

ARITHMETIC in R

Introduction to data science (DSC 105) Fall 2024

Marcus Birkenkrahe (pledged)

September 25, 2024

Contents

1	What will you learn?	3
2	Please Excuse My Dear Aunt Sally	4
3	Practice: Formula Translator	5
3.1	Formula translator I	6
3.2	Formula translator II	7
3.3	Solution II	7
3.4	Formula Translator V	8
4	Mathematical functions	8
5	Logarithmic transformation	9
6	Logarithm rules	13
7	Exponential function	14
8	Practice: logarithms and constants	15
9	Practice: exp vs. log	16
10	Practice: mathematical constants	17
11	E-notation	19
12	E-xamples	20

13 Practice: e-notation	20
14 Math help in R	22
15 Special numbers	23
16 Practice: special numbers	23
17 Special functions	25
18 Practice: special functions	25
19 Logical values and operators	26
20 Practice: logical values	26
21 Logical operators	28
22 Practice: logical operators	29
23 Concept summary	31
24 Code summary	32
25 References	32



1 What will you learn?



Figure 1: The Arithmetic 1492-94 Fresco Palazzi Pontifici, Vatican Borgia Apartment

- Perform basic numerical operations
- Translate complex mathematical formulas
- Use logarithms and exponentials
- Brush up on mathematical E-notation

- Know R's special numbers
- Understand logical values and operators

Image: The Arithmetic 1492-94 Fresco Palazzo Pontificio, Vatican Borgia Apartment (Wikipedia).

Sources: Some material for this lesson comes from Davies (2016) and Matloff (2020). These and other sources have been important to me in preparing this course and getting into R in the first place. Check them out for a more systematic treatment of R. There is also a more philosophical, personal view on my use of sources in the Wiki for the 2020 version of this course.

What is this? When we say "Arithmetic", we don't mean that we "study" numbers but that we use them to perform computations. After this section, you'll be able to perform any arithmetic operation using R.

We will look at *operators* first, then at simple but important functions that occur again and again, especially in *statistics*.

How can you learn better? This presentation consists mostly of text and code chunks. Because this is dry stuff, I urge you (both if you hear this in class, and if you work through this on your own) to open an R session on the side and type along - this will build muscle memory and keep you entertained, too! Another trick, which you will find in Matloff's excellent beginner's tutorial, is to make your own little exercises by varying the instructions. A third way is to go through the lecture and create your own Emacs Org-mode notebook with R code blocks.

2 Please Excuse My Dear Aunt Sally

Operator order:

1. **P**arentheses: `()`
2. **E**xponentiation: `^` or `**`
3. **M**ultiplication: `*`
4. **D**ivision: `/`
5. **A**ddition: `+`
6. **S**ubtraction: `-`

Tip: You can check identity in R with the `identical` function.

In R, standard mathematical rules apply. The order of operators is as usual - left to right, parentheses, exponents, multiplication, division, addition, subtraction (PEMDAS = Please Excuse My Dear Aunt Sally) mnemonic).

The operators `^` and `**` for exponentiation are identical, though `^` is more common. You can check that in the R console with the `identical` function - the result should be `TRUE` (this is a truth or Boolean value - more on this below) - see figure:

```
> 2**3
[1] 8
> 2^3 # same as 2 * 2 * 2
[1] 8
> 2**3 # same as 2 * 2 * 2
[1] 8
> identical(2^3, 2**3) # checking that these are indeed the same
[1] TRUE
```

3 Practice: Formula Translator



- For the practice exercises, create a new Org-mode file.
- Make sure your file contains this in the top line:

```
#+PROPERTY: header-args:R :session *R* :results output
```

- If you enter this line for the first time, run it with `C-c C-c`.

3.1 Formula translator I

- What is the result of this expression?

$$24 + 6/3 \times 5 \times 2^3 - 9 \quad (1)$$

- Why?

```
2**3 * ((6/3 * 5)) + (24 - 9)
```

```
[1] 95
```

- Why? Remember the PEMDAS order:

```
2**3 = 2^3 = 8
6/3 = 2
2 * 5 * 8 = 80
24 + 80 = 104
104 - 9 = 95
```

- Instead of `^` you can use `**`
- R code:

```
24 + 6/3 * 5 * 2**3 - 9 # as written
2**3 * ((6/3 * 5)) + (24 - 9) # as computed
```

```
[1] 95
[1] 95
```

3.2 Formula translator II

$$10^2 + \frac{3 \times 60}{8} - 3 \quad (2)$$

$$\frac{5^3 \times (6 - 2)}{61 - 3 + 4} \quad (3)$$

$$2^{2+1} - 4 + 64^{-2^{2.25 - \frac{1}{4}}} \quad (4)$$

$$\left(\frac{0.44 \times (1 - 0.44)}{34} \right)^{\frac{1}{2}} \quad (5)$$

- Compute the expressions (2)-(5)
- You need parentheses in the exponent
- Note: -2 is interpreted as -1 * 2
- Use your Org-mode file so that you can fix errors more easily!

