Floating Point

Introduction

Int and unsigned int are approximations to the set of integers and the set of natural numbers. Unlike int and unsigned, the set of integers and the set of natural numbers is infinite. Because the set of int is finite, there is a maximum int and a minimum int.

Ints are also contiguous. That is, between the minimum and maximum int, there are no missing values.

To summarize:

- The set of valid ints is finite.
- Therefore, there is a minimum and maximum int.
- ints are also contiguous. That is, there are no missing integer values between the minimum and maximum int.

There's also another key feature of ints that doesn't appear in the set of integers. ints have an underlying representation. The representation is binary numbers. The set of integers is often represented as base 10 numerals, but is often thought of more abstractly. That is, the set is independent of its representation (i.e., we can represent the set of integers in any way we want).

What issues comes up when trying to devise a data representation for floating point numbers? It turns out these issues are more complicated than representing integers. While most people agree that UB and 2C are the ways to represented unsigned and signed integers. Representing real numbers has traditionally been more problematic.

In particular, depending on which company manufactured the hardware, there were different ways to represent real numbers, and to manipulate it. There's no particularly obvious choice of how to represent real numbers. In the mid 1980's, the need for uniform treatment of real numbers (called floating point numbers) lead to the IEEE 754 standard.

Standards are often developed to give consistent behavior. For example, when C was first being developed, what was considered a valid C program very much depended on the compiler. A program that compiled on one C compiler might not compile on another. Effectively, many different flavors of C were being created, and the need to have a standard definition of a language was seen as important.

Similar, for purposes of agreeing on results performed on floating point numbers, there was a desire to standardize the way floating point numbers were represented.

Before we get to such issues, let's think about what restrictions will have to be imposed on floating point numbers.

- Since the number of bits used to represent a floating point number is finite, there must be a maximum float and a minimum float.
- However, since real numbers are dense (i.e., between any two distinct real numbers, there's another real number), there's no way to make any representation of real numbers contiguous. Integers do not have this denseness property.

This means we need to decide which real numbers to keep and which ones to get rid of. Clearly, any number that has repeated decimals or never repeats is not something that can be represented as a floating point number.

Scientific Notation

Why do we need to represent real numbers? Of course, it's important in math. However, real numbers are important for measurements in science.

Precision vs. Accuracy

Let's define these two terms:

> **Definition** *Precision* refers to the number of significant digits it takes to represent a number. Roughly speaking, it determines how fine you can distinguish between two measurements.
>
> **Definition** *Accuracy* is how close a measurement is to its correct value.

A number can be precise, without being accurate. For example, if you say someone's height is 2.0002 meters, that is precise (because it is precise to about 1/1000 meters). However, it may be inaccurate, because a person's height may be significantly different.

In science, precision is usually defined by the number of significant digits. This is a different kind of precision than you're probably used to. For example, if you have a scale, you might be lucky to have precision to one pound. That is, the error is +/- 1/2 pound. Most people think of precision as the smallest measurement you can make.

In science, it's different. It's about the number of significant digits. For example, $1.23 * 10^{10}$ has the same precision as $1.23 * 10^{-10}$, even though the second quantity is much, much smaller than the first. It may be unusual, but that's how we'll define precision.

When we choose to represent a number, it's easier to handle precision than accuracy. I define accuracy to mean the result of a measured value to its actual value. There's not much a computer can do directly to determine accuracy (I suppose, with sufficient data, use statistical methods to determine accuracy).

Accuracy of Computations

There are two distinct concepts: the accuracy of a value that's recorded or measured, and the accuracy of performing operations with numbers.

We can't do much about the accuracy of the recorded value (without additional information). However, the hardware does perform mathematical operations reasonably accurately. The reason the computations aren't perfectly accurate is because one needs infinitely precice math, and that requires an infinite number of bits, which doesn't exist on a computer.

Since floating point numbers can not be infinite precise, there's always a possibility of error when performing computations. Floating point numbers often approximate the actual numbers. This is precisely because floating point numbers can not be infinitely precise.

