2^4$.

NOTE: $1 \leq S < 2$ is decimal (base10) for $1_2 \leq S < 10_2$ in binary (base2). All it means is that the whole number portion of S must contain a single binary digit that isn't 0: 1. Just like how in decimal (base 10), the condition $1 \leq S < 10$ says S must contain contain a single decimal digit that isn't 0: 1,2,3,4,5,6,7,8,9.

Then, each part/variable of this normalized representation goes into a specific region of some larger bitstring. If we were to fit our example in a 32-bit word, we could organize it such that…

- ± (sign) goes into bit 0
- E (exponent) goes into bits 1-8
- S (significand/mantissa) goes into bits 9-31 (fractional portion only -- explained later)



> **NOTE**: Remember that E (exponent) can be negative independent of the ± (sign). For example, $-(1.1)_2*2^{-2}$ has both a negative E and a ± set to negative. As such, E would have to be represented in some integer format that supports sign (e.g. two's complement, but the book says in practice it's something other than two's complement).

For S (significand/mantissa), only the fractional portion is stored because the whole number portion will always be 1. It will never be any number greater or smaller. If it were, it would no longer be a valid normalized representation: $\pm S * 2^E$ where $1 \leq S < 2$.

For example, none of these are valid normalized representations…

- $(10.1101)_2*2^2$ (or if S converted to base10, $2.8125*2^2$)
- $(0.1)_2*2^1$ (or if S converted to base10, $0.5*2^2$)
- $(11.1)_2*2^7$ (or if S converted to base10, $3.5*2^2$)

because they don't meet the condition required for normalized representation: $1 \leq S < 2$. All that condition says is that the whole number portion of S can only be 1 binary digit that isn't 0. If it can't be 0, the only other binary digit that leaves is 1. Since it can only ever be 1 (in other words, it will always be 1), we don't actually need to store it. This is called a hidden bit.

The field that stores S is sometimes called the fractional field. When referring to the field, it's necessary to automatically account for any hidden bit: it may not be included in the field but it's implied. In other words, given a fractional field, it's necessary to imagine that the symbol '1.' appears in front of it even though it isn't stored.

> **NOTE**: Since the symbol '1.' always appears in front of whatever is stored, we won't ever be able to represent the number 0. Workaround are explained in a subsequent section.

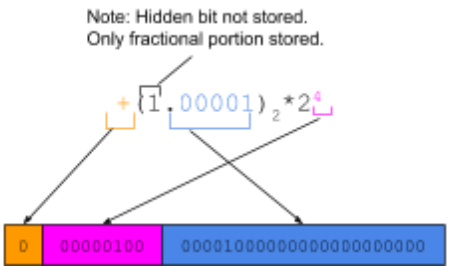
> **NOTE**: It's important to note that, depending on how the number is normalized, you can end up with more than 1 hidden bits. See Exercise 3.11 for details.

Truncation

> **NOTE**: Be aware that truncation and rounding aren't the same thing. They both approximate a real number such that it can be stored in the underlying floating point format, but rounding is more desired/reliable way of approximation. It's detailed later on.

If a real number can be represented EXACTLY (in whatever bit layout that's used by that platform), it's called a floating point number. So for example, 16.5 is a floating point number because it can be stored in the 32-bit example format (detailed in the parent section)…

$$
\begin{aligned}
(16.5)_{10} &= (10000.1)_2 * 2^0 \\
&= (1000.01)_2 * 2^1 \\
&= (100.001)_2 * 2^2 \\
&= (10.0001)_2 * 2^3 \\
&= (1.00001)_2 * 2^4
\end{aligned}
$$



If a real number can't be represented exactly, it needs to be truncated. Examples of numbers that first need to be truncated before being stored in that 32-bit example format...

$$
\begin{aligned}
(1.99999998509883880615234375)_{10} &= (1.11111111111111111111111)_2 \\
(16.1)_{10} &= (10000.000110011...)_2
\end{aligned}
$$

The first number (1.999999985...) has too many fractional digits to be represented in our example 32-bit format: 26 vs a max of 23. The second (16.1) is non-terminating in binary, so it can't be represented by any format no matter how wide the fractional portion of the storage format is.

Truncating the fractional portion of a number in way to make it fit into storage. Always make sure to normalize before you truncate. If you don't, you may end up with more bits being truncated than is necessary. For example, converting the number 0.2 to binary results in a non-terminating fractional portion…

$$(0.2)_{10} = (0.00110011001100110011001100110011...)_2$$

If we first truncate the fractional portion to 23 bits….

$$(0.00110011001100110011001)_2$$

and then normalize it for storage…

$$+(0.00110011001100110011001)_2 = +(0.00110011001100110011001)_2 * 2^0$$
$$= +(0.01100110011001100110010)_2 * 2^{-1}$$
$$= +(0.11001100110011001100100)_2 * 2^{-2}$$
$$= +(1.10011001100110011001000)_2 * 2^{-3}$$

we end up only using 20 bits of the fractional field instead of 23 bits, which is more of a loss in precision than is required...

$(1.10011001100110011001000)_2$

Had we normalized first...

$+(0.00110011001100110011001100110011...)_2 * 2^0$
$+(0.01100110011001100110011001100110...)_2 * 2^{-1}$
$+(0.11001100110011001100110011001100...)_2 * 2^{-2}$
$+(1.10011001100110011001100110011001...)_2 * 2^{-3}$

and then truncated...

$+(1.10011001100110011001100)_2 * 2^{-3}$

those 3 digits have been set to their correct value…

$(1.10011001100110011001100)_2$

> **NOTE**: Truncating the fractional portion of a number is specified as “terminating the expansion of a number” in the book.

Precision

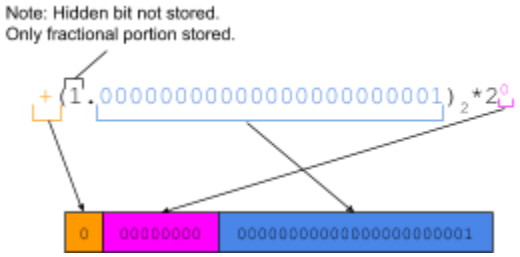
The precision (denoted as p) of a floating point format is the number of bits in the significand/mantissa, including any hidden bits. The 32-bit example format (detailed in the parent section) has precision of 24: 23 bits in the fractional field and 1 hidden bit…

$$p = 1 + 23 = 24$$

> **NOTE**: Be aware that you can have formats with more than 1 hidden bit: See Exercise 3.11 for more info.

Machine Epsilon

The gap between the number 1 and the first fully representable number greater than 1 is called machine epsilon (denoted as e). For the 32-bit example format (detailed in the parent section), the smallest fully representable number greater than 1 is…



$$e = (1)_2 - (1.00000000000000000000001)_2$$
$$e = (0.00000000000000000000001)_2$$
$$e = 2^{-(p-1)}$$
$$e = 2^{-23}$$

> **NOTE**: Unsure what p is? Read the section on precision.
>
> **NOTE**: It seems that machine epsilon doesn't have a standard definition. A footnote in the book says that other books treat machine epsilon as HALF the gap. ~~It isn't even made clear what machine epsilon is for. Wikipedia says that it's the "relative error" for any rounding that happens: https://en.wikipedia.org/wiki/Machine_epsilon.~~ Machine epsilon is used for calculations in subsequent sections.
>
> Judging from the table on the Wikipedia page, the actual formula for machine epsilon is $e = b^{-(p-h)}$, where b is the base and p is the precision and h is the number of hidden bits (there can be more than 1 hidden bit see Exercise 3.11)????

Unit in the Last Place (ulp)

The spacing between floating point numbers is called unit in the last place (denoted as ulp), and is calculated using $ulp(x) = e \cdot 2^E$ where…

- e is the machine epsilon for the floating point format.
- x is a number that's fully representable in the floating point format.
- E is the exponent of x once it has been normalized.

That is, for some number x that can be fully represented in the storage layout (without any kind of truncation/rounding), ulp(x) will calculate the gap to the next fully representable number. When…

- $x > 0$, ulp(x) is the gap between x and the next larger fully representable number.
- $x < 0$, ulp(x) is the gap between x and the next smaller fully representable number.

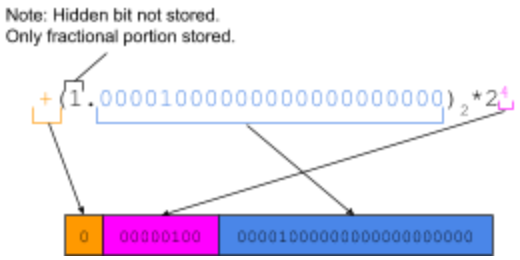
NOTE: According to Wikipedia, ulp is also sometimes called unit of least precision.

For example, to get the next fully representable number after 16.5 using the 32-bit example format (detailed in the parent section), begin by normalizing it…

$(16.5)_{10} = (10000.1)_2$

$$
\begin{aligned}
+(10000.1)_2 &= +(10000.1)_2 * 2^0 \\
&= +(1000.01)_2 * 2^1 \\
&= +(100.001)_2 * 2^2 \\
&= +(10.0001)_2 * 2^3 \\
&= +(1.00001)_2 * 2^4
\end{aligned}
$$

Notice how the normalized representation is fully representable in the 32-bit example format. It can be fully stored without any truncation/rounding…



Then, plug in E (exponent) from the normalization process above and e (machine epsilon) from the machine epsilon calculations above into the ulp algorithm…

$e=2^{-23}$ <-- machine epsilon for example 32-bit number system
$E=4$ <-- exponent from normalization
$ulp(x) = e * 2^E$

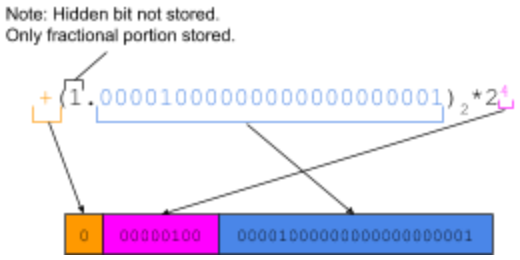
$$
\begin{aligned}
ulp(16.5) &= 2^{-23} * 2^4 \\
ulp(16.5) &= 2^{-23} * 2^4 \\
ulp(16.5) &= 2^{((-23)+4)} \quad \text{// confused? look up exponent rules} \\
ulp(16.5) &= 2^{-19}
\end{aligned}
$$

The gap to the next larger representable floating point value is 2^{-19}. That means that the next floating point value fully representable in the 32-bit example format is $16.5 + 2^{-19}$...

// ADD NUMBERS
$(16.5)_{10} = (10000.1)_2$
$(2^{-19})_{10} = (0.00000100000000000000001)_2$

10000.10000100000000000000

```
00000.00000000000000000001 +
-------------------------
10000.10000000000000000001

// NORMALIZE RESULT
+(10000.10000000000000000001)2 = +(10000.10000000000000000001)2*2^0
                               = +(1000.01000000000000000001)2*2^1
                               = +(100.00100000000000000001)2*2^2
                               = +(10.00010000000000000001)2*2^3
                               = +(1.00001000000000000001)2*2^4

// SHOW THAT RESULT FULLY REPRESENTABLE IN 32-BIT EXAMPLE FORMAT
```



> **NOTE**: If you made this -16.5 instead of 16.5, you should get the same gap: 2^{-19}. But, since -16.5<0, you'll need to subtract it. It'll end up giving you the next SMALLER value instead of the next LARGER value. Refer here.

Using ulp to calculate the gap between numbers helps identify an important property of floating point numbers: as the magnitude gets bigger so do the gaps between numbers. For example, Imagine a number format where $-1 \leq E \leq 1$ and S has 3-bits. If you were to start from the smallest possible number: $\pm(1.000)_2 * 2^{-1}$ and use ulp to continually move forward, plotting the number at each step, this would be the result...