3.3 Solution II

$10^2 + \frac{3 \times 60}{8} - 3$	<hr/> <pre>R> 10^2+3*60/8-3 [1] 119.5</pre> <hr/>
$\frac{5^3 \times (6 - 2)}{61 - 3 + 4}$	<hr/> <pre>R> 5^3*(6-2)/(61-3+4) [1] 8.064516</pre> <hr/>
$2^{2+1} - 4 + 64^{-2^{2.25 - \frac{1}{4}}}$	<hr/> <pre>R> 2^(2+1)-4+64^((-2)^(2.25-1/4)) [1] 16777220</pre> <hr/>
$\left(\frac{0.44 \times (1 - 0.44)}{34} \right)^{\frac{1}{2}}$	<hr/> <pre>R> (0.44*(1-0.44)/34)^(1/2) [1] 0.08512966</pre> <hr/>

- You need parentheses in the exponent

- -2 is interpreted as $-1 * 2$

When you use R, you'll often have to translate a formula into code. Consider the formulas above, which seem pretty complicated: the only trick here is that you often need to use parentheses, e.g. around calculations in the exponent, or when calculating with negative numbers in eq. (4), because the number -2 e.g. is interpreted by R as the operation $-1 * 2$.

3.4 Formula Translator V

- What does $(-1)^{(1/2)}$ return?
- Use a function to compute this last command
- Can you explain the result?
- Solution in R:

```
(-1)^(1/2)
sqrt(-1)
```

```
[1] NaN
```

```
[1] NaN
```

Complex numbers? The "NaN" result, which is also "The Math Problem That Broke the Westworld Simulation" (the 2019 AI TV mini-series). Basically, R will hand you a "Not A Number" whenever you try to, e.g. take the square root of a negative number (try `sqrt(-1)` or $(-1)^{(1/2)}$). We won't need complex numbers in this course, but (of course) there are functions to handle them (see here or run `?complex`).

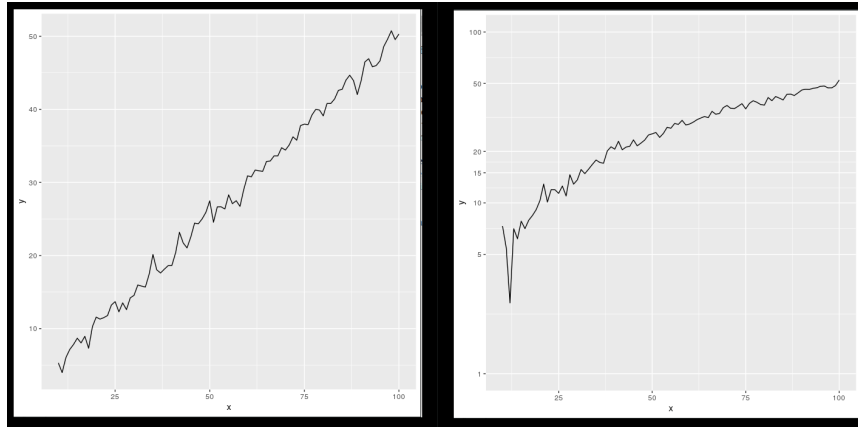
4 Mathematical functions

../img/4_maths1.gif

- `?sqrt` ($\sqrt{}$)
- `?log10` (\log_{10})
- `?exp` (e)
- `?pi` (π)

Do you know how to compute these without library functions?¹
 Do you remember how these functions are defined?²
 Are logarithm, exponential function and π connected somehow?³

5 Logarithmic transformation



Some examples:

$$\log_{10}(1 * 10^7) = 7 \quad \log_{10}(x) = \log(x, b=10) = \log(x, 10)$$

$$\log_{10}(100) = 2, \log_{10}(1000) = 3, \log_{10}(1e3) = \log_{10}(1 * 10^3) = 3$$

$$\log(1) = 0, \log_{10}(1) = 0$$

¹I've recently been reminded through this article how important it may be to be able to do computations without the help of machines. Here are 4 ways to compute `sqrt` in C (though not very fast). In general: 1) using logarithms and exponentials ($\text{sqrt}(x) = e^{0.5 \times \ln(x)}$), 2) using successive approximate numerical methods like Newton's iteration, 3) using modified long division (prime factorization), 4) looking it up in a table (source: quora.com)

