Floating Points Part 2: IEEE 754 Specification

Stats 102A

Miles Chen

Department of Statistics

Week 6 Friday

UCLA

Happy Valentines Day!

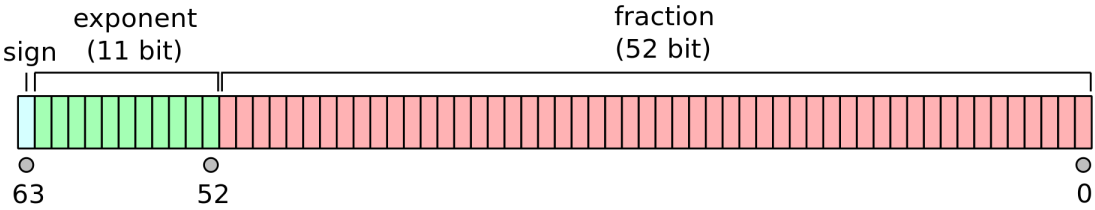
Happy Lunar New Year!

Section 1

IEEE 754

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

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?

1 0 1 0 1 0 1 0

Compare your answers with a neighbor.

Reminder: 1 bit for sign, 3 bits for exponent, 4 bits for mantissa. The exponent bias is 3.

Answer

1 0 1 0 1 0 1 0

Sign bit is 1. The value is negative.

The exponent is 010 = 2. The bias is 3, so the exponent is $2 - 3 = -1$.

The mantissa bits are 1 0 1 0. With a leading 1, it is $(1.1010)_2$

$$= (-1.1010)_2 \times 2^{-1}$$

$$-\left(1 + \frac{1}{2} + \frac{1}{8}\right) \times \frac{1}{2} = \frac{-13}{8} \cdot \frac{1}{2} = \frac{-13}{16} = -0.8125$$

Solution

Alternatively, we can 'roll' the binary point to the left because the exponent is -1:

$$(-1.1010)_2 \times 2^{-1} = (-0.11010)_2$$

$$-\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{16}\right) = \frac{-13}{16} = -0.8125$$

A problematic Mini Float example:

What number do the following bits represent?

0 0 0 0 0 0 0 0

Intuitively, this should probably represent 0.

A problematic Mini Float example:

What number do the following bits represent?

0 0 0 0 0 0 0 0

Intuitively, this should probably represent 0.

However, with our current system, the sign bit is 0, the exponent bits are 000 and the mantissa bits are 0000.

With the bias, the exponent is $0 - 3 = -3$.

The mantissa bits are 0000. With a leading 1 will become 1.0000

The number is thus:

$$(1.0000)_2 \times 2^{-3} = \frac{1}{8} = 0.125$$

Denormalized mode

It was useful for us to assume that in normalized binary scientific notation there is always a leading 1 in front of the mantissa bits. This reduces redundancy because we do not need to "waste" one of the mantissa bits which will always be a 1 anyway.

This does lead to a problem: if the leading digit in front of the binary point is always a 1, we have no way to represent the value 0.

So we must introduce **denormalized mode**. When the exponent bits are all zeros, *the digit in front of the mantissa bits is* **zero**.

To prevent a discontinuity in the numbers, in denormalized mode, *the exponent bias is one less than usual*. So for our mini-float system, the exponent bias is 2 in denormalized mode. For double-precision the exponent bias is 1022 in denormalized mode.

Denormalized mode example:

Let's revisit the case of all zeros.

0 0 0 0 0 0 0 0

Denormalized mode example:

Let's revisit the case of all zeros.

0 0 0 0 0 0 0 0

The exponent bits 000 are all zero. We are in denormalized mode. Our bias is one less than usual: 2. With a bias of 2, the exponent is -2.

The mantissa bits are 0000.

In denormalized mode, the leading digit is 0.

$$0.0000 \times 2^{-2} = 0$$

Having all zero bits represents the number 0.

Other special exponent case

When the exponent is all ones, it has a special meaning.

