Floating Point Numbers

Stats 102A

Miles Chen

Department of Statistics

Week 6 Monday

UCLA

Section 1

Approximate Storage of Numbers

Real Numbers

The set of **real numbers** $\mathbb{R}$ has many definitions and constructions.

Intuitively, $\mathbb{R}$ can be thought of as all the points on an infinitely long line (the **number line** or **real line**), where integer points are equally spaced.

There are infinitely many real numbers. In fact, there are infinitely many real numbers between any two distinct real numbers.

Properties of Real Numbers

The real numbers have many properties that we take for granted when working with numbers. Some key ones are given below.

Let a, b, and c denote real numbers.

Associativity:

$$\begin{aligned}(a+b)+c &= a+(b+c)\\ (a\times b)\times c &= a\times(b\times c)\end{aligned}$$

Distributive Property:

$$a\times(b+c) = (a\times b)+(a\times c)$$

Commutative Property:

$$\begin{aligned}a+b &= b+a\\ a\times b &= b\times a\end{aligned}$$

Decimal Representation

Another property of $\mathbb{R}$ is that every real number has a **decimal representation**, i.e., any real number r can be expressed as

$$r = (-1)^s \sum_k a_k 10^k,$$

where $s \in \{0, 1\}$ determines the sign of r, $a_k \in \{0, 1, 2, \ldots, 9\}$ are the digits of r, and 10 is the base (or **radix**) of the number system.

A shorthand for the full summation representation is to represent r by its a_k coefficients, with a dot (decimal point) to denote the boundary between non-negative and negative powers of 10.

For example, the decimal number 5413.29 can be expressed as

$$5413.29 = 5 \times 10^3 + 4 \times 10^2 + 1 \times 10^1 + 3 \times 10^0 + 2 \times 10^{-1} + 9 \times 10^{-2}.$$

Scientific Notation

An alternative expression to decimal representation is **scientific notation**, i.e., any real number r can be expressed as

$$r = (-1)^s m \times 10^n,$$

where $s \in \{0, 1\}$ determines the sign of r and m and n are integers. The integer n is called the **exponent** (or **order of magnitude**), and the integer m is called the **significand** (or **mantissa**).

Typically, scientific notation is **normalized** such that the significand is $1 \leq m < 10$. That is, r is in **normalized scientific notation** if r is expressed as

$$r = (-1)^s a_0.a_1a_2a_3 \cdots \times 10^n,$$

where $a_0, a_1, a_2, a_3, \ldots \in \{0, 1, 2, \ldots, 9\}$ are digits with $a_0 \neq 0$.

Scientific Notation

For example, non-normalized scientific notation for 5413.29 could be

$$5413.29 = 541329 \times 10^{-2},$$

while normalized scientific notation is

$$5413.29 = 5.41329 \times 10^{3}.$$

Scientific notation is often useful for representing very small or very large numbers in a compact way, such as $6.02214076 \times 10^{23}$ or $9.10938356 \times 10^{-31}$.

Infinite Decimal Representation

Humans often represent real numbers and perform arithmetic calculations using decimal representation. Some numbers have finite decimal representations such as $1/2 = 0.5$, but most real numbers require infinite decimals to represent them.

All irrational numbers, such as $\sqrt{2}$, π, or e, have infinite decimal representations. Even many rational numbers, such as $1/3$, $1/7$, or $1/9$, have infinite decimal representations.

Thus, every time we write a decimal representation of a real number with infinite decimal representation on paper, it is an *approximation* to the true value.

However, using approximations has profound implications for the properties of real numbers that we take for granted.

Approximate Storage of Numbers

Computers are unable to represent real numbers with infinite precision. Computers do not have infinite storage, so they use a fixed number of bits to represent a number. This results in many numbers being represented by an approximate representation.

Computers technically store numbers in a binary representation (base 2 instead of base 10), but the effects of approximation can also be seen with finite decimal representation.

Breaking the Rules

We can begin to see some of the limitations of representing numbers approximately by looking at a simple calculator. Many simple calculators can only show 8 digits on the screen.

Examine what happens when we perform the following computation on a calculator.

$$(10 \div 3) \times 3$$

https://youtu.be/T9fhVHAj7eU

Breaking the Rules

We can begin to see some of the limitations of representing numbers approximately by looking at a simple calculator. Many simple calculators can only show 8 digits on the screen.