²The square root of x is the number that returns x when the square root is squared: For example, $\text{sqrt}(4) = \text{sqrt}(2^2) = 2$. The logarithm of x to the base b is the power of b that returns x : $\log_{10}(100) = \log_{10}(10^2) = 2$, and $\log_2(4) = \log_2(2^2) = 2$. The exponential (power) function is the inverse of the logarithm: $\log_{10}(10^1) = 1$ ($b=10$), $\log(e) = \log_e(e) = \log_e(e^1) = 1$ ($b=e$), and $\log_2(2^1) = 1$ ($b=2$).

³Logarithm and exponential are the inverse of one another, and the exponential function e and π are connected via this beautiful, mysterious formula by Euler: $e^{i\pi} + 1 = 0$ or more general $e^{ix} = \cos(x) + i\sin(x)$, where i is the imaginary unit, $i^2 = -1$. Used in physics and engineering for example to analyse oscillatory systems.

$\log(x=100, b=100) = 1$, $\log(4.583, 4.583) = 1$

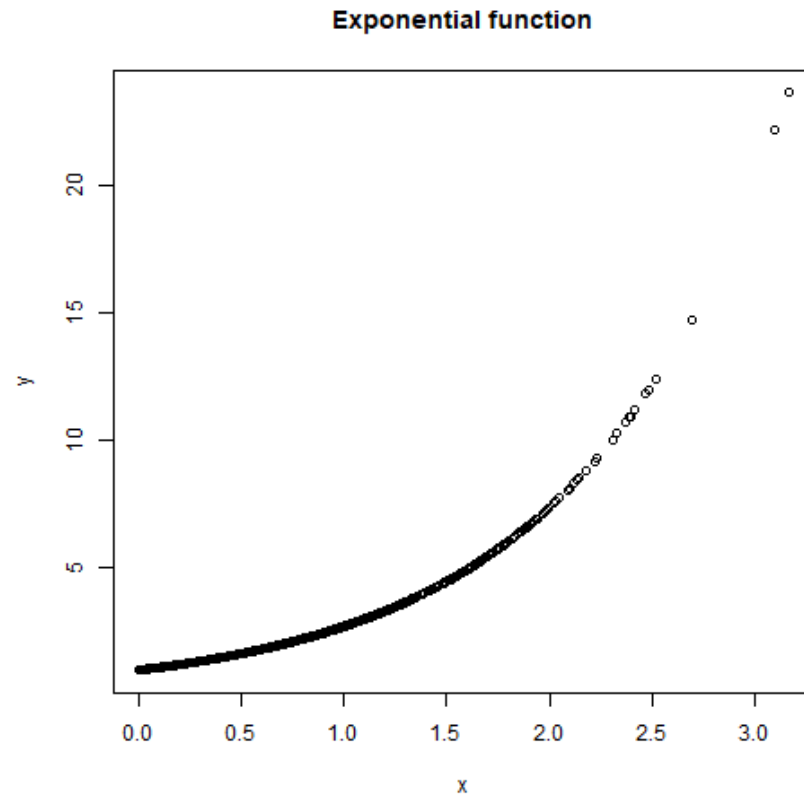
$\log(x=100, b=10) = 2$, $\log(b=10, x=100) = 2$

It is often necessary to transform numerical data, e.g. transforming data using the logarithm leading e.g. from the left to the right graph in the figure. As you can see, this transformation leads to a **compression** of the **y**-values, so that more of these values can be shown. It's a semi-logarithmic transformation (only one axis).

The *logarithm* of a number **x** is always computed using a *base b*. In the diagram, **b=10**, the numbers on the **x** axis were transformed using the `log()` function, the logarithm with base 10. The logarithm of **x=100** to the base 10 is 2, because $10^2 = 100$. In R, `log(x=100, b=10) = 2` (try this yourself!).