In the field on computer science, numerical analysis is concerned with ways of performing scientific computations accurately on a computer. In particular, there are ways to minimize the effect of "round-off errors", errors that are due to the approximate nature of floating point representation.

Canonical Representation

When representing numbers in scientific notation, it has the following form:

```
+/- S X 10^exp
```

where **S** is the significand or the mantissa, and **exp** is the exponent. "10" is the base.

In scientific notation, there can be more than one way of writing the same value. For example, **6.02×10^{23}** is the same as **60.2×10^{22}** is the same as **602×10^{21}**.

For any number represented in this way, there are an infinite number of other ways to represent this (by moving the decimal point, and adjust the exponent).

It might be nice to have a single, consistent way of doing this, i.e. a *canonical* or standard way of doing this. And, so there is such a way.

```
+/- D.FFFF X 10^exp
```

You can write the significand as **D.FFF...** where **$1 <= D <= 9$**, and **FFF...** represents the digits after the decimal point. If you follow this restriction on **D**, then there's only one way to write a number in scientific notation. The obvious exception to this rule is representing 0, which is a special case.

You can generalize this formula for other bases than base 10. For base K (where **$K > 1$**), you write the canonical scientific notation form as:

```
+/- D.FFFF X K^exp
```

where **$1 <= D <= K - 1$**. The F's that appear in the fraction must follow the rule: **$0 <= D <= K - 1$**.

The *number of significant digits*, which is also the *precision*, is the number of digits after the radix point. We call it the *radix point* rather than the *decimal point* because decimal implies base 10, and we could be talking about any other base.

Binary Scientific Notation

As with any representation on a computer, we need to represent numbers in binary. So, this means we specialize the formula to look like:

```
+/- D.FFFF X 2^exp
```

This creates an interesting constraint on **D**. In particular, **$1 <= D <= 1$**, which means **D** is forced to be 1. We'll use this fact later on.

IEEE 754 Single Precision

IEEE 754 floating point numbers was a standard developed in the 1980s, to deal with the problem of non-standard floating point representation.

There is a standard for single precision (32 bits) and double precision (64 bits). We'll primarily focus on single precision.

Chart

Sign	Exponent	Fraction
1	8 bits	23 bits
b_{31}	**b_{30-23}**	**b_{22-0}**
X	XXXX XXXX	XXXX XXXX XXXX XXXX XXXX XXX

An IEEE 754 single precision number is divided into three parts. The three parts match the three parts of a number written in canonical binary scientific notation.

- **sign bit**

  This is **b_{31}**. If this value is 1, the number is negative. Otherwise, it's non-negative.

- **exponent**

  The exponent is excess/bias 127. Normally, one expect the excess/bias to be half the number of representations. In this case, the number of representations is 256, and half of that is 128. Nevertheless, the excess is 127. Thus, the range of possible exponents is **-127 <= exp <= 128**.

- **fraction**

  Normally, you would represent the significant (also called the *mantissa*. This would mean representing **D.FFFF...**. However, recall that for base 2, **D = 1**. Since **D** is always 1, there's no need to represent the 1. You only need to represent the bits after the radix point.

  Thus, the "1" left of the radix point is NOT explicitly represented. We call this the *hidden one*.

IEEE 754 single precision has 24 bits of precision. 23 of the bits are explicitly represented, and the additional "hidden 1" is the 24th bit.

There is something of a fallacy when we say that a IEEE single precision has 24 bits of precision. In particular, it's very much like saying that a calculator with 15 digits has 15 digits of precision. It's true that it may represent all numbers with 15 digits, but the question is whether the value is really that precise.

For example, suppose a measured number has only 3 digits of precision. There's no way to indicate this on a calculator. The calculator is prepared to have 15 digits of precision, even though that's more accurate than the number.

The same can be said about representing floating point numbers. It has 24 bits of precision, but that may not accurately represent the true number of significant bits. Unfortunately, that's the best computers can do. One could store additional information to determine exactly how many bits really are significant, but this is not usually done.

Categories

Unlike UB or 2C, floating point numbers in IEEE 754 do not all fall in the same category. IEEE 754 identifies 5 different categories of floating point numbers.