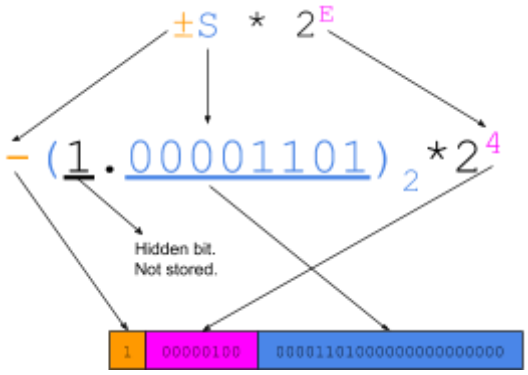
In the above format, there are...

- 8 fully representable numbers in the range [0.5, 1] -- gap every 0.0625
- 8 fully representable numbers in the range [1, 2] -- gap every 0.125
- 8 fully representable numbers in the range [2, 4] -- gap every 0.25
- ...

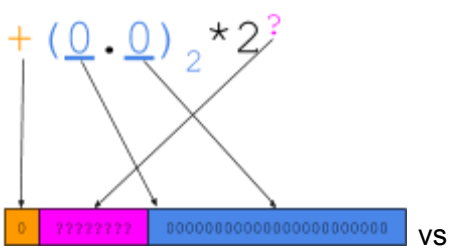
> **NOTE**: The number 0 is not representable in this format, which is why it's marked in red.

Zero

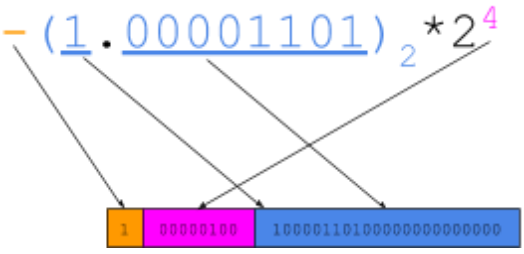
A normalized number cannot represent 0: $\pm S * 2^E$ where $1 \leq S < 2$. The problem is that the whole number portion of the normalized number will always be 1, and as such it isn't actually stored (it's a hidden bit)...



The first way to solve the problem is to explicitly store the leading bit of the S (significand/mantissa) -- forget about the idea of a hidden bit. This way, you can store the number 0 by setting all bits in S to 0, but you also lose 1 bit of precision in S. For example, given the 32-bit example format detailed in the parent section, storing zero vs storing a non-zero...



vs



The second way to solve the problem is to have a magic value/number for E (exponent) that signifies that the number is 0. This ends up reducing the number of possible exponents by 1. For example, given the 32-bit example format detailed in the parent section, storing zero means E=-127...

Both of these approaches open up the possibility of having a negative 0 value.

> **NOTE**: The book mentions that it discusses a way of handling this later on in the book.

> **NOTE**: The first way (explicitly store the hidden bit in S) was used up until around 1975. The second way (magic value for E) is the way IEEE-754 does it.

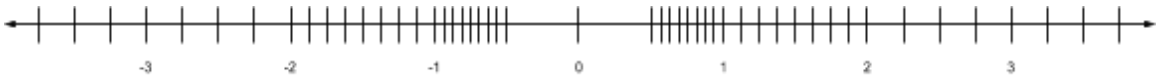
Subnormals

Imagine a format where…

- ± (sign) is 1-bit where 0 means positive
- E (exponent) is 2-bits in two's complement
- S (significand/mantissa) is 3-bits with a leading 1
- 0 (zero) is represented by E=$(10)_2$ and S=$(000)_2$

> **NOTE**: So, given that E is a 2-bit two's complement number, the range of E is $-2 \leq E \leq 1$. But, remember that we're using E=-2 as a special case for representing 0 (-2 in two's complement is 10_2), so for representing normalized numbers the range is actually $-1 \leq E \leq 1$.

If we were to plot out all representable numbers in this format, this would be the result...



Notice the large gaps between 0 and the first non-zero numbers. This gap can be filled by taking advantage of the special E bitstring used to represent 0: E=$(10)_2$. When…

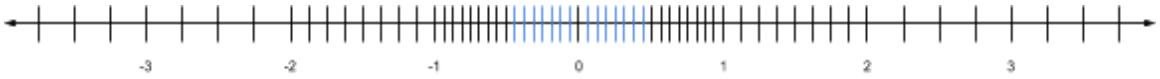
- E=$(10)_2$ and S=$(000)_2$, it represents the number 0.
- E=$(10)_2$ and S is non-zero, the number is undefined.

These undefined numbers (where E=$(10)_2$ and S is non-zero) can be used to fill in the gaps. Specifically, they're called subnormals and are defined as $\pm(0.SSSS...)_2 * 2^{min(E)}$.

For example, if we were to calculate the subnormals for the format discussed above and add them to the plot…

```
min(E) = -1  // minimum value E can be for a normalized number
             //   -1 in our case
subnormals = ±(0.SSS)₂*2^min(E)=±(0.SSS)*2^-1

±=0 E=10 S=111 --> +(0.111)₂*2^-1 = +(0.0111)₂ = +0.4375
±=0 E=10 S=110 --> +(0.110)₂*2^-1 = +(0.0110)₂ = +0.375
±=0 E=10 S=101 --> +(0.101)₂*2^-1 = +(0.0101)₂ = +0.3125
±=0 E=10 S=100 --> +(0.100)₂*2^-1 = +(0.0100)₂ = +0.25
±=0 E=10 S=011 --> +(0.011)₂*2^-1 = +(0.0011)₂ = +0.1875
±=0 E=10 S=010 --> +(0.010)₂*2^-1 = +(0.0010)₂ = +0.125
±=0 E=10 S=001 --> +(0.001)₂*2^-1 = +(0.0001)₂ = +0.0625
±=0 E=10 S=000 --> +(0.000)₂*2^-1 = +(0.0000)₂ = +0.0
±=1 E=10 S=000 --> +(0.000)₂*2^-1 = +(0.0000)₂ = -0.0
±=1 E=10 S=001 --> -(0.001)₂*2^-1 = -(0.0001)₂ = -0.0625
```

±=1 E=10 S=010 --> $-(0.010)_2 * 2^{-1} = -(0.0010)_2 = -0.125$
±=1 E=10 S=011 --> $-(0.011)_2 * 2^{-1} = -(0.0011)_2 = -0.1875$
±=1 E=10 S=100 --> $-(0.100)_2 * 2^{-1} = -(0.0100)_2 = -0.25$
±=1 E=10 S=101 --> $+(0.101)_2 * 2^{-1} = +(0.0101)_2 = -0.3125$
±=1 E=10 S=110 --> $+(0.110)_2 * 2^{-1} = +(0.0110)_2 = -0.375$
±=1 E=10 S=111 --> $+(0.111)_2 * 2^{-1} = +(0.0111)_2 = -0.4375$



As the new plot shows, 7 new numbers have been added to each side of the gap. These numbers are not normalized and cannot be normalized (hence the name subnormals). For a number to normalized it needs to be representable in the form $\pm S * 2^E$ where $1 \leq S < 2$ -- none of these numbers can be represented in that form without exceeding the bounds of either E or S.

Notice the algorithm here. To get a number into subnormal format, start in normalized form and move the dot to the left until E=min(E). For example, the number $+(1.100)_2 * 2^{-3}$ is in normalized form, but E goes below min(E) so of the format detailed above (-3 is below -1). We can represent this number in subnormal form by moving the dot to the left until E=min(E)…

$+(1.100)_2 * 2^{-3}$
$+(0.110)_2 * 2^{-2}$
$+(0.011)_2 * 2^{-1}$

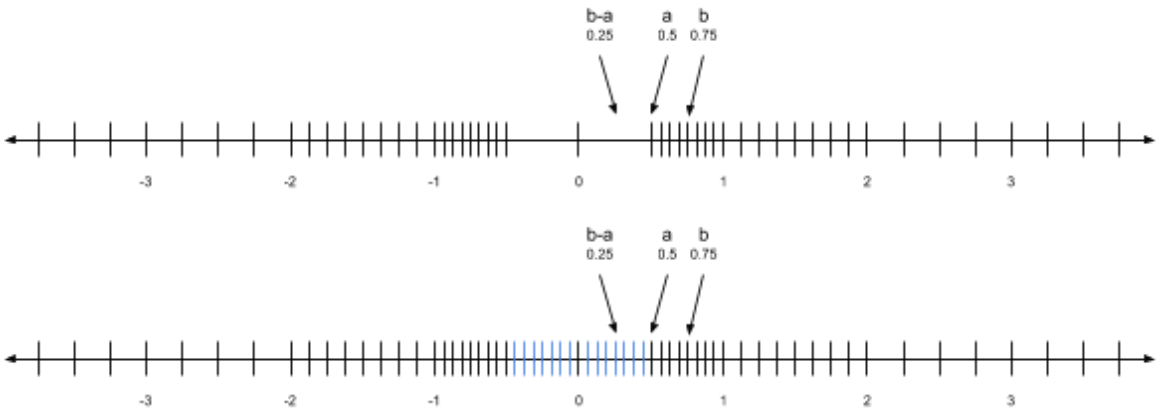
Then, just copy over the fractional portion of S while setting E to the bitstring that that signals that the number is a subnormal…

±=0
$E=(10)_2$ ← special bitstring used for zero and subnormals
$S=(0.011)_2$

> **NOTE**: Confused? Try it yourself. Take the first number in the list: $+(0.0111)_2$. Are you able to get that number intos a normalized representation while keeping within the limits of the format: $-1 \leq E \leq 1$ and S limited to 3-bits (hidden leading bit of 1)?

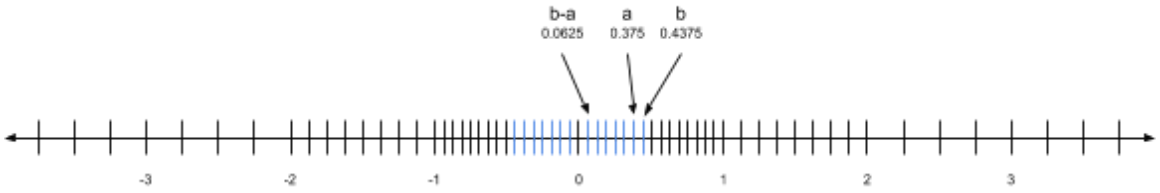
Aside from the extra precision, one of the benefits of subnormal numbers is that they prevent abrupt underflow. That is, the underflow that occurs is more gradual/graceful (gradual underflow). For example, as long as a and b are fully representable in the number format (no truncation/rounding required) and $a \neq b$, $b - a \neq 0$.

To illustrate this, imagine subtracting a=0.5 and b=0.75 without subnormals vs with subnormals…



The result of b-a is 0.25. In the first diagram, 0.25 gets truncated to 0. In the second diagram, there is a representation for 0.25.

The same thing applies if a and b are subnormal. For example, if a=0.375 and b=0.4375, b-a=0.0625…



> **NOTE**: The book mentions that gradual/graceful underflow "significantly eases the writing of stable numerical software." You've seen this term (numerical stability) used once before with GANNs: in order to make GANN training work properly, the algorithms were shifted around to prevent diving into obscenely small values that couldn't be represented in normal IEEE-754 format (I think?) -- aka prevent underflow?

The downside of subnormal numbers is that they're difficult to implement efficiently in hardware. On some platforms, subnormals are implemented in software rather than hardware, which can lead to long execution times.

> **NOTE**: The book states that subnormal numbers were the most controversial part of IEEE-754-1985.

> **NOTE**: Previous versions of the IEEE-754 called this denormals instead of subnormals, I guess because they aren't technically normalized numbers / can't be represented in normalized form (without exceeding the number of bits for each component). Later revisions change the term to subnormals because denormals was too confusing?

Arithmetic: Add/Subtract

Basic addition and subtraction of floating point numbers are done by…

1. adjusting the exponents (E) of the numbers to match.
2. performing the add/subtract on the significand (S).

For example, imagine adding the following 2 floating point numbers together…

$x=+(1.001)_2*2^6$

$y=+(1.000)_2*2^8$

Align the significands (S) by adjusting the numbers so they have the same exponent (E)...

$x=+(1.001)_2*2^6$

$y=+(0.010)_2*2^6$ <-- adjusted y to match x's exponent

Add the significands together to get the final result...