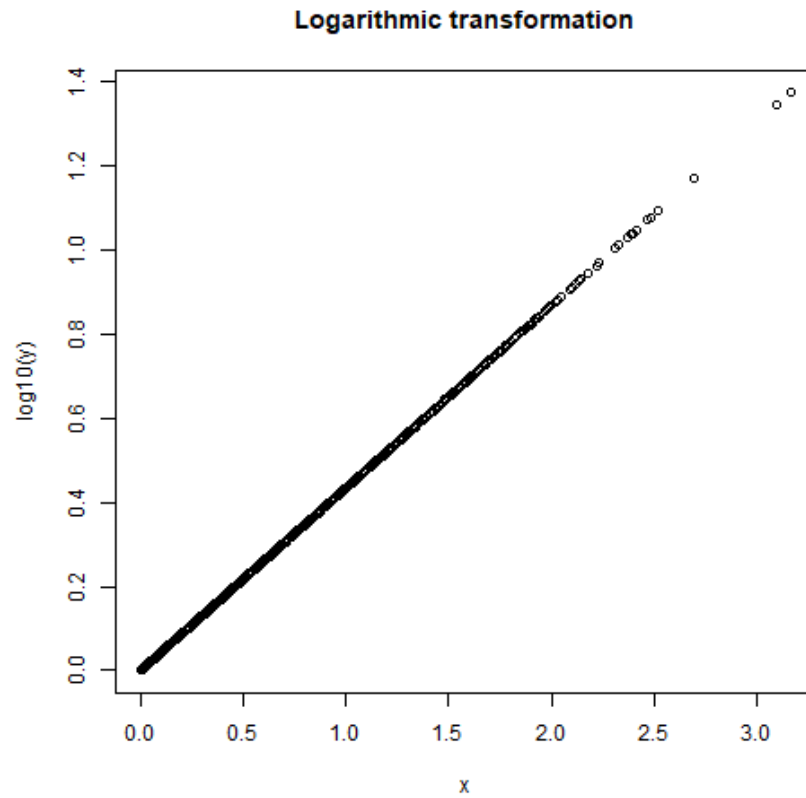
Let's make such a graph - it's not hard in R:

```
x <- abs(rnorm(1000))
y <- exp(x) # f(x)
plot(x,y)
title("Exponential function")
```



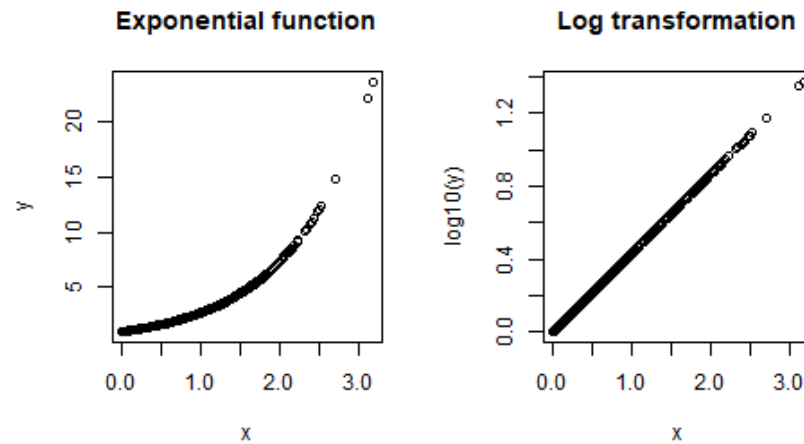
And now the logarithmic transformation:

```
plot(x,log10(y))  
title("Logarithmic transformation")
```



But to see what's going on you need to see them next to one another:

```
par(mfrow=c(1,2),pty='s')  
plot(x,y,main="Exponential function")  
plot(x,log10(y),main="Log transformation")
```



Notice that the logarithm of x values between 0 and 1 is negative, and that I take the absolute value because the logarithm of negative values is not defined and leads to NaN.

```
r <- head(abs(rnorm(1000)))
r
log(r)
```

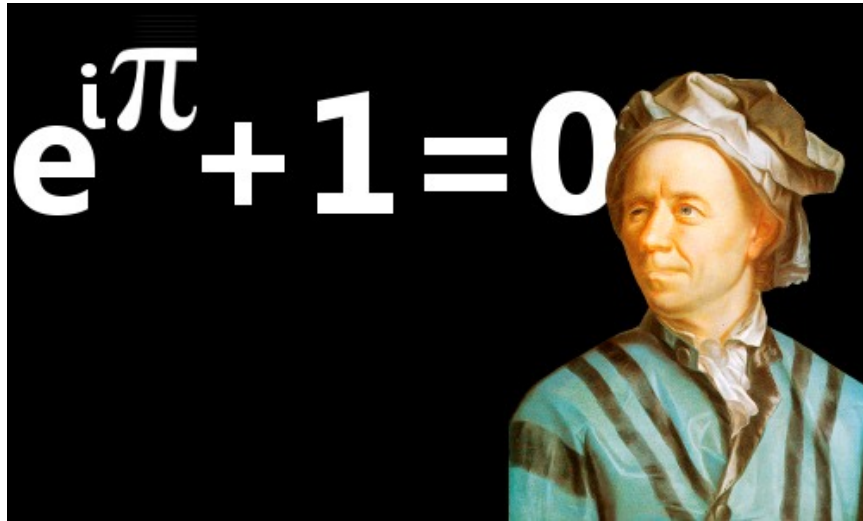
```
[1] 1.769035718444387 0.038109184381543 0.522720146917154 1.101925351534558
[5] 2.706824672249167 0.601501610877056
[1] 0.5704346062934144 -3.2672999660167292 -0.6487090500666024
[4] 0.0970589693569794 0.9957762402483749 -0.5083260654711454
```