You might wonder why they do this. Here's one reason. Given the representation, as is, there would be no way to represent 0. If all the bits were 0, this would be the number $\mathbf{1.0 \times 2^{-127}}$. Although this is a small number, it's not 0.

Thus, we designate the bitstring containing all 0's to be zero.

The following is a list of categories of floating point numbers in IEEE 754.

- **zero** Because there is a sign bit, there is a positive and negative representation of 0.
- **infinity** There is also a positive and negative infinity. Infinity occurs when you divide a non-zero number by zero. For example, 1.0/0.0 produces infinity.
- **NaN** This stands for "not a number". NaN usually occurs when you do an ill-defined operation. The canonical example is dividing 0.0/0.0, which does not have a defined value.
- **denormalized numbers** These are numbers which have fewer bits of precision, and are smaller (in magnitude) than normalized numbers. We'll discuss denormalized numbers in detail momentarily.
- **normalized numbers** These are standard floating point numbers. Most bitstring patterns in IEEE 754 are normalized numbers.

How to Tell Which Category a Float is

It would be useful to know which category a given a bitstring falls in. Here's the chart.

Category	Sign Bit	Exponent	Fraction
Zero	Anything	$\mathbf{0^8}$	$\mathbf{0^{23}}$
Infinity	Anything	$\mathbf{1^8}$	$\mathbf{0^{23}}$
NaN	Anything	$\mathbf{1^8}$	Not $\mathbf{0^{23}}$
Denormalized numbers	Anything	$\mathbf{0^8}$	Not $\mathbf{0^{23}}$
Normalized numbers	Anything	Not $\mathbf{0^8}$, nor $\mathbf{1^8}$	Anything

Again, we write $\mathbf{0^8}$ to mean 0, repeated 8 times, i.e. 0000 0000.

Notice that there is a positive and negative 0.

Denormalized Numbers

Suppose that we allowed all 0's to be a normalized number (it's not though, it's really designated as zero).

A bitstring with 32 zeros would be $\mathbf{1.0 \times 2^{-127}}$. That's a pretty small value. However, we could represent numbers to get even smaller, if we do the following when the exponent is $\mathbf{0^8}$.

- Do not have a hidden 1. $\mathbf{b_{23-0}}$ would be the bits appearing after a radix point.
- Fix the exponent to -126.

Recall that the exponent is written with a bias of 127. So, you would expect that if the exponent is $\mathbf{0^8}$, this bitstring would represent the exponent -127, and not -126. However, there's a good reason why it's -126. We'll explain why in a moment. For now, let's accept the fact that the exponent is -126 whenever the exponent bitstring is $\mathbf{0^8}$.

Largest Positive Denormalized Number

What's the largest positive denormalized number? This is when the fraction is $\mathbf{1^{23}}$. It looks like:

```
S    Exp               Fraction
- --------- -----------------------------
0 0000 0000 1111 1111 1111 1111 1111 111
```

This bitstring maps to the number $\mathbf{0.(1^{23}) \times 2^{-126}}$. This number has 23 bits of precision, since there are 23 1's after the radix point.

Smallest Positive Denormalized Number

What's the smallest positive denormalizd number?

- Exponent bitstring $\mathbf{0^8}$. (All denormalized numbers have this bitstring). It's value is -126.
- The fraction is $\mathbf{(0^{22})1}$, i.e., 22 zeroes followed by a single 1.

This looks like:

```
S    Exp               Fraction
- --------- -----------------------------
0 0000 0000 0000 0000 0000 0000 0000 001
```

This bitstring pattern maps to the number $\mathbf{0.0^{22}1 \times 2^{-126}}$, which is $\mathbf{1.0 \times 2^{-149}}$. This number has 1 bit of precision. The 22 zeroes are merely place holders and do not affect the number of bits of precision.