Infinity:

- sign bit: 0 for positive infinity, 1 for negative infinity
- exponent bits are all 1
- mantissa bits are all 0

Not a Number NaN

- sign bit: 0 or 1
- exponent bits are all 1
- mantissa bits are any combination of bits other than all 0

Smallest non-zero value we can represent:

0 0 0 0 0 0 0 1

Smallest non-zero value we can represent:

0 0 0 0 0 0 0 1

The exponent bits 000 are all zero. We are in denormalized mode. Our bias is one less than usual: 2. With a bias of 2, the exponent is -2.

The mantissa bits are 0001.

In denormalized mode, the leading digit is 0.

$$= (0.0001)_2 \times 2^{-2}$$

Smallest non-zero value we can represent:

0 0 0 0 0 0 0 1

The exponent bits 000 are all zero. We are in denormalized mode. Our bias is one less than usual: 2. With a bias of 2, the exponent is -2.

The mantissa bits are 0001.

In denormalized mode, the leading digit is 0.

$$= (0.0001)_2 \times 2^{-2}$$

$$\frac{1}{16} \times 2^{-2} = \frac{1}{64} = 0.015625$$

Second smallest non-zero value we can represent:

Second smallest non-zero value we can represent:

0 0 0 0 0 0 1 0

Second smallest non-zero value we can represent:

0 0 0 0 0 0 1 0

The exponent bits 000 are all zero. We are in denormalized mode. Our bias is one less than usual: 2. With a bias of 2, the exponent is -2.

The mantissa bits are 0010. In denormalized mode, the leading digit is 0.

$$= (0.0010)_2 \times 2^{-2}$$

Second smallest non-zero value we can represent:

0 0 0 0 0 0 1 0

The exponent bits 000 are all zero. We are in denormalized mode. Our bias is one less than usual: 2. With a bias of 2, the exponent is -2.

The mantissa bits are 0010. In denormalized mode, the leading digit is 0.

$$= (0.0010)_2 \times 2^{-2}$$

$$\frac{1}{8} \times \frac{1}{4} = \frac{1}{32} = \frac{2}{64}$$

The increment between the smallest and second smallest values is $\frac{1}{64}$

A few more denormalized values:

$$0\ 0\ 0\ 0\ 0\ 0\ 1\ 1 = \frac{3}{64}$$

$$0\ 0\ 0\ 0\ 0\ 1\ 0\ 0 = \frac{4}{64}$$

$$0\ 0\ 0\ 0\ 0\ 1\ 0\ 1 = \frac{5}{64}$$

$$0\ 0\ 0\ 0\ 0\ 1\ 1\ 0 = \frac{6}{64}$$

The increment between values is $\frac{1}{64}$

Largest denormalized value we can represent:

0 0 0 0 1 1 1 1

The exponent bits 000 are all zero. We are in denormalized mode. Our bias is one less than usual: 2. With a bias of 2, the exponent is -2.

The mantissa bits are 1111. In denormalized mode, the leading digit is 0.

$$= (0.1111)_2 \times 2^{-2}$$

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

$$= \frac{15}{16} \cdot \frac{1}{4} = \frac{15}{64}$$

Smallest Normalized value we can represent:

0 0 0 1 0 0 0 0

The exponent bits are 001. This is the smallest exponent we can have that is normalized. Our bias is the usual 3. With a bias of 3, the exponent is -2.

The mantissa bits are 0000. The leading digit is 1 (as usual).

$$= (1.0000)_2 \times 2^{-2}$$

$$= 1 \times 2^{-2} = \frac{1}{4} = \frac{16}{64}$$

The increment is still $\frac{1}{64}$

Second Smallest Normalized value we can represent:

0 0 0 1 0 0 0 1

The exponent bits are 001. Our bias is the usual 3. With a bias of 3, the exponent is -2.