6 Logarithm rules

../img/4_rules.gif

- Argument x and base b must be positive
- For all x : $\log_x(x) = \log(x, b=x) = 1$ since only $x^1 = x$
- For all b : $\log_b(1) = \log(x=1, b) = 0$ since $b^0 = 1$

7 Exponential function



- In \mathbb{R} , $\log(x)$ implies $b = e \approx 2.7182$
- In mathematics, the *Euler constant* e is as magical as the other mysterious constants π , 0 , 1 and i (the imaginary unit). There are different ways to arrive at its value of approximately 2.718282.
- The Wikipedia entry on e contains some fun stuff for nerds (here). Apparently, *Steve Wozniak* computed e to 116,000 digits on an "ancient" Apple II computer in 1981!
- For now, we only care about the fact that e is the base of the natural logarithm, denoted as \ln or $\log_e(x)$.

8 Practice: logarithms and constants



$$e^{\ln x} = \ln(e^x) \text{ or } \exp(\log(x)) = \log(\exp(x)) = x$$

1. Compute the log of 10,000,000 to base 10 in R
2. Enter `log10(10,000,000)` in R. What's going on?
3. Find the logarithm to base 10 for 10,000,010.
4. Why is the result the same as before?
5. Tip: enter `log10(10000100)`
6. Solution:

```
log10(10000000) # 7 because 10,000,000 = 10^7
log(10000000,10) # same thing
```

```
log10(10000010) # still 7
log10(10000100) # 7.000004 (six digits is default)
```

```
options(digits=10) # change digit formatting default
log10(10000010) # 7.000000434
```

```

[1] 7
[1] 7
[1] 7.00000043429426
[1] 7.00000043429231
[1] 7.000000434

```

See figure below:

1. The error in the first line results from the fact that in R functions, the comma separates arguments, so it looks to R as if 3 arguments were provided where only one is required, because, unlike the function `log()`, `log10()` already has a fixed base `b=10`. This is fixed in the next line.
2. The trouble with the seemingly identical results of `log10(10000010)` and `log10(10000000)` lies in the suppression of digits. This can be fixed with the `options()` utility function, which we met in an earlier lecture. After setting `options(digits=10)`, the missing numbers appear.
3. Typing `log10(10000100)` would have revealed the problem, because this result can be shown with the default number of digits (7).

```

> log10(10,000,000)
Error in log10(10, 0, 0) : 3 arguments passed to 'log10' which requires 1
> log10(10000000)
[1] 7
> log10(10000010)
[1] 7
> log10(10000100)
[1] 7.000004
> options(digits=10)
> log10(10000010)
[1] 7.000000434
↵ log10(10000100)
[1] 7.000004343

```

9 Practice: exp vs. log

Verify this for `x=10`, `x=2.718282`, and `x=0` using R:

$$e^{\ln(x)} = \ln(e^x) = x$$

1. Assign the three values to a vector `x`
2. Verify manually that `log` and `exp` are inverse to one another

3. Use `identical` and `all.equal` to verify the same thing

Solution:

```
## Store values in vector
x <- c(10,exp(1),0)
x

## Verify that log and exp are inverse to one another
exp(log(x)) -> exp_log
log(exp(x)) -> log_exp

## Use identical() and all.equal()
identical(exp_log,log_exp)
all.equal(exp_log,log_exp)

print(exp_log)
print(log_exp)

options(digits=16)
print(exp_log)
print(log_exp)

[1] 10.000000000  2.718281828  0.000000000
[1] FALSE
[1] TRUE
[1] 10.000000000  2.718281828  0.000000000
[1] 10.000000000  2.718281828  0.000000000
[1] 10.0000000000000002  2.718281828459045  0.0000000000000000
[1] 10.0000000000000000  2.718281828459045  0.0000000000000000
```

To compare e.g. in selection statements (`if`), always use `identical`

10 Practice: mathematical constants

Think about the mathematical packages you have to load.