$+(1.001)_2*2^6$

$+(0.010)_2*2^6$ +

\-----

$+(1.011)_2*2^6$

To see why this works, you need to expand out the operands. Once expand, the added digits are all 0. When the expanded digits get added, they result in all 0s. As such, you only really need to add the significands together -- the exponent stays the same in the result...

$+(1.001)_2*2^6$ --> $(1001000)_2$

$+(0.010)_2*2^6$ + --> $(0010000)_2$ +

\----- -------

$+(1.011)_2*2^6$ --> $(1011000)_2$

> **NOTE**: The sign may change and the number may need to be renormalized and potentially rounded to fit.

Arithmetic: Multiply/Divide

Basic multiplication and division are done by multiplying/dividing the significands (S) and adding/subtracting exponents (E) together to produce the final number. For example, imagine multiplying the following 2 floating point numbers together…

$x=+(1.001)_2*2^1$

$y=+(1.000)_2*2^3$

Add the exponents (E)...

$1+3=4$

Multiply the significand (S)...

$+(1.001)_2$ <-- significand (S) of x

$+(1.000)_2$ <-- significand (S) of y

\-----

$+(1.001)_2$ <-- x*y (significands only)

Combine to get the final number…

$+(1.001)_2 * 2^4$ <-- x*y

To see why this works, you need to do some basic algebra…

$x = S * 2^E$

$y = T * 2^F$

For multiplication...

$(S \cdot 2^E) \cdot (T \cdot 2^F)$

$S \cdot 2^E \cdot T \cdot 2^F$ ← remove brackets -- commutative

$S \cdot T \cdot 2^E \cdot 2^F$ ← reorder operands -- associative

$S \cdot T \cdot 2^{E+F}$ ← exponent rule

For division...

$\frac{S \cdot 2^E}{T \cdot 2^F}$

$\frac{S}{T} \cdot \frac{2^E}{2^F}$ ← break up fraction

$\frac{S}{T} \cdot 2^{E-F}$ ← exponent rule

> **NOTE**: The sign may change and the number may need to be renormalized and potentially rounded to fit.

IEEE-754 Floating Point

IEEE-754 standardizes floating point numbers across platforms such that there's…

- consistent representation (e.g. number of bits used for each component).
- consistent handling of exceptional situations (e.g. divide by zero).
- different rounding modes for floating point ops (e.g. trunc, round to floor, etc..)

> **NOTE**: In addition to add/subtract and multiply/divide operation, the IEEE-754 standard also defines the operations remainder and square root (and even more with IEEE-754-2008 update). All of these operations are said to be "correctly rounded" -- the book dedicates an entire section to this topic, but I skipped it here.

Representations

There are multiple representations available in the IEEE-754-1985 standard…

- single format (32-bits)
- double format (64-bits)
- extended format (unspecified number of bits)

> **NOTE**: The word "format" and "precision" are often used interchangeably in this context (e.g. single format vs single precision). I tried to stick with "format" but may have

forgotten in a few places. Either way, the book seems to emphasise that precision should refer to the precision (p) of the whatever representation you're working with. For example, "single precision" should refer to the precision (p) of the single format (p=24 -- layout of single format detailed in a subsection below). Other sources (especially Wikipedia) don't follow this.

Single and double format representations are exposed to the user, while extended format is used internally to minimizing errors such as rounding errors. All 3 of these representations work similarly and provide the same set of features, including...

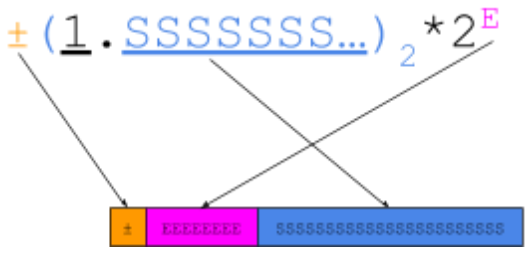
- representation for infinity and negative infinity (e.g. divide by 0)
- representation for "not a number"/NaN (e.g. sqrt of a negative is imaginary)
- representation for zero
- subnormal numbers

NOTE: IEEE-754-2008 adds several new representations (probably all extensions of the existing representations but wider)...
16-bit (called half/binary16)
128-bit (called quadruple/binary128)
256-bit (called octuple/binary256)
32-bit but in decimal (called decimal32)
64-bit but in decimal (called decimal64)
128-bit but in decimal (called decimal128)

Single Format

Single format is a required part of the IEEE-754 standard -- it must be implemented on the platform. It's a 32-bit word broken down as…

- 1-bit for sign (±)
- 8-bits for exponent (E)
- 23-bits for significand/mantissa (S) with 1 hidden leading bit



For numbers in normalized form, E isn't encoded in any of the standard ways to represent an integer (e.g. two's complement, sign-and-magnitude, etc..). Instead, it's encoded in binary as `E+127` and has the range of $-126 \leq E \leq 127$.

NOTE: There's a name for this type of encoding. It's called biased representation. The number being added to E (127) is called the exponent bias. The rationale for doing this is explained on the wikipedia page for this: "Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder." -- https://en.wikipedia.org/wiki/Exponent_bias.

For special numbers, E is set to a custom bitstring. Specifically, when…

- $E=(00000000)_2$ and $S=(000…000)_2$, it represents ±0 (zero).
- $E=(00000000)_2$ and $S!=(000…000)_2$, it represents a subnormal number.
- $E=(11111111)_2$ and $S=(000…000)_2$, it represents ±∞ (infinity).
- $E=(11111111)_2$ and $S!=(000…000)_2$, it represents NaN (not a number).

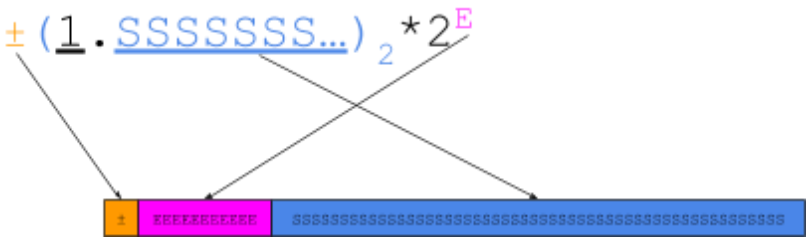
The following is a map of E strings to numeric representations...

$E=(00000000)_2=(0)_{10}$ → $±(0.SSSS…)_2*2^{-126}$ (zero and subnormals)
$E=(00000001)_2=(1)_{10}$ → $±(1.SSSS…)_2*2^{-126}$
$E=(00000010)_2=(2)_{10}$ → $±(1.SSSS…)_2*2^{-125}$
$E=(00000011)_2=(3)_{10}$ → $±(1.SSSS…)_2*2^{-124}$
…
$E=(01111111)_2=(127)_{10}$ → $±(1.SSSS…)_2*2^{0}$
$E=(10000000)_2=(128)_{10}$ → $±(1.SSSS…)_2*2^{1}$
…
$E=(11111100)_2=(252)_{10}$ → $±(1.SSSS…)_2*2^{125}$
$E=(11111101)_2=(253)_{10}$ → $±(1.SSSS…)_2*2^{126}$
$E=(11111110)_2=(254)_{10}$ → $±(1.SSSS…)_2*2^{127}$
$E=(11111111)_2=(255)_{10}$ → ±∞ if $S=(000…000)_2$, otherwise NaN

Double Format

Double format is not a required part of the IEEE-754 standard (it's optional), but pretty much every platform implements it. It's a 64-bit word broken down as…

- 1-bit for sign (±)
- 11-bits for exponent (E)
- 52-bits for significand/mantissa (S) with 1 hidden leading bit



For numbers in normalized form, E isn't encoded in any of the standard ways to represent an integer (e.g. two's complement, sign-and-magnitude, etc..). Instead, it's encoded in binary as `E+1023` and has the range of $-1022 \leq E \leq 1023$.

> **NOTE**: There's a name for this type of encoding. It's called biased representation. The number being added to E (1023) is called the exponent bias. The rationale for doing this is explained on the wikipedia page for this: "Biasing is done because exponents have to be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison harder." -- https://en.wikipedia.org/wiki/Exponent_bias.

For special numbers, E is set to a custom bitstring. Specifically, when…

- $E=(00000000000)_2$ and $S=(000\ldots000)_2$, it represents ±0 (zero).
- $E=(00000000000)_2$ and $S!=(000\ldots000)_2$, it represents a subnormal number.
- $E=(11111111111)_2$ and $S=(000\ldots000)_2$, it represents ±∞ (infinity).
- $E=(11111111111)_2$ and $S!=(000\ldots000)_2$, it represents NaN (not a number).

The following is a map of E strings to numeric representations...

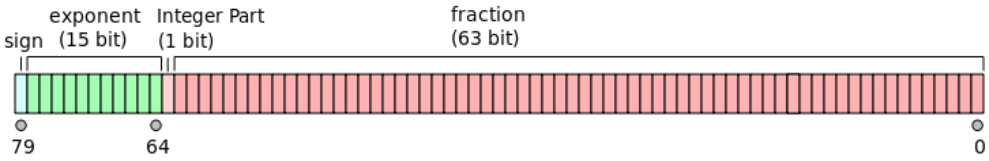
$E=(00000000000)_2=(0)_{10} \rightarrow \pm(0.SSSS\ldots)_2*2^{-1022}$ (zero and subnormals)
$E=(00000000001)_2=(1)_{10} \rightarrow \pm(1.SSSS\ldots)_2*2^{-1022}$
$E=(00000000010)_2=(2)_{10} \rightarrow \pm(1.SSSS\ldots)_2*2^{-1021}$
$E=(00000000011)_2=(3)_{10} \rightarrow \pm(1.SSSS\ldots)_2*2^{-1019}$
…
$E=(01111111111)_2=(1023)_{10} \rightarrow \pm(1.SSSS\ldots)_2*2^{0}$
$E=(10000000000)_2=(1024)_{10} \rightarrow \pm(1.SSSS\ldots)_2*2^{1}$
…
$E=(11111111100)_2=(2044)_{10} \rightarrow \pm(1.SSSS\ldots)_2*2^{1021}$
$E=(11111111101)_2=(2045)_{10} \rightarrow \pm(1.SSSS\ldots)_2*2^{1022}$
$E=(11111111110)_2=(2046)_{10} \rightarrow \pm(1.SSSS\ldots)_2*2^{1023}$
$E=(11111111111)_2=(2047)_{10} \rightarrow \pm\infty$ if $S=(000\ldots000)_2$, otherwise `NaN`

Extended Format

Extended format is not a required part of the IEEE-754 standard (it's optional), but its implementation is strongly recommended. There is no word size specified for it, but it's required to hold on to use at least…

- 15-bits for the exponent (E).
- 63-bits for the fractional portion of the significand/mantissa (S).

Unlike with single and double formats, the encoding for extended foramt isn't specified -- it's the implementer's choice how to encode extended format numbers. So, for example, an implementation can choose to include the leading bit for the significand instead of having it as a hidden bit.

> **NOTE**: This is what Intel processors do. Extended format on Intel is 80-bits, where S is 64-bits but includes the leading bit for both normalized and subnormal numbers. The following layout diagram is from Wikipedia…
>
> 
>
>
> Other platforms may keep the leading bit hidden.

> **NOTE**: Even though the encoding for E is unspecified, I imagine it has to include the range of E for double: $-1022 \leq E \leq 1023$. I suspect most implementations stick to having a biased representation encoding.

Extended format numbers are primarily used as intermediate values to minimizing errors (e.g. roundoffs, overflow, underflow?, etc..). Floating point operations upcast to extended format, perform the operation, then downcast back to the original format once complete (e.g. single format, double format, etc..).

> **NOTE**: Wikipedia says that users can directly access these intermediate values if needed, but I don't know if this is mandated by the spec or if it's something unique to Intel platforms: "To enable intermediate subexpressions results to be saved in extended precision scratch variables and continued across programming language statements, and otherwise interrupted calculations to resume where they were interrupted, it provides instructions which transfer values between these internal registers and memory without performing any conversion, which therefore enables access to the extended format for calculations also reviving the issue of the accuracy of functions of such numbers, but at a higher precision."