You may not *believe* that this number has only 1 bit of precision, but it does. Consider the decimal number 123. This number has 3 digits of precision. Consider 00123. This also has 3 digits of precision. The leading 0's do not affect the number of digits of precision. Similary, if you have 0.000123, the zeroes are merely to place the 123 correctly, but are not significant digits. However, 0.01230 has 4 significant digits, because the rightmost 0 actually adds to the precision.

Thus, for our example, we have 22 zeroes followed by a 1 after the radix point, and the 22 zeroes have nothing to do with the number of significant bits.

By using denormalized numbers, we were able to make the smallest positive float to be $\mathbf{1.0 \text{ X } 2^{-149}}$, instead of $\mathbf{1.0 \text{ X } 2^{-127}}$, which we would have had if the number had been normalized.

Thus, we were able to go 22 orders of magnitude smaller, by sacrificing bits of precision.

Why -126 and not -127?

When the exponent bitstring is **0^8**, this is mapped to exponent -126. Yet, for normalized IEEE 754 single precision floating point numbers, the bias on the exponent is -127. Why is it -126 instead of -127.

To answer this question, we need to look at the smallest positive normalized number. This occurs with the following bitstring pattern

```
S    Exp              Fraction
- --------- -----------------------------
0 0000 0001 0000 0000 0000 0000 0000 000
```

This bitstring maps to **1.0×2^{-126}**.

Let's look at the two choices for the largest positive denormalized numbers.

- **$0.(1^{23}) \times 2^{-127}$** (exponent is 127)
- **$0.(1^{23}) \times 2^{-126}$** (exponent is 126---this is what's really used in IEEE 754 single precision)

Both choices are smaller than **1.0×2^{-126}**, the smallest *normalized* (In particular, notice that the number with the exponent of -126 is smaller). That's *good* because we want to avoid overlap between normalized and denormalized numbers.

Also notice that the number with the -126 as exponent is larger than the number that has -127 as exponent (both have the same mantissa/significand, and -126 is larger than -127).

Thus, by picking -126 instead of -127, the gap between the largest denormalized number and the smallest normalized number is smaller. Is this a necessary feature? Is it really necessary to make that gap small? Maybe not, but at least there's some rationale behind the decision.

Converting Normalized from Base 10 to IEEE 754

Let's convert 10.25 from base 10 to IEEE 754 single precision. Here's the steps:

- **Convert the number left of the radix point to base 2**

  Thus, 10_{10} is 1010_2.

- **Convert the number right of the radix point to base 2.**

  Thus, $.25_{10}$ is $.01_2$.

- **Add the two.**

  This results in 1010 + 0.01, which is 1010.01.

- **Write this in binary scientific notation.**

This is **1010.01 X 2^0**, which is **1.01001 X 2^3**.

- **Write this in IEEE 754 single precision.**

  This is **1010.01 X 2^0**, which is **1.01001 X 2^3**.

- Convert 3 to the correct bias. Since the bias is 127, add 127 to 3 to get 130 and convert to binary. This turns out to be 1000 0010.

- Write out the number in the correct representation

```
S    Exp               Fraction
- --------- -----------------------------
0 1000 0010 0100 1000 0000 0000 0000 000
```

  Notice that the hidden "1" is *not* represented in the fraction.

An Algorithm for Writing Positive Exponent in Excess 127

Converting 130 to binary seems a bit painful. It seems to require many steps. However, there's a fairly easy way to convert positive exponents to binary.

First, we take advantage of the following fact: **1000 0000** maps to exponent +1 in excess 127. If this were excess 128, it would map to 0. It would be nice, in fact, if it were excess 128, because then we would write out the positive number in unsigned binary, then flip the most significant bit from 0 to 1, and we'd be done. (Verify this for yourself with an example or two).

However, excess 127 and excess 128 are only off-by-one, so it's not too hard to adjust the algorithm appropriately. Here's what you do to convert positive exponents to excess 127.

- Subtract 1 from the positive exponent.
- Convert the number to unsigned binary, using 8 bits.
- Flip the MSb to 1

For example, we had an exponent of 3 in the previous example. Subtract 1 to get 2, convert to UB to get 0000 0010. Flip the MSb to get 1000 0010. That's the answer from the previous section.