1. Enter `pi` ($\pi \approx 3.14$)
2. Enter `LETTERS` and `letters`

3. What data type are `LETTERS` and `letters`?
4. Print `month.name` and `month.abb`
5. Is `month.abb` a vector? Check it!
6. Print Euler's number `e` to precision 10

Solution:

```
pi #1
LETTERS #2
letters
class(LETTERS) #3
class(letters)
month.name #4
month.abb
is.vector(month.name) #5
options(digits=10) #6
exp(1)
```

```
[1] 3.141592653589793
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
[1] "character"
[1] "character"
[1] "January" "February" "March" "April" "May" "June"
[7] "July" "August" "September" "October" "November" "December"
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
[1] TRUE
[1] 2.718281828
```

11 E-notation

Positive Powers of 10

$$10^1 = 10$$

$$10^2 = 100$$

$$10^3 = 1,000$$

$$10^4 = 10,000$$

etc.

Negative Powers of 10

$$10^{-1} = \frac{1}{10} = 0.1$$

$$10^{-2} = \frac{1}{100} = 0.01$$

$$10^{-3} = \frac{1}{1,000} = 0.001$$

$$10^{-4} = \frac{1}{10,000} = 0.0001$$

etc.

Calcworkshop.com

Scientific Notation is Based on Powers of 10

You already know that the number of digits that is displayed by R can be changed using the `options()` utility function. The default number of digits displayed is 7.

In order to display values with many more digits than that - either very large, or very small numbers, we use the scientific or e-notation. In this notation, any number is expressed as a multiple of 10.

12 E-xamples



- $10,0000 = 10 * 10 * 10 * 10 * 10 = 1 * 10^5 = 1e+05$
- $7.45678389e12 = 7.45678389 * 10^{12} = 745.678389 * 10^{10}$
- $\exp(1) = e = 271828182845e-11 = 271828182845 \times 10^{-11}$

13 Practice: e-notation

Look at the `help` for the `options` function if necessary.

1. Show the current value of how many digits are displayed
2. Reset this value to 15 (you know two ways to do this)
3. Store 100,000,000 in an object `foo` and print it
4. Print `foo` using `format` and set the attribute `scientific` to `FALSE`:
`format(foo, scientific=FALSE)`
5. Enter `0.000'000'000'000'000'10` (without the apostrophes)
6. Enter `exp(1000)`
7. Enter `(-1)/0`

8. Enter `sqrt(-1)`

```
options()$digits
options(digits=15)
100000 -> foo
foo
format(foo,scientific=FALSE)
as.integer(format(foo,scientific=FALSE)) # to compute with it
0.00000000000000010
exp(1000)
(-1)/0
sqrt(-1)

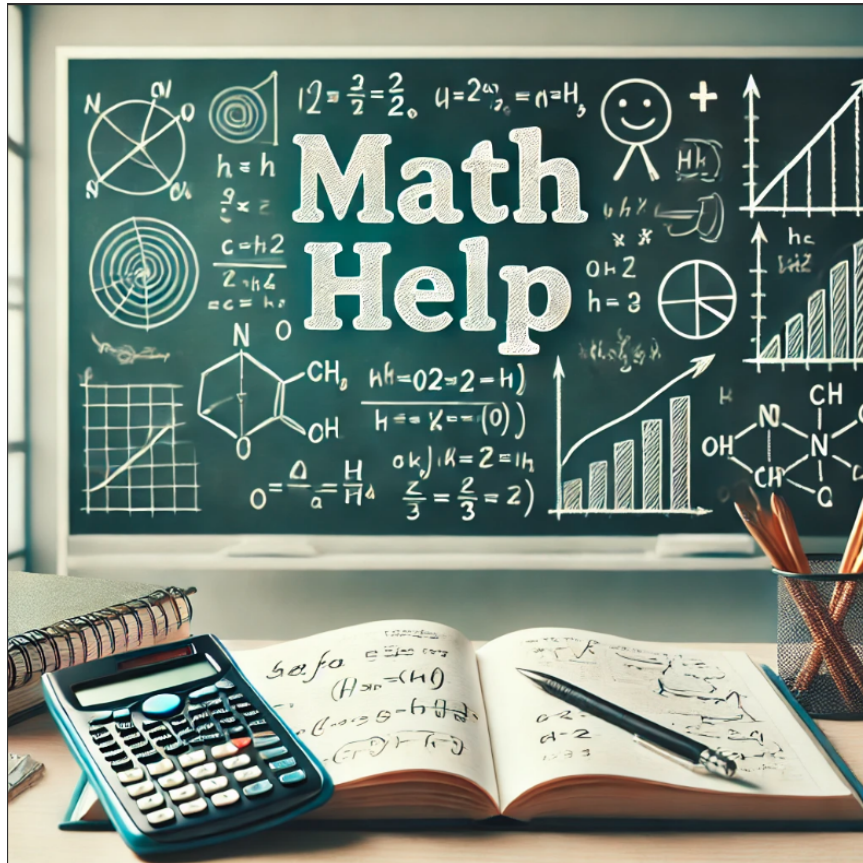
[1] 10
[1] 1e+05
[1] "100000"
[1] 100000
[1] 1e-16
[1] Inf
[1] -Inf
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
```

To get from the e-notation with exponent y or $-y$ to the complete number of digits, simply move the decimal point by y places to the right or to the left, resp.