Rounding

If a floating point format isn't wide enough (not enough bits for exponent or significand) to represent some real number x, that number needs to be rounded. Rounding is the way of approximating x such that it's less exact but can still fit into the destination format. It's similar to truncation, except that it gives the user more control on how the number is approximated and can lead to better (more accurate) results when performing calculations.

IEEE-754 provides 4 different rounding modes…

- round towards -∞ / negative infinity (also referred to as round down)
- round towards ∞ / positive infinity (also referred to as round up)
- round towards 0
- round towards nearest

The rounding modes work as their name implies. If a floating point operation results in a number that can't be fully represented in the destination format, it'll get nudged to the closest representable number based on whatever rounding mode is chosen.

> **NOTE**: Confused about round towards nearest? It nudges towards to whichever representable number is closest to it. For example, if the number you're trying to represent is 5.6, but the floating point format only supports the numbers 5.5 and 5.75, it'll choose 5.5 -- abs(5.6-5.5)=0.1 vs abs(5.75-5.5)=0.25. If both numbers are of equal closeness, the number with the least significant bit set to 0 is chosen (documented in subsection below).

The rounding mode used most often is round towards nearest. Many higher-level languages hardcode the rounding mode to round towards nearest (e.g. Java and Python), while lower-level languages provide APIs to change it (e.g. fesetround in C and C++).

Calculation

The easiest way to calculate what x gets rounded to is to use truncation and ulp(x).

Assume that x is real number that is not a floating point number -- in other words, x is a real number but we don't have enough bits in our float point format to fully represent it. That means that either (or both) of the following is true…

- x is too small/large to be in the normalized range -- once normalized, it requires an exponent (E) smaller/larger than our floating point format can represent.
- x, once normalized, requires more precision than is available -- it requires more bits for the significand/mantissa (S) than the floating point format can represent.

> **NOTE**: Some real number x is in the normalized range of floating point system if $N_{min} \leq |x| \leq N_{max}$.
> N_{min} → smallest positive normalized number.
> N_{max} → largest positive normalized number.
> ±∞ (infinities), ±0 (zeros), and subnormals are not in the normalized range. They may be valid floating point values, but they can't be put into normalized form. NaN is also not a part of the normalized range (NaN stands for not a number).

In either case, we need to approximate x by some other value that's actually representable. For some x, we define…

- x_- → closest floating point number <= x
- x_+ → closest floating point number >= x

that we can calculate using the following algorithm…

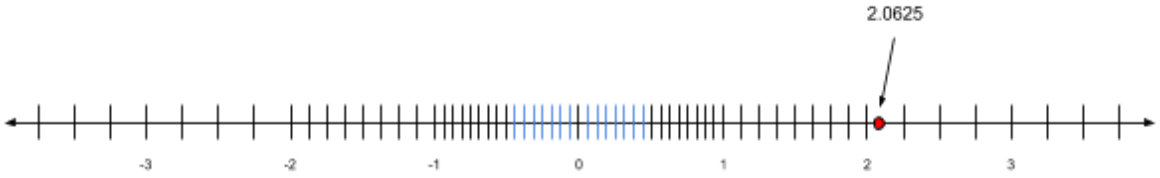
```
//   REMEMBER: ±0, ±∞, NaN, and subnormals are not normalized numbers
```

```
//   REMEMBER: ulp(x) give the gap between x and the next representable num
//   REMEMBER: trunc(x) truncates significand of x such that it can fit
//             into our floating point number format... it will always lead
//             to the closest representable number in the direction of 0
// for some normalized number x...
if (x > 0) { // if x is positive
    x_=trunc(x);
    x+=x_+ulp(x_);
} else if (x < 0) { // if x is negative
    x+=trunc(x);
    x_=x+-ulp(x);
}
```

For example, imagine a format similar to the [IEEE-754 single format](), except that…

- E (exponent) is 3-bits (still in biased representation, encoded as E+3)
- S (significand/mantissa) is 3-bits

If you were to try to represent the number 2.0625 in this format, it wouldn't be possible. The format doesn't have enough a significand/mantissa (S) wide enough to represent it...



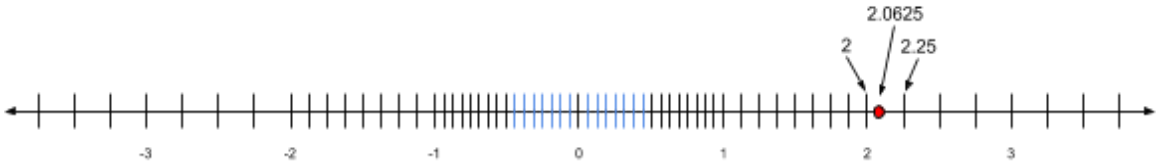
As such, rounding needs to occur. First, we need to calculate x_- and x_+...

```
// convert to binary
2.0625 = (10.0001)_2

// normalize
+(10.0001)_2 = +(10.0001)_2*2^0
             = +(1.00001)_2*2^1

// calculate ulp(x)
e=2^-3  <-- machine epsilon for our format
E=1   <-- exponent from normalization
ulp(x) = e * 2^E
ulp(2.0625) = 2^-3 * 2^1
ulp(2.0625) = 2^((-3)+1)     // confused? look up exponent rules
ulp(2.0625) = 2^-2

// calculate x_ and x+
```

$x_- = trunc(x) = trunc((1.00001)_2 * 2^1) = (1.000)_2 * 2^1 = 2$
$x_+ = x_- + ulp(x) = 2 + ulp(2.0625) = 2 + 2^{-2} = 2 + 0.25 = 2.25$

Notice how…

- x_- = 2 -- the closest representable number before 2.0625
- x_+ = 2.25 -- the closest representable number after 2.0625



Based on the rounding mode, either x_- or x_+ would be chosen to represent x. For example, if we chose a rounding mode of…

- round towards -∞ → 2 is chosen because 2 is closer to -∞ than 2.25 is
  round(x) = x_-
- round towards ∞ → 2.25 is chosen because 2.25 is closer to ∞ than 2 is
  round(x) = x_+
- round towards 0 → 2 is chosen because 2 is closer to 0 than 2.25 is
  round(x) = if (x > 0) { x_- } else if (x < 0) { x_+ } else { 0 }
- round towards nearest → 2 is chosen 2 is closer to 2.0625 than 2.25 is

```
round(x) = {
  if (x > N_max) { // special case 1 : past max
    return x < N_max + ulp(N_max)/2) ? N_max : ∞;
  }
  else if (x < -N_max) { // special case 2: past negative max
    return x < -N_max - ulp(N_max)/2 ? -N_max : -∞;
  }
  else if (abs(x - x_-) < abs(x - x_+)) { return x_-; }
  else if (abs(x - x_-) > abs(x - x_+)) { return x_+; }
  else { // special case 3: tiebreaking logic
    if (least_significant_bit_of_significand(x_-) == 0) { return x_-; }
    else if (least_significant_bit_of_significand(x_+)  == 1) { return x_+; }
  }
}
```

There are several special cases to be aware, regardless of rounding mode:

- If round(x)==0, then the sign of 0 becomes whatever the sign of x was.
- If $x>N_{max}$, then $x_-=N_{max}$ and $x_+=+\infty$ (should be obvious).
- If $x<N_{min}$, then $x_-=-\infty$ and $x_+=N_{min}$ (should be obvious).
- If x is subnormal, then $x_-=0$ or subnormal and x_+=subnormal or N_{min} (should be obvious).

> **NOTE**: The book actually wrote out the last point~~, I'm not sure why it needed to be stated as it seems obvious. Maybe because it explicitly mentioned x being in the normalized range prior to talking about how rounding is done? Subnormal numbers are not in the normalized range~~. I just realized that the books is not actually defining how rounding of subnormals is done. For example, all it says is that $x_{-}=0$ or subnormal, it doesn't say if x_{-} will be less than x or much less it'll be (it could be more than 1 representable number away?)

Round towards nearest is the default rounding mode for IEEE-754. As noted in the pseudo-code above, there are 3 special cases to be aware of for round to nearest:

1. If x is ahead of N_{max} by a certain amount, round(x) will result in a ∞. Technically, this doesn't match the definition of "round to nearest" (x will never be closer to ∞ vs N_{max}), but from a practical perspective it's good because always rounding to N_{max} may end up giving misleading results in larger calculations.
2. If x is behind N_{min} by a certain amount, round(x) will result in a -∞. This is essentially the same rule as above, but it's switched up for the opposite side. The same reasoning applies.
3. Tiebreaking happens when x if neither x_{-} nor x_{+} is closer to x (they're both equally as near). The specific name for this type of tiebreaking is called "banker's rounding" and apparently it's used a lot in finance because it doesn't cause a bias when summing rounded numbers. See https://en.wikipedia.org/wiki/Rounding#Round_half_to_even. I don't fully understand this yet.

Error

For some real number x, the "absolute error" is defined as $abserr(x) = |round(x) - x|$. It's used to determine how far off the rounded number is from the original.

For some real number x, the "relative error" is defined as $relerr(x) = |\frac{round(x)-x}{x}|$. The portion inside the absolute is sometimes to as δ: $\delta = \frac{round(x)-x}{x}$.

> **NOTE**: The book uses these 2 functions to work out a theorem (Theorem 5.1) that defines how exact a floating point number can be? It isn't worth repeating and I've skipped over the exercises for it.

Cancellation

Cancellation is the term used for the total/partial loss in accuracy when…

- x-y, where x and y are nearly equal to each other
- x+(-y), where the magnitudes of x and y are nearly equal to each other

For example, take the following block of code…

```
float x = 1.12345678f;
float y = 1.12345679f;
float res = x-y;
System.out.println(res);

double d_x = 1.12345678;
double d_y = 1.12345679;
double d_res = d_x-d_y;
System.out.println(d_res);
```

The output of this code is…

```
0.0
-1.0000000161269895E-8
```

In the first block, the single-precision floats can't represent the needed amount of precision for the literals stated. Both numbers get rounded to the same single-precision float number, which is why it results in a 0 once they get subtracted.

In the second block, the double-precision floats can't represent the exact precision need for the literals stated. This is why it results in roughly the correct value (with some extra garbage) once they get subtracted.

I think the takeaway here is that, in the course of some larger calculation, you need to be vigilant if you end up subtracting large numbers that are very close to each other. The problem has to do with rounding. Each operation will potentially produce a rounded result. When you end up subtracting those rounded results, you may end up expecting to see the results of the pre-rounded numbers.

For example, take the 2 following numbers…

$x = +(1.00000000000000000000000111111111111111111111)_2 * 2^{100}$

$y = +(1.00000000000000000000000101010101010101010101)_2 * 2^{100}$

Once rounded to fit into single format floats …

$round(x) = +(1.00000000000000000000001)_2 * 2^{100}$

$round(y) = +(1.00000000000000000000000)_2 * 2^{100}$

Then subtract the rounded numbers…

$$\begin{array}{l} +(1.00000000000000000000001)_2 * 2^{100} \\ +(1.00000000000000000000000)_2 * 2^{100} \; - \\ \quad \text{-----------------------} \\ +(0.00000000000000000000001)_2 * 2^{100} \end{array}$$

Then normalized the subtracted result...

$+(1.00000000000000000000000)_2 * 2^{77}$

Had we subtracted the real numbers instead of the rounded numbers, the results would have been…

$+(0.00000000000000000000001010101010101010101010)_2 * 2^{100}$

Then normalized the subtracted result…

$+(1.01010101010101010100)_2 * 2^{77}$

You can see how much of a loss in precision there is. The subtraction on the rounded numbers is way less accurate than the original numbers -- everything after the binary point is zero'd out. Depending on what you were doing, you may be expecting something closer to the subtraction done on the real numbers.