Before you memorize this algorithm, you should really try to understand where it comes from.

This is where it comes from. Consider a positive exponent, **x**, represented in base 10. To convert it to excess 127, we add 127. Thus, we have **x + 127**. We can rewrite this as: **(x - 1) + 128**. This is simple algebra.

128 is 1000 0000 in binary. And we have **x - 1**, which is where the subtraction of 1 occurs. As long as **x - 1** is smaller than 128 (and it will be, since the maximum value of **x** is 128), then it's easy to add this binary number to 1000 0000.

Remember, memorization is a poor second to understanding. It's better to understand why something works than to memorize an answer. However, it's even better to understand why something works and remember it too.

Converting Denormalized from Base 10 to IEEE 754

Suppose you're asked to convert **1.1×2^{-128}** to IEEE 754 single precision. How would you do this? If you're not careful, you might think the number is normalized, and you might convert this to a normalized number using the procedure from before.

You'd get stuck trying to convert the exponent, because you'd discover the number is negative, and the number has to be non-negative when convert from base 10 (after adding the bias) to UB.

You can save yourself this hassle if you recall that the smallest, positive normalized number has an exponent of -126, and that the exponent we have is -128, which is less than -126. If you've written the number in binary scientific notation (in canonical form), and the exponent is less than -126, then you have a denormalized number.

Since -128 < -126, the number we're trying to represent is a denormalized number.

The rules for representing denormalized numbers is different from representing normalized numbers.

To represent a denormalized number, you need to shift the radix point so that the exponent is -126. In this case, the exponent must be increased by 2 from -128 to -126, so the radix point must shift left by 2.

This results in: **0.011×2^{-126}**

At this point, it's easy to convert. The exponent bitstring is **0^8**. You copy the bits after the radix point into the fraction. The sign bit is 0.

```
S    Exp             Fraction
- --------- ---------------------------
0 0000 0000 0110 0000 0000 0000 0000 000
```

No Unsigned Float

Unlike ints, there isn't an unsigned float. One reason for this may be the complicated nature of representing floating point numbers. If we get rid of the sign bit, how would we use it? Would we add one more bit to the exponent? That would make the most sense, since it sits adjacent to the exponent, but the bias would have to be changed.

We could add one more bit to the fraction. At least, that would cause the least amount of disruption. Would that one additonal bit help us in any meaninful way? On the one hand, it allows us to represent twice as many floating point numbers. On the other, it does so by adding a single bit of precision.

Perhaps through this kind of reasoning, the developers of the IEEE 754 standard felt that having an unsigned float did not make sense, and thus there is no unsigned float in IEEE 754 floating point.

Why Sign Bit, Exponent, then Fraction?

If you look at the representation for IEEE 754, you'll notice that it's sign bit, then exponent, then fraction.

Why do it in that order?

Here's a plausible explanation. Suppose you want to compare two dates. The date includes month, day, and year. You use two digits for the month, two for the day, and four for the year. Suppose you want to store the date as a string, and want to use string comparison to compare dates.

Which order should you pick?

You should pick the year, the month, and the day. Why? Because when you are doing string comparison, you compare left to right, and you want the most significant quantity to the left. That's the year.

When you look at a floating point number, the exponent is the most important, so it's to the left of the fraction.

You can also do comparisons because the exponent is written in bias notation (you *could* use two's complement, as well, although it would make the comparison only a little more complicated).

So why is the sign bit to the far left? Perhaps the answer is because that's where it appears in signed int representation. It may be unusual to have the sign in any other position.

Summary

After reading and practicing, you should be able to do the following:

- Give the names of each of the five categories of floating point numbers in IEEE 754 single precision.
- Given a 32 bitstring, determine which category the bitstring falls in.
- Given a normalized or denormalized number, write the number in canonical binary scientific notation (you can leave the exponent written in base 10).
- Given a number in base 10 or canonical binary scientific notation, convert it to an IEEE 754 single precision floating point number.
- Know what bias is used for normalized numbers.
- Know what exponent is used for denormalized numbers.
- Know what the hidden 1 is.