Examine what happens when we perform the following computation on a calculator.

$$(10 \div 3) \times 3$$

https://youtu.be/T9fhVHAj7eU

We get 9.9999999

Breaking the Rules

On the other hand, examine what happens when we perform

$$(10 \times 3) \div 3$$

https://youtu.be/zOeffxHlgCc

Breaking the Rules

On the other hand, examine what happens when we perform

$$(10 \times 3) \div 3$$

https://youtu.be/zOeffxHlgCc

We get 10

Breaking the Rules

The previous example breaks a fundamental property of the real numbers (a combination of associativity and the commutative property):

$$(10 \div 3) \times 3 \neq (10 \times 3) \div 3$$

The order in which these operations are performed does not matter for real numbers, but the theoretical properties of real numbers do not always apply with approximate representations.

Another example

$$(\sqrt{2})^2 \approx 1.9999998$$

https://youtu.be/Hai5SUYon6o

$$\sqrt{2^2} = 2$$

https://youtu.be/9_RDEZpO3VU

Breaking the Rules

Even in R, which is much more powerful than a simple calculator, the numbers are stored with an approximate binary representation of the value. This can cause unexpected results.

```
a <- 0.1
print(a)
```

```
## [1] 0.1
```

```
b <- (0.3 / 3)
print(b)
```

```
## [1] 0.1
```

```
a == b
```

Breaking the Rules

Even in R, which is much more powerful than a simple calculator, the numbers are stored with an approximate binary representation of the value. This can cause unexpected results.

```
a <- 0.1
print(a)
```

```
## [1] 0.1
```

```
b <- (0.3 / 3)
print(b)
```

```
## [1] 0.1
```

```
a == b
```

```
## [1] FALSE
```

Breaking the Rules

```
a <- 0.1 + 0.2
print(a)
```

```
## [1] 0.3
```

```
b <- 0.3
print(b)
```

```
## [1] 0.3
```

```
a == b
```

Breaking the Rules

```
a <- 0.1 + 0.2
print(a)
```

```
## [1] 0.3
```

```
b <- 0.3
print(b)
```

```
## [1] 0.3
```

```
a == b
```

```
## [1] FALSE
```

Breaking the Rules

```
s <- seq(0, 1, by = 0.1)
print(s)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
a <- s[4]
print(a)
```

```
## [1] 0.3
```

```
b <- 0.3
print(b)
```

```
## [1] 0.3
```

```
a == b
```

Breaking the Rules

```
s <- seq(0, 1, by = 0.1)
print(s)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
a <- s[4]
print(a)
```

```
## [1] 0.3
```

```
b <- 0.3
print(b)
```

```
## [1] 0.3
```

```
a == b
```

```
## [1] FALSE
```

One more example

```
a <- ((4 / 5) * 3) * (5 / 4)
a
```

```
## [1] 3
```

```
b <- (4 / 5) * (3 * (5 / 4))
b
```

```
## [1] 3
```

```
a == b
```

One more example

```
a <- ((4 / 5) * 3) * (5 / 4)
a
```

```
## [1] 3
```

```
b <- (4 / 5) * (3 * (5 / 4))
b
```

```
## [1] 3
```

```
a == b
```

```
## [1] FALSE
```

Breaking the Rules

```
options(digits = 20) # Or use print(x,digits=20)
0.1
```

```
## [1] 0.10000000000000001
```

```
0.2
```

```
## [1] 0.20000000000000001
```

```
0.1 + 0.2
```

```
## [1] 0.30000000000000004
```

```
0.3
```

```
## [1] 0.29999999999999999
```

Breaking the Rules

Another interesting example

```
x <- c(0.3, 0.4 - 0.1, 0.5 - 0.2, 0.6 - 0.3, 0.7 - 0.4)
x
```

```
## [1] 0.3 0.3 0.3 0.3 0.3
```

Question: How many of these do you think are distinct values? How many distinct values *should* there be?

This example comes from "The R Inferno" by Patrick Burns:
https://www.burns-stat.com/documents/books/the-r-inferno/

Breaking the Rules

They should all be equal, but there are *three* distinct values!

```
x <- c(0.3, 0.4 - 0.1, 0.5 - 0.2, 0.6 - 0.3, 0.7 - 0.4)
x
```

```
## [1] 0.3 0.3 0.3 0.3 0.3
```

```
unique(x) # Remove duplicate values
```

```
## [1] 0.3 0.3 0.3
```

```
print(x, digits = 20)
```

```
## [1] 0.29999999999999999 0.30000000000000004 0.29999999999999999
## [4] 0.29999999999999999 0.29999999999999993
```

Binary Representation

Rather than a decimal (base 10) representation, computers represent numbers using a **binary** (or **base 2**) representation, where each bit (or digit) is either a 0 or 1.