No information is lost even if R hides digits; e-notation is purely to improve readability. Extra bits are stored by R

`Inf`, `-Inf` and `NaN` are special numbers.

14 Math help in R



- ?Arithmetic (try also `example(Arithmetic)`)
- ?Math
- ?Comparison etc.

15 Special numbers



- `Inf` for positive infinity (∞)
 - `-Inf` for negative infinity ($-\infty$)
 - `NaN` for "Not-a-Number" (not displayable)
 - `NA` for "Not Available" (missing value)
1. `NA` values are especially important when we clean data and must remove missing values. There are Boolean (logical) functions to test for special values.
 2. Non-computable values can be created easily by doing "forbidden" stuff. An example is trying to compute the square root of a negative number, e.g. `(-2)^(1/2)`. The result is a complex number (in this case the solution to the quadratic equation $x + 1 = 0$, called the imaginary number i). You can also use the function `is.na` to test for missing values: compute `is.nan(sqrt(-1))` for example.

16 Practice: special numbers

Execute the following expressions in an R code block - before running it with `C-c C-c`, think about the likely output!

1. $\text{Inf} + 1$
2. $\text{Inf} - 1$
3. $\text{Inf} - \text{Inf}$
4. Inf / Inf
5. $1/0$
6. $-1/0$
7. NA
8. NA + NA
9. NaN
10. NaN + NaN

```

Inf + 1
Inf - 1
Inf - Inf
Inf / 0
Inf / Inf
1/0
NA
NA + NA
NaN
NaN + NaN
NaN + NA
1 + NA

```

```

[1] Inf
[1] Inf
[1] NaN
[1] Inf
[1] NaN
[1] Inf
[1] NA
[1] NA
[1] NaN

```



```
[1] NaN
[1] NaN
[1] NA
```

Solution:

17 Special functions

```
is.finite(Inf)  is.infinite(Inf)
is.finite(NA)   is.na(NA)
is.nan(NaN)     is.nan(NA)
```

```
> is.finite(NA) # Missing values don't count as 'finite'
[1] FALSE
> is.infinite(Inf) # Checking infinity is dodgy but works
[1] TRUE
> is.finite(Inf)
[1] FALSE
> is.nan(NaN) # Checking "Not a Number"
[1] TRUE
> is.na(NA) # Checking missing values "Not Available"
[1] TRUE
> is.nan(NA) # Missing values are not non-numbers!
[1] FALSE
```

18 Practice: special functions

1. Enter 10^{309} and 10^{308} . What's going on?
2. Subtract `sqrt(2)**2` from 2. What's going on?
3. Show that `sqrt(2)**2` and 2 are not identical using `identical`
4. Show that `sqrt(2)**2` and 2 are almost identical using `all.equal`
5. Is NA finite in R?
6. Show that NaN is not a number
7. Are missing values numbers in R?

Solution:

```

10^309 # Inf - cannot be represented by a 64-bit computer
10^308
2-sqrt(2)**2
identical(sqrt(2)**2,2)
all.equal(sqrt(2)**2,2)
is.finite(NA)
is.nan(NA) # missing values are not non-numbers

```

1. 10^{309} is Inf. The last number is infinite, because the largest number that can be represented by a 64-bit computer is $1.7976931348623157e+308$.
2. Subtract $\sqrt{2}^2$ from 2. The answer is: $4.440892e-16$.

19 Logical values and operators

TRUE and FALSE are reserved in R for logical values, and the variables T and F are already predefined. This can cause problems, because these variable names are not reserved, i.e. you can redefine them. So better stay away from saving time by using the short versions of these values.

20 Practice: logical values

1. Print TRUE
2. Can you use `true` instead of TRUE?
3. Can you use T instead of TRUE?
4. Assign FALSE to an R object named T
5. Print T
6. What type of R object is TRUE?
7. Solution:

```

TRUE
T # printing true generates an error
T <- FALSE
T
class(TRUE)

```



Figure 2: George Boole

```
[1] TRUE
[1] TRUE
[1] FALSE
[1] "logical"
```

Richard Cotton (2011) calls R's logic "Troolean" logic, because besides the so-called Boolean values `TRUE` and `FALSE`, R also has a third logical value, the "missing" value, `NA`.