> **NOTE**: The example a lot of the sites give is the quadratic equation. It's a terrible first example to give because it isn't isolated, but it does show you how to organize your calculation such that subtractive cancellation doesn't end up causing a wildly inaccurate result. The book also has an example with derivatives that's equally as terrible. Page 94 of https://www.uio.no/studier/emner/matnat/math/MAT-INF1100/h08/kompendiet/round-off.pdf seems to give the best introduction, but doesn't go into enough detail. Either way, I think this is just one of the things with floating point. The system is fast but has a lot of edge cases and peculiarities that trip you up.

Exceptions

There are 5 different exception types that IEEE-754 supports:

- Invalid operation → operation was invalid or has invalid inputs (e.g. sqrt(-1))

  ```
  result           = NaN
  status_invalid   = true
  ```

- Division by zero → denominator of a division was 0 (e.g. 5/0)

  ```
  result           = ±∞
  status_divbyzero = true
  ```

- Overflow → result is finite, but too large to be representable normalized number

  ```
  result           = ±∞ or ±N_max
  fe_overflow      = true
  ```

- Underflow → result is finite but too small to be representable normalized number

  ```
  result           = ±0, subnormal, or ±N_min
  fe_underflow     = true
  ```

- Inexact → result had to be rounded to be representable

  ```
  result           = round(x)
  fe_inexact       = true
  ```

When an exception triggers, both an exception flag and a result are set. The exception flag indicates to the programmer what exceptional floating point situation occurred. The result value is suppose to be some mathematically sane value for that exceptional situation, such that even

if you didn't handle the exception the final result should still make sense. For example, if your result overflowed to ∞ but you added 1 to it, ∞+1=∞.

> **NOTE**: NaN doesn't count as a mathematically sane value. It almost always means something went wrong. ±∞ may also mean that something went wrong, but it depends on the context.

Exception flags are sticky. Once set, they'll remain set for the remainder of your calculations unless you specifically clear them. However, many higher-level languages won't let you access exception flags. For example, there's no way to access them in Java while lower-level languages like C have standard APIs to access them (see fenv.h in standard C library).

> **NOTE**: Important quote from the book: "The IEEE approach to exceptions permits a very efficient and reliable approach to programming in general, which may be summarized as: Try the easy fast way first; fix it later if an exception occurs." For example, if you run your calculations normally but notice an overflow exception, either re-work your calculations or scale down the magnitudes of the numbers you're working with so it doesn't overflow when you run it again.

One thing to be aware of are overflow/underflow exceptions are nuanced and can trigger differently across platforms…

For underflow, the book mentions: "In IEEE arithmetic, the standard response to underflow is to return the correctly rounded value, which may be a subnormal number, ±0 or $\pm N_{min}$. This is known as gradual underflow. ... Even today, some IEEE compliant microprocessors support gradual underflow only in software. The standard gives several options for defining exactly when the underflow exception is said to occur; see [CKVV02] for details."

For overflow, the book mentions: "To be precise, overflow is said to occur in IEEE arithmetic when the exact result of an operation is finite but so big that its correctly rounded value is different from what it would be if the exponent upper limit E_{max} were sufficiently large. In the case of round to nearest, this is the same as saying that overflow occurs when an exact finite result is rounded to ±∞, but it is not the same for the other rounding modes. For example, in the case of round down or round towards zero, if an exact finite result x is more than N_{max}, it is rounded down to N^{max} no matter how large x is, but overflow is said to occur only if $x > N_{max} + ulp(N_{max})$, since otherwise the rounded value would be the same even if the exponent range were increased."

Book Exercises

Chapter 2

Exercise 2.1

It's easier to convert a decimal integer to binary by determining the bits from right-to-left. This is done by iteratively dividing the decimal number, where the remainder from each division produces a bit. The first bit is the right-most and the last bit is the left-most. I've documented this in the "Decimal To Binary Conversion" section of this document.

I don't know what determine the bits from left-to-right involves. I imagine it involves lookup tables that map bit positions to decimal numbers…

$(1)_{10} = (1)_2$
$(2)_{10} = (10)_2$
$(4)_{10} = (100)_2$
$(8)_{10} = (1000)_2$
$(16)_{10} = (10000)_2$
$(32)_{10} = (100000)_2$

Then, you need to binary search the lookup table to determine which bits to use, left-to-right. For example, imagine wanting to convert the decimal number 12 to binary…

1. 12 is between 8 and 16, so map to 8: 1000
2. Subtract 12 from 8: 12-8=4
3. 4 is in the table, so map to 4: 100
4. Subtract 4 from 4: 4-4=0

You have determined the bits left-to-right. You can do a bit-wise OR on all the numbers you pulled to get the final number:

```
1000
 100
----
1100
```

As for the fractional portion, it's easier to do it the iterative way as well (detailed in the same section of this document). The only difference is that for the fractional portion, the bits come out left-to-right instead of right-to-left.

Having said that, you can probably use a lookup table for the fractional portion as well. The only difference is that the lookup table will have a limit to it while the iterative model can go on for as

long as you want it to. So, if you were going to convert a terminating fractional portion in base10 to a non-terminating/repeating fraction portion in base 2, the iterative model would allow you to pull in an arbitrary amount of precision…

$(0.1)_{10} = (0.0001100110011\ldots)_2$

Chapter 3

Exercise 3.1

$2^{32} - 1$ different integers are available for 2's complement
$2^{32} - 2$ different integers are available for sign-and-magnitude (because 0 appears twice? so we take off an extra 1)

2's complement is the one with a unique zero.

Exercise 3.2

For a two's complement b-bit format, you can represent the integers $[-2^{(b-1)}, 2^{(b-1)} - 1]$. So, for a 16-bit format, it would be $[-2^{(16-1)}, 2^{(16-1)} - 1]$ or $[-2^{(15)}, 2^{(15)} - 1]$ or $[-32768, 32767]$

Exercise 3.3

For 2s complement…

$(1)_{10} = (00000001)_2$
$(10)_{10} = (00001010)_2$
$(100)_{10} = (01100100)_2$
$(-1)_{10} = (11111111)_2$
$(-10)_{10} = (11110110)_2$
$(-100)_{10} = (10011100)_2$

Exercise 3.4

I don't understand the question? What's below is the closest I can do. I don't know if this is asking for a formal proof or something.

2^{31}-1 is the maximum value of a 32-bit two's complement integer. This is represented as...
0111 1111 1111 1111 1111 1111 1111 1111

If you continually subtract 1 from this value using standard unsigned subtraction arithmetic (the kind used by two's complement), the left-most bit will stay at 0 until it reaches 0.

-1 in 32-bit two's complement is represented as...
`1111 1111 1111 1111 1111 1111 1111 1111`
If you continually add 1 to this value using standard unsigned addition arithmetic (the kind used by two's complement), the left-most bit will stay at 1 until it reaches -2^{31}...
`1000 0000 0000 0000 0000 0000 0000 0000`

Exercise 3.5

To change the sign of a positive integer to a negative integer in two's complement, bitwise not the integer and add 1. For example, plot out the numbers for a 3-bit integer space...

```
0 = 000
1 = 001    111 = -1
2 = 010    110 = -2
3 = 011    101 = -3
           100 = -4
```

Notice that both the negatives and positives have 4 integers, but the negatives start from -1 while the positives start at 0. You need to account for this skew either before or after performing a bitwise not.

Say we want to negate 2...

2_{10} is 010_2
NOT(010) = 101 (-3_{10}) <-- bitwise NOT the positive number to get -3
101 + 1 = 110 (-2_{10}) <-- add 1 to -3 to adjust skew, giving you -2

Exercise 3.6

50_{10} --> 00110010
-100_{10} --> 10011100 +

11001110 --> -50_{10}

-50_{10} --> 11001110
100_{10} --> 01100100 +

~~1~~00110010 --> 50_{10}

```
50₁₀    --> 00110010
50₁₀    --> 00110010 +
            --------
            01100100 --> 100₁₀
```

Exercise 3.7

Modulo?

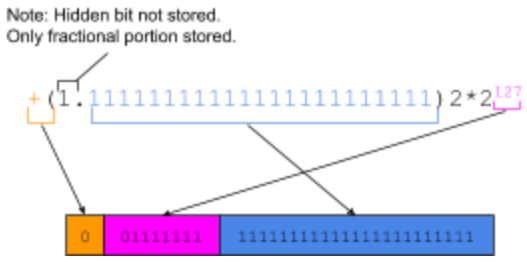
Exercise 3.8

Assuming a 32-bit word where…

- ± (sign) goes into bit 0
- E (exponent) goes into bits 1-8
- E is in two's complement so it's ranged $[-2^7, 2^7 - 1]$ or $[-128, 127]$.
- S (significant/mantissa) goes into bits 9-31

E is in two's complement so it's ranged $[-2^7, 2^7 - 1]$ or $[-128, 127]$.

That means the largest floating point number that can be expressed by this system is…
$+(1.11111111111111111111111)_2 * 2^{127}$



Exercise 3.9

Assuming a 32-bit word where…

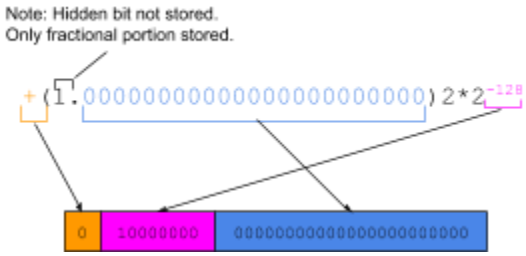
- ± (sign) goes into bit 0
- E (exponent) goes into bits 1-8
- E is in two's complement so it's ranged $[-2^7, 2^7 - 1]$ or $[-128, 127]$.
- S (significant/mantissa) goes into bits 9-31

That means the smallest positive floating point number that can be expressed by this format is…

$+(1.0)_2 * 2^{-128}$

which can also be written as…

$\frac{1}{2^{-128}}$



NOTE: It can't be 0. Also, 0 is neither a positive nor negative integer.
A number that is > 0 is positive.
A number that is < 0 is negative.

NOTE: Even if it could be 0, we can't represent 0 in our system. The hidden bit is always set to 1 which prevents us from being able to represent 0.

Exercise 3.10

Assuming a 32-bit word where…

- ± (sign) goes into bit 0
- E (exponent) goes into bits 1-8
- E is in two's complement, so it's ranged $[-2^7, 2^7 - 1]$ or $[-128, 127]$.
- S (significant/mantissa) goes into bits 9-31

NOTE: Remember only the fractional portion of S is stored. This is called the fractional field. If you need a review of why this is, go here.

What is the smallest positive INTEGER number that CANNOT EXACTLY be expressed by this format?

Remember the format for binary floating point is… $\pm S*2^E$

If we start plotting out positive integers in our 32-bit word format…

$+(1.00000000000000000000000)_2 * 2^0 = (1)_2$ // 1

$+(1.00000000000000000000000)_2 * 2^1 = (10)_2 \quad // 2$
$+(1.10000000000000000000000)_2 * 2^1 = (11)_2 \quad // 3$
$+(1.00000000000000000000000)_2 * 2^2 = (100)_2 \quad // 4$
$+(1.01000000000000000000000)_2 * 2^2 = (101)_2 \quad // 5$
$+(1.10000000000000000000000)_2 * 2^2 = (110)_2 \quad // 6$
$+(1.11000000000000000000000)_2 * 2^2 = (111)_2 \quad // 7$
...
$+(1.11111111111111111111110)_2 * 2^{23} = (11111111111111111111110)_2$
$+(1.11111111111111111111111)_2 * 2^{23} = (11111111111111111111111)_2$
$+(1.00000000000000000000000)_2 * 2^{24} = (10000000000000000000000)_2$
$+(1.00000000000000000000001)_2 * 2^{24} = (10000000000000000000001)_2$

So, as we keep increasing E (exponent), the point floats to the right. Once E = 23 (number of bits for storing S), that's the limit. The moment E goes past 23, we risk running into an integer that cannot be exactly represented. Pretty much any integer that has a 1 bit set at index >= 24 CANNOT be EXACTLY represented by our format.

The last number in the above list is $(10000000000000000000000001)_2$. That number is the smallest positive integer that cannot be exactly expressed by our 32-bit word format.

This is easier to think about if we lower the width of E (exponent) to 3-bits (range of $[-4, 3]$) and S (significand) to 2-bits...