In binary representation, any real number r can be expressed as

$$r = (-1)^s \sum_k b_k 2^k,$$

where $s \in \{0, 1\}$ determines the sign of r, $b_k \in \{0, 1\}$ are the digits of r (in binary), and 2 is the base (or radix).

Note: When ambiguous, we will write $(r)_{10}$ to represent the decimal representation and $(r)_2$ to represent the binary representation.

Binary Representation

Any integers can be represented exactly with binary representation ($b_0 = 0$ for even integers and $b_0 = 1$ for odd integers).

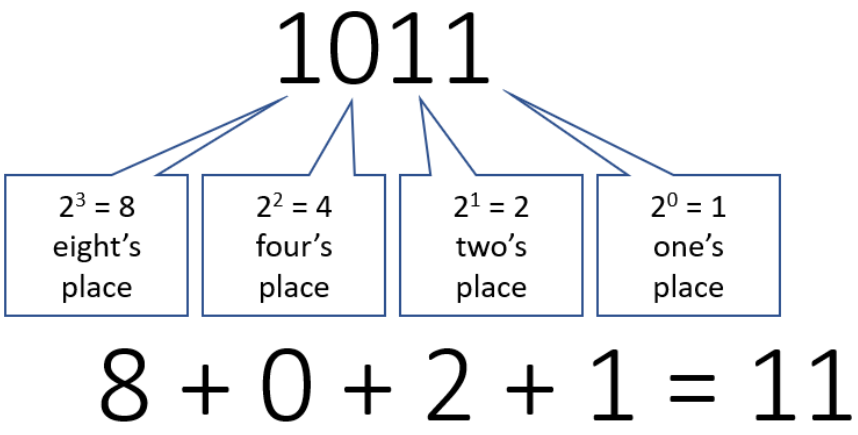
For example, the first 16 non-negative integers (using 4 bits/digits) are represented below:

Decimal	Binary	Decimal	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Binary Representation

Let's take a closer look at the number 11

In binary, it is represented with: `1011`



Finite Binary Representation

A real number has a finite binary representation if and only if it can be written as a fraction such that the denominator is a power of 2 (i.e., it has the form 2^n for some non-negative integer n). A real number has a finite *decimal* representation if and only if it can be written as a fraction such that the denominator has the form $2^m 5^n$ for some non-negative integers m and n.

For example:

Decimal	Full Binary Representation	Binary
0.5	1×2^{-1}	0.1
0.25	$0 \times 2^{-1} + 1 \times 2^{-2}$	0.01
0.125	$0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$	0.001
0.375	$0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$	0.011

Infinite Binary Representation

Just like in decimal representation, most real numbers have an infinite binary representation.

For example, the decimal number 0.1 has the binary representation given by

$$(0.1)_{10} = (0.00011001100110011\ldots)_2 = (0.0\overline{0011})_2,$$

where the bar represents a repeating pattern.

Since computers can only use finitely many bits to store numbers, the binary representation is truncated at a certain number of digits, so the decimal number 0.1 in a computer will always be an approximation to the true value.

This is why `0.1 + 0.2 == 0.3` returns `FALSE` in R: The binary approximation of 0.1 plus the binary approximation of 0.2 is not equal to the binary approximation of 0.3.

Section 2

Floating Point Representation

IEEE 754 Standard

Most programming languages, including R, use **floating point representation**, which is a binary variation on scientific notation.

Modern computers use the **IEEE 754** standard for floating point representation. This is implemented at the hardware level on the actual computer chip itself.

The IEEE 754 standard was revised in 2008, but (for educational purposes) we will discuss the original 1985 specification, which is still fully contained in the 2008 revision.

More on IEEE 754: https://en.wikipedia.org/wiki/IEEE_754

Floating Point Representation

Floating point representation expresses numbers in a normalized binary notation. The real number r is expressed as

$$r = (-1)^s 1.b_1 b_2 \dots b_m \times 2^n,$$

where $s \in \{0, 1\}$ determines the **sign** of r, $b_1, b_2, \dots, b_m \in \{0, 1\}$, and n is an integer.

As in scientific notation in base 10, the fraction part $b_1 b_2 \dots b_m$ is the **significand** (or **mantissa**) and n is the **exponent**.