21 Logical operators

There are three logical operators in R:

```
! for "not": 1 != 1
& for "and": (1==1) & (1==2)
| for "or": (1==2) ~| (1!=1)
```

```
> 1 == 1
[1] TRUE
> 1 == 2
[1] FALSE
> 1 != 1
[1] FALSE
> 1 != 2
[1] TRUE
> 1 | 2
[1] TRUE
> 1 | 1
[1] TRUE
> (1==2) | (1!=1)
[1] FALSE
```

In the last command, we generated a **FALSE** value by comparing two **FALSE** values, which is the only way to make an **|** statement **FALSE**.

22 Practice: logical operators

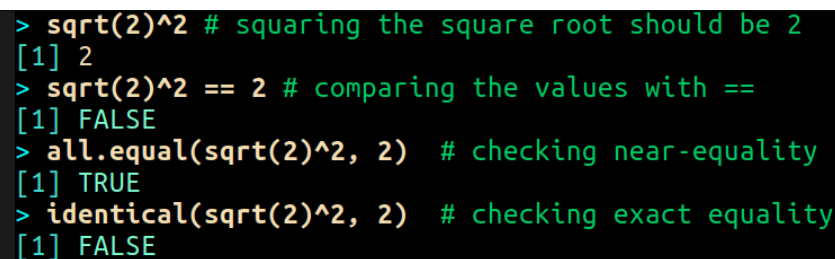
1. Check equality of $\text{sqrt}(2)^2$ and 2 using a logical operator.
2. Check that "1 is equal 2 or 1 is not equal 1" is **FALSE**.

Solution:

```
sqrt(2)^2 == 2
(1 == 2) | (1 != 1)
```

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

Comparing non-integers is iffy, because non-integers (floating-point numbers) are only an approximation of the "pure", real numbers - how accurate they are depends on the architecture of your computer. In practice, this means that rounding errors can creep in your calculations, leading to wildly wrong answers. The R FAQ has an own entry about it. The figure shows a simple example: `sqrt(2)^2` and 2 should be the same, but they aren't as far as R is concerned - a logical comparison with `==` gives `FALSE`. To test near equality (bar rounding errors), you can use the function `all.equal`. To test for exact equality, use `identical`:



```
> sqrt(2)^2 # squaring the square root should be 2
[1] 2
> sqrt(2)^2 == 2 # comparing the values with ==
[1] FALSE
> all.equal(sqrt(2)^2, 2) # checking near-equality
[1] TRUE
> identical(sqrt(2)^2, 2) # checking exact equality
[1] FALSE
```

CHALLENGE: (1) Check the help pages `?all.equal` and `?identical`. (2) Which of these numbers are infinite? 0, Inf, -Inf, NaN, NA, 10^{308} , 10^{309} . (3) How small is the rounding error in the example in the figure actually?

23 Concept summary



- In R mathematical expressions are evaluated according to the *PEMDAS* rule.
- The natural logarithm $\ln(x)$ is the inverse of the exponential function e^x .
- In the scientific or e-notation, numbers are expressed as positive or negative multiples of 10.
- Each positive or negative multiple shifts the digital point to the right or left, respectively.
- Infinity `Inf`, not-a-number `NaN`, and not available numbers `NA` are *special values* in R.

24 Code summary

CODE	DESCRIPTION
<code>log(x=,b=)</code>	logarithm of <code>x</code> , base <code>b</code>
<code>exp(x)</code>	e^x , exp[onential] of x
<code>is.finite(x)</code>	tests for finiteness of <code>x</code>
<code>is.infinite(x)</code>	tests for infiniteness of <code>x</code>
<code>is.nan(x)</code>	checks if <code>x</code> is not-a-number
<code>is.na(x)</code>	checks if <code>x</code> is not available
<code>all.equal(x,y)</code>	tests near equality
<code>identical(x,y)</code>	tests exact equality
<code>1e2, 1e-2</code>	$10^2 = 100$, $10^{-2} = \frac{1}{100}$

25 References

- Richard Cotton (2013). Learning R. O'Reilly Media.
- Tilman M. Davies (2016). The Book of R. (No Starch Press).
- Rafael A. Irizarry (2020). Introduction to Data Science (also: CRC Press, 2019).
- Norman Matloff (2020). faster: Fast Lane to Learning R!.