The mantissa bits are 0001. The leading digit is 1 (as usual).

$$= (1.0001)_2 \times 2^{-2}$$

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

$$= \frac{17}{16} \cdot \frac{1}{4} = \frac{17}{64}$$

The increment between the smallest and second smallest normalized value is still $\frac{1}{64}$

A few more values:

$$0\ 0\ 0\ 1\ 0\ 0\ 1\ 0 = \frac{18}{64}$$

$$0\ 0\ 0\ 1\ 0\ 0\ 1\ 1 = \frac{19}{64}$$

$$0\ 0\ 0\ 1\ 0\ 1\ 0\ 0 = \frac{20}{64}$$

$$0\ 0\ 0\ 1\ 0\ 1\ 0\ 1 = \frac{21}{64}$$

$$0\ 0\ 0\ 1\ 0\ 1\ 1\ 0 = \frac{22}{64}$$

The increment between values is $\frac{1}{64}$

Largest value with exponent bits 001:

0 0 0 1 1 1 1 1

The exponent bits are 001. Our bias is the usual 3. With a bias of 3, the exponent is -2.

The mantissa bits are 1111. The leading digit is 1 (as usual).

$$= (1.1111)_2 \times 2^{-2}$$

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

$$= \left(1 + \frac{15}{16}\right) \cdot \frac{1}{4} = \frac{31}{16} \cdot \frac{1}{4} = \frac{31}{64}$$

Next value

0 0 1 0 0 0 0 0

The exponent bits are 010, which represents 2. Our bias is 3. The exponent is $2 - 3 = -1$.

The mantissa bits are 0000. The leading digit is 1.

$$= (1.0000)_2 \times 2^{-1}$$

$$= 1 \times 2^{-1} = \frac{1}{2} = \frac{32}{64}$$

The increment is still $\frac{1}{64}$

Next value

0 0 1 0 0 0 0 1

The exponent bits are 010, which represents 2. Our bias is 3. The exponent is $2 - 3 = -1$.

The mantissa bits are 0001. The leading digit is 1.

$$= (1.0001)_2 \times 2^{-1}$$

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

$$= \frac{17}{16} \cdot \frac{1}{2} = \frac{17}{32} = \frac{34}{64}$$

The increment is now $\frac{2}{64} = \frac{1}{32}$!!

Next value

0 0 1 0 0 0 1 0

The exponent bits are 010, which represents 2. Our bias is 3. The exponent is $2 - 3 = -1$.

The mantissa bits are 0010. The leading digit is 1.

$$= (1.0010)_2 \times 2^{-1}$$

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

$$= \frac{9}{8} \cdot \frac{1}{2} = \frac{9}{16} = \frac{18}{32}$$

The increment remains $\frac{1}{32}$

A few more values:

$$0\ 0\ 1\ 0\ 0\ 0\ 1\ 0 = \frac{18}{32}$$

$$0\ 0\ 1\ 0\ 0\ 0\ 1\ 1 = \frac{19}{32}$$

$$0\ 0\ 1\ 0\ 0\ 1\ 0\ 0 = \frac{20}{32}$$

$$0\ 0\ 1\ 0\ 0\ 1\ 0\ 1 = \frac{21}{32}$$

$$0\ 0\ 1\ 0\ 0\ 1\ 1\ 0 = \frac{22}{32}$$

The increment between values is $\frac{1}{32}$

Largest value with exponent bits 010:

0 0 1 0 1 1 1 1

The exponent bits are 010. With a bias of 3, the exponent is $2 - 3 = -1$.

The mantissa bits are 1111. The leading digit is 1.

$$= (1.1111)_2 \times 2^{-1}$$

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

$$= \left(1 + \frac{15}{16}\right) \cdot \frac{1}{2} = \frac{31}{16} \cdot \frac{1}{2} = \frac{31}{32}$$

Next value:

0 0 1 1 0 0 0 0

The exponent bits are 011 = 3. With a bias of 3, the exponent is $3 - 3 = 0$.