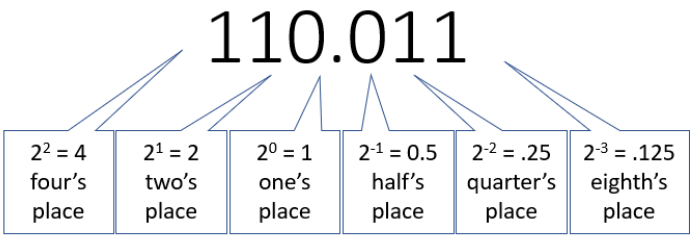
To store this representation, the computer needs to allocate bits for the sign s, the m digits of the signficand, and the exponent n (also represented in binary).

The leading 1 before the dot (technically called a **binary point**) point is not stored in the computer, as it is implied to be 1 for all numbers. If it were 0, it would no longer be in normalized scientific notation.

Example of Floating Point Representation

Consider the decimal number 6.375.

In binary, $(6.375)_{10} = (110.011)_2$.



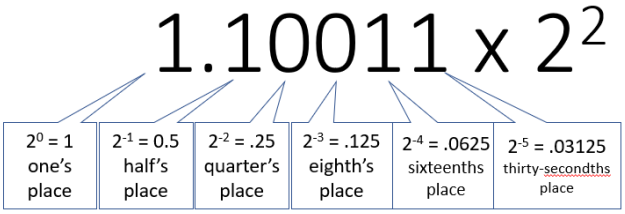
$$4 + 2 + 0 + 0 + .25 + .125 = 6.375$$

Example of Floating Point Representation

In floating point representation, we can normalize the binary representation to

$$110.011 = 1.10011 \times 2^2.$$

Just like scientific notation, moving the dot to the left increases the exponent, and moving the point to the right decreases the exponent.



$$1 + 0.5 + 0 + 0 + .0625 + .03125$$
$$= 1.59375 \times 4$$
$$= 6.375$$

Floating Point Precision

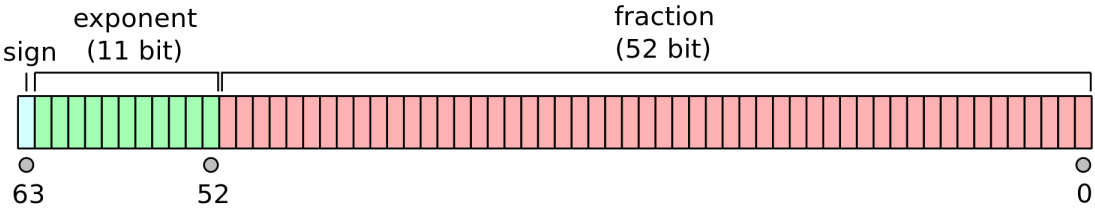
Computers use different levels of **precision** to store numbers, depending on how much storage it will use, i.e., how many bits it will allocate to store a number. Higher levels of precision will produce a better approximation, but they will require more memory.

With each level of precision, the computer allocates a certain number of bits to store the sign, the significand, and the exponent of the floating point number.

Precision	Sign	Exponent	Mantissa	Total
Single	1	8	23	32
Double	1	11	52	64
Long Double	1	15	64	80

Double Precision

R (and most other languages) uses 64-bit **double** precision, which is why non-integer numeric values in R are of type `double`.



1 bit for the sign.

11 bits for the exponent.

52 bits for the mantissa.

Source: https://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg

Exponents and Exponent Bias

The 11 bits allocated for the exponent allows us to represent $2^{11} = 2048$ different integer exponent values.

Two integers are reserved for special cases, which leaves 2046 integers to represent both positive and negative (and 0) exponents.

Thus, the possible exponents range from -1022 to 1023. The integers in this range are stored as 11-bit binary representations of the numbers 1 to 2046: 00000000001 to 11111111110.

This creates the **exponent bias** of 1023: The stored value of the exponent is off from the actual value by $1023 = 2^{10} - 1$.

To represent the actual exponent n, we add the exponent bias 1023 (or $1024 - 1$) to it and store the binary representation $(n + 1023)_2$.

Exponents and Exponent Bias

The table below shows the actual exponent represented and the corresponding stored integer, in decimal and 11-bit binary forms.