$+(1.00)_2 * 2^0 = (1)_2 \quad // 1$
$+(1.00)_2 * 2^1 = (10)_2 \quad // 2$
$+(1.10)_2 * 2^1 = (11)_2 \quad // 3$
$+(1.00)_2 * 2^2 = (100)_2 \quad // 4$
$+(1.01)_2 * 2^2 = (101)_2 \quad // 5$
$+(1.10)_2 * 2^2 = (110)_2 \quad // 6$
$+(1.11)_2 * 2^2 = (111)_2 \quad // 7$
$+(1.000)_2 * 2^3 = (1000)_2 \quad // 8$
$+(1.001)_2 * 2^3 = (1001)_2 \quad // 9$ <-- NOT REPRESENTABLE

The number $(1000)_2$ CAN be represented because when we float the point past the width of S (2-bits), those non-existent bits pretty much default to 0.

The number $(1001)_2$ CANNOT be represented for the same reason. When we float past the width of S (2-bits), those non-existent bits default to 0. There's no way to represent the 1 at the end of $(1001)_2$ because it's position is past the past the width of S.

Exercise 3.11

Up until this point in the book S has been defined as ±S * 2^E where $1 \leq S < 2$. However, for this problem we change the constraint on S to $\frac{1}{2} \leq S < 1$. In other words, for this problem S is $[0.5, 1)$.

> **NOTE**: Converting 0.5 comes out to binary is $(0.1)_2$

If you start listing out ascending values for S where $\frac{1}{2} \leq S < 1$, you can see how the…

- whole number portion of S will always be the binary digit 0
- fractional portion of S will always start with the binary digit 1

$0.5 = (0.1)_2$
$0.625 = (0.101)_2$
$0.75 = (0.11)_2$
$0.875 = (0.111)_2$
$0.90625 = (0.11101)_2$
$0.96875 = (0.11111)_2$
$0.9921875 = (0.1111111)_2$
...

Essentially, instead of having 1 hidden bit for S, we now have 2:

- index0 = 0 (for the whole number portion).
- index1 = 1 (for the fractional portion).

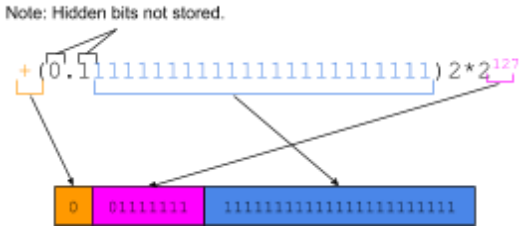
All bits after these 2 are the bits we store.

With this new way of storing S, assume a 32-bit word where…

- ± (sign) goes into bit 0
- E (exponent) goes into bits 1-8
- E is in two's complement, so it's ranged $[-2^7, 2^7 - 1]$ or $[-128, 127]$.
- S (significant/mantissa) goes into bits 9-31

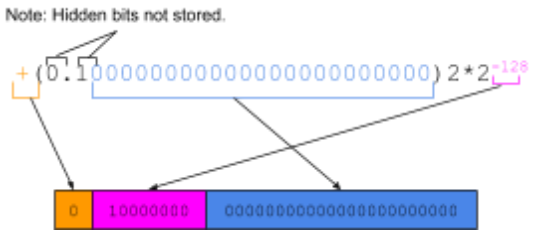
The largest floating point number that can be expressed by this format is…

$+(0.1111111111111111111111)_2 * 2^{127}$



The smallest positive floating point number that can be expressed by this format is…

$+(0.10000000000000000000000)_2 * 2^{-128}$



The smallest positive INTEGER number that CANNOT EXACTLY be expressed by this system is…

$+(0.1000000000000000000000000)_2 * 2^1 = (1)_2 \quad // 1$
$+(0.1000000000000000000000000)_2 * 2^2 = (10)_2 \quad // 2$
$+(0.1100000000000000000000000)_2 * 2^2 = (11)_2 \quad // 3$
$+(0.1000000000000000000000000)_2 * 2^3 = (100)_2 \quad // 4$
$+(0.1010000000000000000000000)_2 * 2^3 = (101)_2 \quad // 5$
$+(0.1100000000000000000000000)_2 * 2^3 = (110)_2 \quad // 6$
$+(0.1110000000000000000000000)_2 * 2^3 = (111)_2 \quad // 7$
...
$+(0.1111111111111111111111110)_2 * 2^{24} = (111111111111111111111110)_2$
$+(0.1111111111111111111111111)_2 * 2^{24} = (111111111111111111111111)_2$
$+(0.1000000000000000000000000)_2 * 2^{25} = (100000000000000000000000)_2$
$+(0.1000000000000000000000001)_2 * 2^{25} = (100000000000000000000001)_2$

The last item in the list is the integer that cannot be represented exactly. We don't have enough bits in S to be able to represent it exactly. If you want insight into why this is, see the answer for exercise 3.10.

Exercise 3.12

Assuming a 32-bit word where…

- ± (sign) goes into bit 0
- E (exponent) goes into bits 1-8
- E is in two's complement, so it's ranged $[-2^7, 2^7 - 1]$ or $[-128, 127]$.
- S (significant/mantissa) goes into bits 9-31

Calculate the unit in the last place (ulp)...

```
p = 24       // precision (num of bits in S + hidden bits)
e = 2^-(p-1)    // machine epsilon
```

$= 2^{-(24-1)}$

$= 2^{-23}$

$ulp(x) = e * 2^{E}$ // unit in last place

$= 2^{-23} * 2^{E}$

$= 2^{(-23 + E)}$ // confused? look up exponent rules

> **NOTE**: Remember that precision needs to include the hidden bits. See the precision section for more information.

ulp(0.25)

// get into normalized form and extract E (exponent)

$(0.25)_{10} = +(0.01000000000000000000000)_2 * 2^0$

$= +(0.10000000000000000000000)_2 * 2^{-1}$

$= +(1.00000000000000000000000)_2 * 2^{-2}$

// calculate ulp

$ulp(0.25) = 2^{(-23 + -2)}$

$= 2^{(-25)}$

ulp(2)

// get into normalized form and extract E (exponent)

$(2)_{10} = +(10.00000000000000000000000)_2 * 2^0$

$= +(1.00000000000000000000000)_2 * 2^1$

// calculate ulp

$ulp(2) = 2^{(-23 + 1)}$

$= 2^{(-22)}$

ulp(3)

// get into normalized form and extract E (exponent)

$(3)_{10} = +(11.00000000000000000000000)_2 * 2^0$

$= +(1.10000000000000000000000)_2 * 2^1$

// calculate ulp

$ulp(3) = 2^{(-23 + 1)}$

$= 2^{(-22)}$

ulp(4)

// get into normalized form and extract E (exponent)

$(4)_{10} = +(100.00000000000000000000)_2 * 2^0$

$= +(10.00000000000000000000000)_2 * 2^1$

$= +(1.00000000000000000000000)_2 * 2^2$

// calculate ulp

$ulp(4) = 2^{(-23 + 2)}$

$= 2^{(-21)}$

```
ulp(10)
// get into normalized form and extract E (exponent)
(10)10 = +(1010.00000000000000000000)2 * 2^0
       = +(101.00000000000000000000)2 * 2^1
       = +(10.10000000000000000000)2 * 2^2
       = +(1.01000000000000000000)2 * 2^3
       = +(1.01000000000000000000000)2 * 2^4
// calculate ulp
ulp(10) = 2^(-23 + 4)
        = 2^(-19)

ulp(100)
// get into normalized form and extract E (exponent)
(100)10 = +(1100100.00000000000000000)2 * 2^0
        = +(110010.00000000000000000)2 * 2^1
        = +(11001.00000000000000000)2 * 2^2
        = +(1100.10000000000000000)2 * 2^3
        = +(110.01000000000000000000)2 * 2^4
        = +(11.00100000000000000000)2 * 2^5
        = +(1.10010000000000000000000)2 * 2^6
// calculate ulp
ulp(100) = 2^(-23 + 6)
         = 2^(-17)

ulp(1030)
// get into normalized form and extract E (exponent)
(1030)10 = +(10000000110.0000000000000)2 * 2^0
         = +(1000000011.000000000000000)2 * 2^1
         = +(100000001.100000000000000)2 * 2^2
         = +(10000000.110000000000000)2 * 2^3
         = +(1000000.011000000000000000)2 * 2^4
         = +(100000.001100000000000000)2 * 2^5
         = +(10000.000110000000000000)2 * 2^6
         = +(1000.000011000000000000)2 * 2^7
         = +(100.000001100000000000)2 * 2^8
         = +(10.000000110000000000)2 * 2^9
         = +(1.00000001100000000000000)2 * 2^10
// calculate ulp
ulp(1030) = 2^(-23 + 10)
          = 2^(-13)
```

You can verify the correctness of any of these by adding the result to the input. It should fit into the storage format without any rounding/truncation. For example, the verifying the last answer…

$(1030)_{10} = (10000000110.0)_2$ // input into ulp

$(2^{(-13)})_2 = (0.0000000000001)_2$ // result of ulp

// add them together

```
10000000110.0000000000000
00000000000.0000000000001 +
-------------------------
10000000110.0000000000001
```

// normalize the result of the add

$+(10000000110.0000000000001)_2 * 2^0$

$+(1000000011.0000000000001)_2 * 2^1$

$+(100000001.1000000000001)_2 * 2^2$

$+(10000000.1100000000001)_2 * 2^3$

$+(1000000.0110000000001)_2 * 2^4$

$+(100000.0011000000001)_2 * 2^5$

$+(10000.0001100000001)_2 * 2^6$

$+(1000.0000110000001)_2 * 2^7$

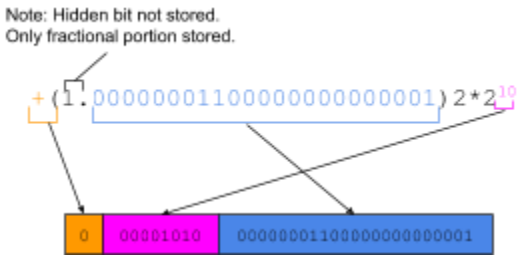
$+(100.0000011000001)_2 * 2^8$

$+(10.0000001100001)_2 * 2^9$

$+(1.0000000110000)_2 * 2^{10}$

// it's the next number that's representable in the storage format

// no rounding/truncation needed

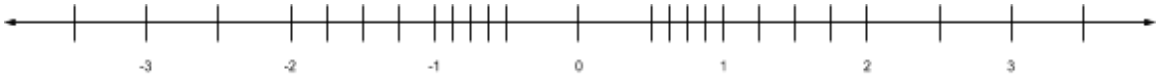


Exercise 3.13

For a storage format where…

- $-1 \leq E \leq 1$
- S has 3-bits: S.SS where 0 is explicitly represented as 0.00 (no hidden bits in S)

All representable numbers are…



$(1.00)_2 * 2^{-1} = (0.100)_2 = 0.5$

$(1.01)_2 * 2^{-1} = (0.101)_2 = 0.625$

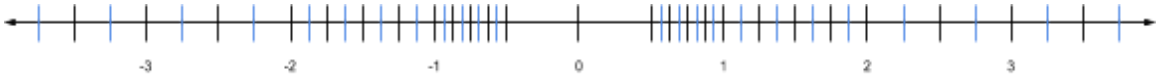
$(1.10)_2 * 2^{-1} = (0.110)_2 = 0.75$

$(1.11)_2 * 2^{-1} = (0.111)_2 = 0.875$

$(1.00)_2 * 2^{0} = (1.00)_2 = 1.0$
$(1.01)_2 * 2^{0} = (1.01)_2 = 1.25$
$(1.10)_2 * 2^{0} = (1.10)_2 = 1.5$
$(1.11)_2 * 2^{0} = (1.11)_2 = 1.75$

$(1.00)_2 * 2^{1} = (10.0)_2 = 2.0$
$(1.01)_2 * 2^{1} = (10.1)_2 = 2.5$
$(1.10)_2 * 2^{1} = (11.0)_2 = 3.0$
$(1.11)_2 * 2^{1} = (11.1)_2 = 3.5$