The mantissa bits are 0000. The leading digit is 1.

$$= (1.0000)_2 \times 2^0$$

$$= 1 = \frac{32}{32}$$

Next value:

0 0 1 1 0 0 0 1

The exponent bits are 011 = 3. With a bias of 3, the exponent is $3 - 3 = 0$.

The mantissa bits are 0001. The leading digit is 1.

$$= (1.0001)_2 \times 2^0$$

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

$$= \frac{17}{16}$$

The increment between values is $\frac{1}{16}$

Values:

$$0\ 0\ 1\ 1\ 0\ 0\ 0\ 0 = \frac{16}{16} = 1$$

$$0\ 0\ 1\ 1\ 0\ 0\ 0\ 1 = \frac{17}{16}$$

$$0\ 0\ 1\ 1\ 0\ 0\ 1\ 0 = \frac{18}{16}$$

$$0\ 0\ 1\ 1\ 0\ 0\ 1\ 1 = \frac{19}{16}$$

$$\ldots$$

$$0\ 0\ 1\ 1\ 1\ 1\ 1\ 1 = \frac{31}{16}$$

The increment between values is $\frac{1}{16}$

Values:

$$0\ 1\ 0\ 0\ 0\ 0\ 0\ 0 = \frac{16}{8} = 2$$

$$0\ 1\ 0\ 0\ 0\ 0\ 0\ 1 = \frac{17}{8}$$

$$0\ 1\ 0\ 0\ 0\ 0\ 1\ 0 = \frac{18}{8}$$

$$0\ 1\ 0\ 0\ 0\ 0\ 1\ 1 = \frac{19}{8}$$

$\ldots$

$$0\ 1\ 0\ 0\ 1\ 1\ 1\ 1 = \frac{31}{8}$$

The increment between values is $\frac{1}{8}$

Values:

$$0\ 1\ 0\ 1\ 0\ 0\ 0\ 0 = \frac{16}{4} = 4$$

$$0\ 1\ 0\ 1\ 0\ 0\ 0\ 1 = \frac{17}{4}$$

$$0\ 1\ 0\ 1\ 0\ 0\ 1\ 0 = \frac{18}{4}$$

$$0\ 1\ 0\ 1\ 0\ 0\ 1\ 1 = \frac{19}{4}$$

$$\ldots$$

$$0\ 1\ 0\ 1\ 1\ 1\ 1\ 1 = \frac{31}{4}$$

The increment between values is $\frac{1}{4}$

Values:

$$0\ 1\ 1\ 0\ 0\ 0\ 0\ 0 = \frac{16}{2} = 8$$

$$0\ 1\ 1\ 0\ 0\ 0\ 0\ 1 = \frac{17}{2}$$

$$0\ 1\ 1\ 0\ 0\ 0\ 1\ 0 = \frac{18}{2}$$

...

$$0\ 1\ 1\ 0\ 1\ 1\ 1\ 1 = \frac{31}{2} = 15.5$$

$$0\ 1\ 1\ 1\ 0\ 0\ 0\ 0 = +\infty$$

The increment between values is $\frac{1}{2}$

Largest value before infinity is 15.5

Section 2

Double Precision:

Smallest positive non-zero value:

- sign bit 0
- exponent: eleven 0-bits. Denormalized mode. Bias is 1 less than normal: 2^{-1022}
- Mantissa: 51 0-bits followed by a 1-bit. $= 2^{-52}$

$$= (0.0000...0001)_2 \times 2^{-1022} = 1 \times 2^{-1074}$$

```
2 ^ -1074
```

```
## [1] 4.940656e-324
```

If we ask for anything smaller, we get 0.

```
2 ^ -1075
```

```
## [1] 0
```

Smallest normalized value:

- sign bit 0
- exponent: ten 0-bits followed by a 1-bit. 00000000001 $= 1$. Bias is 1023. Exponent is $1 - 1023 = -1022$
- Mantissa: 52 0-bits.