Exponent	Decimal	Binary
Special	0	00000000000
-1022	1	00000000001
-1021	2	00000000010
$\vdots$	$\vdots$	$\vdots$
1023	2046	11111111110
Special	2047	11111111111

Special Exponents

Special cases:

- The exponent of 2047, or $(11111111111)_2$, represents ∞ (`Inf`) if the significand is all zeroes and Not A Number (`NaN`) otherwise.
- The exponent of 0, or $(00000000000)_2$, is used to represent **subnormal** or **denormalized** numbers, which are numbers less than 1.0×2^{-1022} (i.e., the non-stored leading number in the floating point representation is 0 instead of the implicit 1).

Section 3

Mini-float representation

Mini-float

To fully explore the IEEE-754 binary representation, we will explore the system in detail using an analogous Mini-float system.

In our minifloat system, we will represent values with 8 bits.

1 bit for the sign.

3 bits for the exponent.

4 bits for the mantissa.

The sign bit

The first bit is the sign bit.

If the number is positive, the sign bit is 0.

If the number is negative, the sign bit is 1.

A side effect is that there are two representations of zero: Positive zero and negative zero.

3 bits for the exponent

With 3 bits for the exponent, we can represent $2^3 = 8$ unique values. In order to represent both positive and negative exponents, we use an **exponent bias**.

Binary	Decimal	With Bias
000	0	-3 (special)
001	1	-2
010	2	-1
011	3	0
100	4	1
101	5	2
110	6	3
111	7	4 (special)

In general, if n bits are used for the exponent, the exponent bias is equal to $2^{(n-1)} - 1$. For the mini-float, with 3 bits in the exponent, **our exponent bias is 3**.

As noted earlier, all zeros 000 and all ones 111 in the exponent have special meanings.

4 bits for the mantissa

In normalized binary scientific notation, we can always assume the leading digit in front of the dot is always a 1.

$$1._{\_}\ _{\_}\ _{\_}\ _{\_} \times 2^{(\text{exponent})}$$

The four bits used in the mantissa represent the digits that trail the leading 1.

Minifloat Example

What number do the following bits represent?

0 1 0 1 0 1 1 1

Minifloat Example

What number do the following bits represent?

0 1 0 1 0 1 1 1

Sign bit is 0. This is a positive number.

Exponent bits are 101

Mantissa bits are 0111

The exponent bits are 101.

101 in binary is $4 + 0 + 1 = 5$. The exponent bias is 3. So the exponent value is $5 - 3 = 2$.

The exponent bits are 101.

101 in binary is $4 + 0 + 1 = 5$. The exponent bias is 3. So the exponent value is $5 - 3 = 2$.

Mantissa bits are 0111. We can plug those into the following and we get:

The exponent bits are 101.

101 in binary is $4 + 0 + 1 = 5$. The exponent bias is 3. So the exponent value is $5 - 3 = 2$.

Mantissa bits are 0111. We can plug those into the following and we get:

1._ _ _ _ $\times 2^{(\text{exponent})}$

1.0 1 1 1 $\times 2^2$

1.0 1 1 1 $\times 2^2$

$$= \left(1 + \frac{0}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16}\right) \times 2^2$$

$$= \left(4 + 0 + 1 + \frac{1}{2} + \frac{1}{4}\right)$$

$$= 5.75$$

Another Example

Represent 2.25 in binary with the minifloat system. 2.25 represented in powers of two:

$$2.25 = 2 + 0 + \frac{0}{2} + \frac{1}{4}$$

1 0 . 0 1

Another Example

Represent 2.25 in binary with the minifloat system. 2.25 represented in powers of two:

$$2.25 = 2 + 0 + \frac{0}{2} + \frac{1}{4}$$

1 0 . 0 1

We shift the dot to the left one position, so it can be expressed in normalized binary representation:

$1 . 0 0 1 \times 2^{1}$

2.25 into mini-float

$$1\ .\ 0\ 0\ 1 \times 2^{1}$$

2.25 into mini-float

$$1\ .\ 0\ 0\ 1 \times 2^{1}$$

Sign bit is 0 for positive

Mantissa bits are what come after the dot: `0 0 1 0`. (There's a trailing zero so we have a total of 4 bits)

2.25 into mini-float

$$1 \ . \ 0 \ 0 \ 1 \times 2^{1}$$

Sign bit is 0 for positive

Mantissa bits are what come after the dot: `0 0 1 0`. (There's a trailing zero so we have a total of 4 bits)

The exponent value is 1. The bias is 3, so we need to represent $1 + 3 = 4$ in binary. 4 in binary is 100, which will be our exponent bits.

The number in binary is

$$0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0$$