If you update to such that S has 4-bits (new lines are inserted in blue)…



$(1.000)_2 * 2^{-1} = (0.1000)_2 = 0.5$
$(1.001)_2 * 2^{-1} = (0.1001)_2 = 0.5625$
$(1.010)_2 * 2^{-1} = (0.1010)_2 = 0.625$
$(1.011)_2 * 2^{-1} = (0.1011)_2 = 0.6875$
$(1.100)_2 * 2^{-1} = (0.1100)_2 = 0.75$
$(1.101)_2 * 2^{-1} = (0.1101)_2 = 0.8125$
$(1.110)_2 * 2^{-1} = (0.1110)_2 = 0.875$
$(1.111)_2 * 2^{-1} = (0.1111)_2 = 0.9375$

$(1.000)_2 * 2^{0} = (1.000)_2 = 1.0$
$(1.001)_2 * 2^{0} = (1.001)_2 = 1.125$
$(1.010)_2 * 2^{0} = (1.010)_2 = 1.25$
$(1.011)_2 * 2^{0} = (1.011)_2 = 1.375$
$(1.100)_2 * 2^{0} = (1.100)_2 = 1.5$
$(1.101)_2 * 2^{0} = (1.101)_2 = 1.625$
$(1.110)_2 * 2^{0} = (1.110)_2 = 1.75$
$(1.111)_2 * 2^{0} = (1.111)_2 = 1.875$

$(1.000)_2 * 2^{1} = (10.00)_2 = 2.0$
$(1.001)_2 * 2^{1} = (10.01)_2 = 2.25$
$(1.010)_2 * 2^{1} = (10.10)_2 = 2.5$
$(1.011)_2 * 2^{1} = (10.11)_2 = 2.75$
$(1.100)_2 * 2^{1} = (11.00)_2 = 3.0$
$(1.101)_2 * 2^{1} = (11.01)_2 = 3.25$
$(1.110)_2 * 2^{1} = (11.10)_2 = 3.5$
$(1.111)_2 * 2^{1} = (11.11)_2 = 3.75$

Chapter 4

Exercise 4.1

The IEEE single format floating point representation for 2 is…

```
// convert to bin
 (2)10=(10)2
// normalize
 +(10.0)2*2^0
 +(1.00)2*2^1
// calculate what E will be encoded as
 E=(1+127)10=(128)10=(10000000)2
// put it all together
 ±=0
 E=10000000
 S=00000000000000000000000
```

The IEEE single format floating point representation for 30 is…

```
// convert to bin
 (30)10=(11110)2
// normalize
 +(11110.0)2*2^0
 +(1111.00)2*2^1
 +(111.100)2*2^2
 +(11.1100)2*2^3
 +(1.11100)2*2^4
// calculate what E will be encoded as
 E=(4+127)10=(131)10=(10000011)2
// put it all together
 ±=0
 E=10000011
 S=11100000000000000000000
```

The IEEE single format floating point representation for 31 is…

```
// convert to bin
 (31)10=(11111)2
// normalize
 +(11111.0)2*2^0
 +(1111.10)2*2^1
 +(111.110)2*2^2
 +(11.1110)2*2^3
 +(1.11110)2*2^4
```

```
// calculate what E will be encoded as
 E=(4+127)10=(131)10=(10000011)2
// put it all together
 ±=0
 E=10000011
 S=11110000000000000000000
```

The IEEE single format floating point representation for 32 is…

```
// convert to bin
 (32)10=(100000)2
// normalize
 +(100000.0)2*2^0
 +(10000.00)2*2^1
 +(1000.000)2*2^2
 +(100.0000)2*2^3
 +(10.00000)2*2^4
 +(1.000000)2*2^5
// calculate what E will be encoded as
 E=(5+127)10=(132)10=(10000100)2
// put it all together
 ±=0
 E=10000100
 S=00000000000000000000000
```

The IEEE single format floating point representation for 33 is…

```
// convert to bin
 (33)10=(100001)2
// normalize
 +(100001.0)2*2^0
 +(10000.10)2*2^1
 +(1000.010)2*2^2
 +(100.0010)2*2^3
 +(10.00010)2*2^4
 +(1.000010)2*2^5
// calculate what E will be encoded as
 E=(5+127)10=(132)10=(10000100)2
// put it all together
 ±=0
 E=10000100
 S=00000100000000000000000
```

The IEEE single format floating point representation for $\frac{23}{4}$ is…

// convert to bin
(23/4)$_{10}$=(10111/100)$_2$=(101.11)$_2$
// normalize
+(101.11)$_2$*2^0
+(10.111)$_2$*2^1
+(1.0111)$_2$*2^2
// calculate what E will be encoded as
E=(2+127)$_{10}$=(129)$_{10}$=(10000001)$_2$
// put it all together
±=0
E=10000001
S=01110000000000000000000

The IEEE single format floating point representation for $\frac{23}{4} \cdot 2^{100}$ is…

// convert to bin
(23/4)$_{10}$*2^{100}=(10111/100)$_2$*2^{100}=(101.11)$_2$*2^{100}
// normalize
+(101.11)$_2$*2^{100}
+(10.111)$_2$*2^{101}
+(1.0111)$_2$*2^{102}
// calculate what E will be encoded as
E=(102+127)$_{10}$=(229)$_{10}$=(11100101)$_2$
// put it all together
±=0
E=11100101
S=01110000000000000000000

The IEEE single format floating point representation for $\frac{23}{4} \cdot 2^{-100}$ is…

// convert to bin
(23/4)$_{10}$*2^{-100}=(10111/100)$_2$*2^{-100}=(101.11)$_2$*2^{-100}
// normalize
+(101.11)$_2$*2^{-100}
+(10.111)$_2$*2^{-99}
+(1.0111)$_2$*2^{-98}
// calculate what E will be encoded as
E=((-98)+127)$_{10}$=(29)$_{10}$=(00011101)$_2$
// put it all together
±=0
E=00011101
S=01110000000000000000000

The IEEE single format floating point representation for $\frac{23}{4} \cdot 2^{-135}$ is…

```
// convert to bin
 (23/4)10*2-135=(10111/100)2*2-135=(101.11)2*2-135
// normalize
 +(101.11)2*2-135
 +(10.111)2*2-134
 +(1.0111)2*2-133   ← INVALID
   // Remember that E must be between [-126,127]
   //
   // -133 is too small, it won't fit into the 8-bits we have
   // for E... as such, this needs to be stored as a subnormal.
// move dot to the left until E=-126 (min)
 +(0.10111)2*2-132
 +(0.010111)2*2-131
 +(0.0010111)2*2-130
 +(0.00010111)2*2-129
 +(0.000010111)2*2-128
 +(0.0000010111)2*2-127
 +(0.000000010111)2*2-126
 E=(00000000)2   ← special bitstring used for zero and subnormals
// put it all together
 ±=0
 E=00000000
 S=00000010111000000000000
```

The IEEE single format floating point representation for $\frac{(1)_2}{(10)_2} \cdot 2$ is...

```
// fraction to number
 (1/10)10*21=(0.0001100110011001100110...)2*21
            =(0.0011001100110011001100...)2
// normalize
   // note that we aren't truncating until AFTER we've normalized
 +(0.0011001100110011001100...)2*20
 +(0.0110011001100110011001...)2*21
 +(0.1100110011001100110011...)2*22
 +(1.1001100110011001100110...)2*23
// calculate what E will be encoded as
 E=((3)+127)10=(130)10=(10000010)2
// put it all together
 ±=0
 E=10000010
 S=10011001100110011001100
```

The IEEE single format floating point representation for $\frac{(1)_2}{(10)_2} \cdot 2^{11}$ is...

// fraction to number
$(1/10)_{10}*2^{11}=(0.000110011001100110011001100...)_2*2^{11}$
// normalize
// note that we aren't truncating until AFTER we've normalized
// note that each...
// decrement of E moves us 1 place to the right
// increment of E moves us 1 place to the left
// don't believe it? Try it out with smaller numbers
$+(0.000110011001100110011001100...)_2*2^{11}$
$+(0.001100110011001100110011001...)_2*2^{10}$
$+(0.011001100110011001100110011...)_2*2^{9}$
$+(0.110011001100110011001100110...)_2*2^{8}$
$+(1.100110011001100110011001100...)_2*2^{7}$
// calculate what E will be encoded as
$E=(7+127)_{10}=(134)_{10}=(10000110)_2$
// put it all together
±=0
E=10000110
S=10011001100110011001100

The IEEE single format floating point representation for $\frac{(1)_2}{(10)_2}\cdot 2^{-140}$ is...

// fraction to number
$(1/10)_{10}*2^{-140}=(0.000110011001100110011001100...)_2*2^{-140}$
// normalize
// note that we aren't truncating until AFTER we've normalized
// note that each...
// decrement of E moves us 1 place to the right
// increment of E moves us 1 place to the left
// don't believe it? Try it out with smaller numbers
$+(0.000110011001100110011001100...)_2*2^{-140}$
$+(0.001100110011001100110011001...)_2*2^{-139}$
$+(0.011001100110011001100110011...)_2*2^{-138}$
$+(0.110011001100110011001100110...)_2*2^{-137}$
$+(1.100110011001100110011001100...)_2*2^{-136}$
//
// Remember that E must be between [-126,127]
//
// -136 is too small, it won't fit into the 8-bits we have
// for E... as such, this needs to be stored as a subnormal.
// move dot to the left until E=-126 (min)
$+(1.100110011001100110011001100...)_2*2^{-136}$
$+(0.110011001100110011001100110...)_2*2^{-135}$

$+(0.011001100110011001100110011...)_2 * 2^{-134}$
$+(0.001100110011001100110011001...)_2 * 2^{-133}$
$+(0.000110011001100110011001100...)_2 * 2^{-132}$
$+(0.000011001100110011001100110...)_2 * 2^{-131}$
$+(0.000001100110011001100110011...)_2 * 2^{-130}$
$+(0.000000110011001100110011001...)_2 * 2^{-129}$
$+(0.000000011001100110011001100...)_2 * 2^{-128}$
$+(0.000000001100110011001100110...)_2 * 2^{-127}$
$+(0.000000000110011001100110011...)_2 * 2^{-126}$
$E=(00000000)_2$ ← special bitstring used for zero and subnormals

// put it all together
±=0
E=00000000
S=00000000011001100110011

Exercise 4.2

The gap between 2 and the first IEEE single format number larger than 2…

// convert to binary
$(2)_{10}=(10)_2$
// normalize
$+(10.0)_2 * 2^0$
$+(1.00)_2 * 2^1$
// calculate ulp
p=24 <-- precision for single format (23 bits + 1 hidden bit)
$e=2^{(-(p-1))}$
$=2^{-23}$ <-- machine epsilon for single format
E=1 <-- exponent from normalization
$ulp(x) = e * 2^E$
$ulp(2) = 2^{-23} * 2^1$
$ulp(2) = 2^{((-23)+1)}$
$ulp(2) = 2^{-22}$ <-- gap between 2 and next representable num in IEEE single format

The gap between 1024 and the first IEEE single format number larger than 1024…

// convert to binary
$(1024)_{10}=(10000000000)_2$
// normalize
$+(10000000000.0)_2 * 2^0$
$+(1000000000.00)_2 * 2^1$
$+(100000000.000)_2 * 2^2$
$+(10000000.0000)_2 * 2^3$

$+(1000000.00000)_2 * 2^4$
$+(100000.000000)_2 * 2^5$
$+(10000.0000000)_2 * 2^6$
$+(1000.00000000)_2 * 2^7$
$+(100.000000000)_2 * 2^8$
$+(10.0000000000)_2 * 2^9$
$+(1.00000000000)_2 * 2^{10}$

// calculate ulp

p=24 <-- precision for single format (23 bits + 1 hidden bit)

$e=2^{(-(p-1))}$

$=2^{-23}$ <-- machine epsilon for single format

E=1 <-- exponent from normalization

$ulp(x) = e * 2^E$

$ulp(2) = 2^{-23} * 2^{10}$

$ulp(2) = 2^{((-23)+10)}$

$ulp(2) = 2^{-13}$ <-- gap between 1024 and next representable num in IEEE single format

Exercise 4.3

Given 2 IEEE signel format floating point numbers, the following code compares them to see if x is less than, equal to, or greater than y...