$$= (1.0000...0000)_2 \times 2^{-1022} = 1 \times 2^{-1022}$$

```
2 ^ -1022
```

```
## [1] 2.2250738585072014e-308
```

This is known as double xmin

```
.Machine$double.xmin
```

```
## [1] 2.2250738585072014e-308
```

Second Smallest normalized value:

- sign bit 0
- exponent: ten 0-bits followed by a 1-bit. Bias is 1023. Exponent is $1 - 1023 = -1022$
- Mantissa: fifty-one 0-bits followed by a 1-bit. $= 2^{-52}$

$$= (1.0000...0001)_2 \times 2^{-1022}$$

```
x = 2 ^ -1022
print(x)
```

```
## [1] 2.2250738585072014e-308
```

```
y = (1 + 2 ^ -52) * 2 ^ -1022
print(y)  # x and y are distinct values
```

```
## [1] 2.2250738585072019e-308
```

Machine Epsilon

What is the smallest value you can add to 1 and have it be distinct from 1?

2^{-52}

```
2 ^ -52
```

```
## [1] 2.2204460492503131e-16
```

This is known as Machine Epsilon

```
.Machine$double.eps
```

```
## [1] 2.2204460492503131e-16
```

Machine Epsilon

We can add Machine Epsilon to 1, and the computer will recognize it as a distinct value from 1.

```
x = 1
print(x)
```

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

```
y = 1 + 2 ^ -52
print(y)
```

```
## [1] 1.0000000000000002
```

```
x == y  # x and y are distinct values
```

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

Machine Epsilon

If we try to add something smaller than Machine Epsilon to 1, the computer will not recognize it as a value different from 1.

```
a = 1
print(a)
```

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

```
b = 1 + 2 ^ -53
print(b)
```

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

```
a == b  # a and b are the same value to the computer
```

```
## [1] TRUE
```

1 in IEEE 754:

The number 1:

- has 0 sign bit
- has exp = 0. The bias is 1023, so the exponent in bits will be 1023. This is one 0-bit followed by ten 1-bits. `01111111111` $(512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1)$
- has all zero bits in the mantissa

`0 01111111111 0000000000000000000000000000000000000000000000000000`

$$= (1.0000...0000)_2 \times 2^0$$

1 plus Machine Epsilon

The number 1 + machine epsilon $\approx 1.0000000000000002$:

- has 0 sign bit
- has exp = 0. The bias is 1023, so the exponent in bits will be 1023. This is one 0-bit followed by ten 1-bits. `01111111111`
- has all 0-bits in the mantissa, except the last bit is 1

```
0 01111111111 0000000000000000000000000000000000000000000000000001
```

$$= (1.0000...0001)_2 \times 2^0 = (1 + 2^{-52}) \times 2^0$$

This is why it is not possible to add a number smaller than 2^{-52} to 1. There are not enough bits to represent a value between 1 and the quantity $(1 + 2^{-52})$.

Thought experiment

Imagine running the following infinite loop.

```
x <- 0
while (TRUE) {
  x <- x + 1
}
```

You let your computer run for a year and come back to check on the value of x. What value will x be?

Thought experiment

Imagine running the following infinite loop.

```
x <- 0
while (TRUE) {
  x <- x + 1
}
```

You let your computer run for a year and come back to check on the value of x. What value will x be?

x will not be "infinity" or `.Machine$double.xmax`. Instead, it will get stuck at 2^{53}

Machine Epsilon is relative

As the numbers in floating point get larger, the size of the increment gets larger as well. There eventually comes a point where the increment has size 2 and adding the whole number 1 can no longer be recognized as producing a different value.

```
x <- 9007199254740990; print(x)
```

```
## [1] 9007199254740990
```

```
x <- x + 1; print(x)
```

```
## [1] 9007199254740991
```

```
x <- x + 1; print(x)
```

```
## [1] 9007199254740992
```

```
x <- x + 1; print(x) # adding 1 does not change the value
```

```
## [1] 9007199254740992
```

Machine Epsilon

When x is equal to 2^{53}, the step size is now 2.

```
x <- 9007199254740992; print(x)
```

```
## [1] 9007199254740992
```

```
x <- x + 1; print(x) # We can't add 1
```

```
## [1] 9007199254740992
```

```
x <- x + 2; print(x) # But we can add 2
```

```
## [1] 9007199254740994
```

Largest Possible double value

- has 0 sign bit
- has exponent bits: ten 1-bits followed by one 0-bit. It cannot be all 1-bits because that would be infinity. 1111111110 $(1024 + 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 = 2046)$ With the bias of 1023, the exponent is 1023.
- has all 1-bits in the mantissa

0 11111111110 1111111111111111111111111111111111111111111111111111

$$= (1.1111...1111)_2 \times 2^{1023} = (1 + 2^{-1} + 2^{-2} + 2^{-3} + ... + 2^{-51} + 2^{-52}) \times 2^{1023}$$

```
a <- sum(2 ^ (0:-52)) * 2 ^ 1023; print(a)
```

```
## [1] 1.7976931348623157e+308
```

```
b <- .Machine$double.xmax; print(b)
```

```
## [1] 1.7976931348623157e+308
```

```
a == b
```

```
## [1] TRUE
```

Machine double xmax

The largest value we can represent that is not infinity is known as Double Max.

$$(1 + 2^{-1} + 2^{-2} + 2^{-3} + ... + 2^{-51} + 2^{-52}) \times 2^{1023} \approx 1.7976931348623157 \times 10^{308}$$

Any value larger than this will be seen as "infinity" by the computer. At this point, the increment size is 2^{970}. If we add anything smaller than that, the computer does not recognize a difference. If we add this increment or anything larger, the number becomes infinity.

```
a <- sum(2 ^ (0:-52)) * 2 ^ 1023; print(a)
```

```
## [1] 1.7976931348623157e+308
```

```
x <- a + 2^969; print(x)
```

```
## [1] 1.7976931348623157e+308
```

```
a == x
```

```
## [1] TRUE
```

```
a + 2^970
```

Checking equality with `all.equal()`

As we saw earlier, because values are approximate representation and not real numbers, certain checks for equality fail.

```
(0.1 + 0.2) == 0.3
```

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

One alternative is to use the value `all.equal()`. This will return `TRUE` for values that are nearly equal. By default, the tolerance is set to be the square-root of Machine Epsilon: $\sqrt{2^{-52}} = 2^{-26} \approx 1.5 \times 10^{-8}$. Values that are within this tolerance will be seen as "nearly equal."

```
a <- (0.1 + 0.2)
b <- 0.3
all.equal(a, b)
```

```
## [1] TRUE
```

all.equal()

```
a <- 1; print(a)
```

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

```
b <- 1 + 2^-52; print(b)
```

```
## [1] 1.0000000000000002
```

```
a == b
```

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

```
all.equal(a, b) # nearly equal, all.equal returns true
```

```
## [1] TRUE
```

all.equal()

```
a <- 1; print(a)
```

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

```
b <- 1 + 2^-26; print(b)
```

```
## [1] 1.0000000149011612
```

```
all.equal(a, b) # still seen as equal
```

```
## [1] TRUE
```

```
c <- 1 + 2^-25; print(c)
```

```
## [1] 1.0000000298023224
```

```
all.equal(a, c) # no longer TRUE
```

```
## [1] "Mean relative difference: 2.9802322387695312e-08"
```

all.equal()

When you use `all.equal()`, it returns TRUE when the values are nearly equal, but returns a charater string when they are not.

Sometimes you just want `TRUE` or `FALSE`. This can be accomplished by wrapping `all.equal()` with the function `isTRUE()`.

Use: `isTRUE(all.equal(a, b))`

```
a <- 1; print(a)
```

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

```
c <- 1 + 2^-25; print(c)
```

```
## [1] 1.0000000298023224
```

```
isTRUE(all.equal(a, c))
```

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