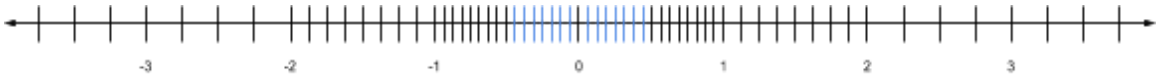
```
def test():
    x_sign_field = 0
    x_exp_field = 0b00000000
    x_significand_field = 0b10000000000000000000000
    y_sign_field = 1
    y_exp_field = 0b00000001
    y_signifcand_field = 0b10000000000000000000000

    ## this is confusing but remember that for sign 1=neg and 0=pos
    if x_sign_field == 0 and y_sign_field == 1:
        print("x>y")
        return
    elif x_sign_field == 1 and y_sign_field == 0:
        print("x<y")
        return

    if x_exp_field > y_exp_field:
        print("x>y")
        return
```

```
    elif x_exp_field < y_exp_field:
        print("x<y")
        return

    if x_significand_field > y_signifcand_field:
        print("x>y")
        return
    elif x_significand_field < y_signifcand_field:
        print("x<y")
        return

    print("x==y")


test();
```

This code does not take into account special numbers: `±0/±∞/NaN`.

Exercise 4.4

Plotting the subnormals for the toy number system from Exercise 3.13 (new lines are inserted in blue)…



$-(0.111)_2*2^{-1} = -0.4375$
$-(0.110)_2*2^{-1} = -0.375$
$-(0.101)_2*2^{-1} = -0.3125$
$-(0.100)_2*2^{-1} = -0.25$
$-(0.011)_2*2^{-1} = -0.1875$
$-(0.010)_2*2^{-1} = -0.125$
$-(0.001)_2*2^{-1} = -0.0625$
$+(0.001)_2*2^{-1} = +0.0625$
$+(0.010)_2*2^{-1} = +0.125$
$+(0.011)_2*2^{-1} = +0.1875$
$+(0.100)_2*2^{-1} = +0.25$
$+(0.101)_2*2^{-1} = +0.3125$
$+(0.110)_2*2^{-1} = +0.375$
$+(0.111)_2*2^{-1} = +0.4375$

Chapter 5

Exercise 5.1

The single format (single-precision float) rounded value for 0.1 is…

```
// convert to binary
0.1 = (0.0001100110011001100110011001...)_2

// normalize
+(0.0001100110011001100110011001...)_2=+(0.0001100110011001100110011001...)_2*2^0

=+(0.0011001100110011001100110011...)_2*2^-1

=+(0.0110011001100110011001100110...)_2*2^-2

=+(0.1100110011001100110011001100...)_2*2^-3

=+(1.1001100110011001100110011001...)_2*2^-4

// calculate ulp(x)
e=2^-23  <-- machine epsilon for single-precision float
E=-4   <-- exponent from normalization
ulp(x) = e * 2^E
ulp(0.1) = 2^-23 * 2^-4
ulp(0.1) = 2^((-23)+(-4))     // confused? look up exponent rules
ulp(0.1) = 2^-27

// calculate x_
x_ = trunc(x)
   = trunc(+(1.10011001100110011001101...)_2*2^-4)
   = +(1.10011001100110011001100)_2*2^-4

// calculate x+
x+ = x_ + ulp(x)
   = +(1.10011001100110011001100)_2*2^-4 + 2^-27
   = +(1.10011001100110011001101)_2*2^-4 // confused? expand + add by
hand

// ROUNDING MODES
round to -∞       = x_ = +(1.10011001100110011001100)_2*2^-4
round to ∞        = x+ = +(1.10011001100110011001101)_2*2^-4
```

round to 0 = x_- = $+(1.10011001100110011001100)_2*2^{-4}$
round to nearest = x_- = $+(1.10011001100110011001100)_2*2^{-4}$
// why is round to nearset = x_-? Because x_- is closer to x than x_+.
// Either plot it on the a number line to see which is closer, or
// do the calculations: abs(x-x_-) vs abs(x-x_+)

The single format (single-precision float) rounded value for $1+2^{-25}$ is…

// convert to binary
$1+2^{-25} = (1.00000000000000000000000001)_2$

// This is in normalized form already, but precision exceeds 23bits.
// Either way, we have enough to handle rounding at this point?
$+(1.00000000000000000000000001)_2*2^{-25}$

// calculate ulp(x)
$e=2^{-23}$ <-- machine epsilon for single-precision float
E=-25 <-- exponent from normalization
$ulp(x) = e * 2^E$
$ulp(1+2^{-25}) = 2^{-23} * 2^{-25}$
$ulp(1+2^{-25}) = 2^{((-23)+(-25))}$ // confused? look up exponent rules
$ulp(1+2^{-25}) = 2^{-48}$

// calculate x_-
$x_- = trunc(x)$
$= trunc(+(1.00000000000000000000000001)_2*2^{-25})$
$= +(1.00000000000000000000000)_2*2^{-25}$

// calculate x_+
$x_+ = x_- + ulp(x)$
$= +(1.00000000000000000000000)_2*2^{-25} + 2^{-48}$
$= +(1.00000000000000000000001)_2*2^{-25}$ //confused? expand + add by hand

// ROUNDING MODES
round to -∞ = x_- = $+(1.00000000000000000000000)_2*2^{-25}$
round to ∞ = x_+ = $+(1.00000000000000000000001)_2*2^{-25}$
round to 0 = x_- = $+(1.00000000000000000000000)_2*2^{-25}$
round to nearest = x_- = $+(1.00000000000000000000000)_2*2^{-25}$
// why is round to nearset = x_-? Because x_- is closer to x than x_+.
// Either plot it on the a number line to see which is closer, or
// do the calculations: abs(x-x_-) vs abs(x-x_+)

The single format (single-precision float) rounded value for 2^{130} is…

```
// no point in converting to binary, obvious what the num is in
binary
2^130

// This is in normalized form already, but precision exceeds 23bits.
// Either way, we have enough to handle rounding at this point?
+(1.00000000000000000000000)_2*2^130

// on point in calculating ulp -- E is 130, which is past the max...
//   this number is past the normalized range

// calculate x_ -- based on special cases
x_ = N_max

// calculate x+ -- based on special cases
x+ = ∞

// ROUNDING MODES
round to -∞      = x_ = N_max
round to ∞       = x+ = ∞
round to 0       = x_ = N_max
round to nearest = x+ = N_max
// why is round to nearset = x_? Because it hit special case #1 for
// round towards nearest...
//
// calculate ulp(N_max)
//   e=2^-23  <-- machine epsilon for single-precision float
//   E=127  <-- exponent from N_max (for single-precision float)
//   ulp(x) = e * 2^E
//   ulp(N_max) = 2^-23 * 2^127
//   ulp(N_max) = 2^((-23)+127)     // confused? look up exponent rules
//   ulp(N_max) = 2^104
//
// calculate ulp(N_max)/2
//   ulp(N_max)/2 = 2^104/2
//                = 2^103
//
// calculate N_max + ulp(N_max)/2
//   N_max = (1.11111111111111111111110)_2*2^127
//   2^103 = (0.00000000000000000000001)_2*2^127  +
//         -----------------------------
```

```
//          (1.11111111111111111111111)₂*2^127
//
// calculate x < N_max + ulp(N_max)/2)
//   2^130 < (1.11111111111111111111111)₂*2^127 -- THIS IS FALSE
//
// because the above calc, we choose ∞ for round to nearest, which
// is how special case #1 is done...
//   x < N_max + ulp(N_max)/2) ? N_max : ∞
```

Exercise 5.2

Using IEEE single format, here's an example where x_- and x_+ are the same distance from x…

```
// note that the significand of x is 1 bit too long to fit into a
// single-precision float... rounding options (x_ and x+) are equal
// distance away from x
x  = +(1.11111111111111111111001)₂*2^1

// calculate ulp(x)
e=2^-23  <-- machine epsilon for single-precision float
E=1    <-- exponent from normalization
ulp(x) = e * 2^E
       = 2^-23 * 2^1
       = 2^((-23)+1)     // confused? look up exponent rules
       = 2^-22

// calculate x_
x_ = trunc(x)
   = +(1.11111111111111111111100)₂*2^1

// calculate x+
x+ = x_ + ulp(x)
   = +(1.11111111111111111111100)₂*2^1 + 2^-22
   = +(1.11111111111111111111101)₂*2^1
```

If the rounding mode was set to "round to nearest", the tiebreaking logic would choose the option with the least-significant bit set to 0. In this case that'd be x_-.

Exercise 5.3

I would imagine x_- for subnormals works the same way as it does for normals -- truncation of the excess bits. However, the book doesn't really say if that's the case (see note)? Either way, if that is the case, here are some examples…

$+(0.11111111111111111111001)_2*2^{-126} \rightarrow$
$+(0.11111111111111111111100)_2*2^{-126}$
$+(0.00000000000000000000011)_2*2^{-126} \rightarrow$
$+(0.00000000000000000000001)_2*2^{-126}$

Exercise 5.4

abserr(0.1) for each of the rounding modes is (builds off of answer to Exercise 5.1)…

$// \text{ convert to bin}$
$0.1 = (0.0001100110011001100110011001...)_2$

$// \text{ normalize}$
$+(0.0001100110011001100110011001...)_2 = +(0.0001100110011001100110011001...)_2*2^0$

$= +(0.0011001100110011001100110011...)_2*2^{-1}$

$= +(0.0110011001100110011001100110...)_2*2^{-2}$

$= +(0.1100110011001100110011001100...)_2*2^{-3}$

$= +(1.1001100110011001100110011001...)_2*2^{-4}$
$// \text{ round}$
$\text{round to } -\infty \quad = x_- = +(1.10011001100110011001100)_2*2^{-4}$
$\text{round to } \infty \quad = x_+ = +(1.10011001100110011001101)_2*2^{-4}$
$\text{round to } 0 \quad = x_- = +(1.10011001100110011001100)_2*2^{-4}$
$\text{round to nearest} = x_- = +(1.10011001100110011001100)_2*2^{-4}$

$// \text{ calculate abserr}$
$\text{abserr(x) when round to } -\infty$
$= +(0.0000000000000000000000000110011\ldots)_2*2^{-4}$
$\text{abserr(x) when round to } \infty$
$= +(0.0000000000000000000000001110011\ldots)_2*2^{-4}$
$\text{abserr(x) when round to 0}$
$= +(0.0000000000000000000000000110011)_2*2^{-4}$
$\text{abserr(x) when round to nearest} = +(0.0000000000000000000000000110011\ldots)_2*2^{-4}$

Exercise 5.5

If $x > N_{max}$, abserr(x) for each of the rounding modes would be…

```
// REMEMBER THAT if x>N_max then x_=N_max and x+=∞
x_=N_max
x+=∞

round to -∞       = N_max
round to ∞        = ∞
round to 0        = N_max
round to nearest = ∞ or N_max (it depends on how far past N_max)

abserr(x) when round to -∞       = |N_max - x|
abserr(x) when round to ∞        = |∞ - x|
abserr(x) when round to 0        = |N_max - x|
abserr(x) when round to nearest = |∞ - x| OR |N_max - x|
```

Exercise 5.6

The absolute error of a subnormal number using “round to -∞” is done the same way as normalized numbers? That is, abserr(x) for some subnormal number x is...

```
abserr(x) = |x-x_|
```

As stated in Exercise 5.3, the book didn’t clearly define how to deal with rounding and subnormals.

Exercise 5.7

Again, the book never clearly defines how rounding is suppose to be done for subnormals, but just by playing around with the numbers I see that those bounds don’t hold for the first one? This is probably the wrong answer.

** SKIPPED THE REMAINDER OF CHAPTER 5 EXERCISES