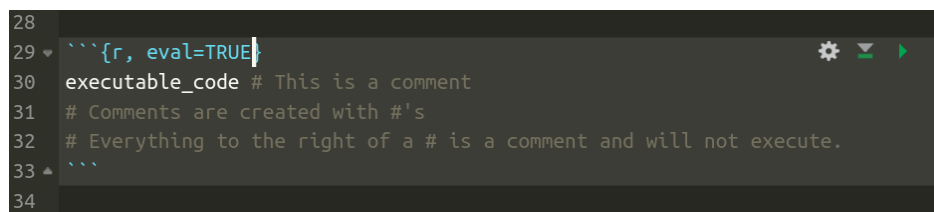


# DUNE Team R/RStudio Tutorial


Sean Gilligan


## Code Chunks and Knitting in RStudio

An R Markdown (`.Rmd`) file in **RStudio** takes advantage of what are called “code chunks” to designate areas in which to write and evaluate R code. A code chunk takes up a number of lines of code as determined by the user, and is started and terminated with dedicated lines of backticks and `{` bracket `}` enclosed arguments related to the code chunk’s evaluation during the R Markdown file’s compilation, in a process called “knitting”. Within **RStudio**, depending on your appearance theme (**Tools** → **Global Options...** → **Appearance**), code chunks will look like

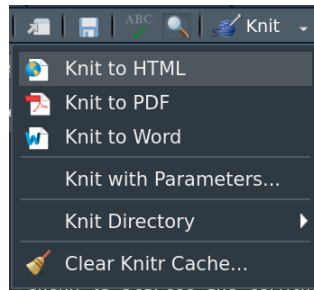


```
28
29 ```${r, eval=TRUE}
30 executable_code # This is a comment
31 # Comments are created with #'s
32 # Everything to the right of a # is a comment and will not execute.
33 ```
34
```

The symbols in the top right of the code chunk are created automatically. The far right green arrow  executes the entire chunk of code, with any outputs being displayed just below the code chunk, in the order they were produced. When the R Markdown file is knitted, the lines with backticks will not appear in the produced document, and outputs will appear just below the line of code that produced them, giving the impression of multiple code chunks. Within **RStudio** individual lines within a code chunk can be evaluated by placing the cursor on the line and hitting **Ctrl+Enter**. Sections of the code selected by the cursor can be executed in the same way. The selected code can be multiple lines of code or a small section within a single line of code. The only requirement being that all the code is within the same chunk.

The middle down facing gray arrow  executes, in order, all code chunks above the current chunk, not including the current chunk. The gear button allows for the selection of chunk options relating to how the chunk evaluates during the “knitting” process, which can also be done by manual entry of arguments to the right of the `r` in `{r}`. In the screenshot of the code chunk above the argument `eval` is set to its default value, which it will take regardless of whether or not we include it. Setting it to `FALSE` would prevent the code chunk from evaluating during the knitting process. The arguments `message` and `warning` will be set to `FALSE` in certain code chunks below to prevent spurious messages and warnings from clogging up the final knitted document.

The knitting process is the compilation process that evaluates all the formatting and R code in the R Markdown file to produce a PDF (like this one), HTML, or WORD document that is hopefully nice enough to show your peers, teachers, students, employers, employees, editor, advisor, et cetera. While clicking **File** → **Knit Document** or using the indicated keyboard shortcut is one way to carry this out, the button featuring the little ball of yarn with the knitting needle poking out in the toolbar gets the most action. Clicking the down arrow next to it shows some additional, and usually unnecessary, options as shown below.



The file type to knit to should have been decided when you created the R Markdown file, and that information should appear as an output instruction in the YAML header at the top of the R Markdown document. For a PDF it will be “output: pdf\_document”.

## Variables and Functions in R

Variables are containers for objects and are mainly created at the same time an object is assigned to them. Data types and structures are communicated automatically based on the properties of the assigned object, so no special declarations are necessary as might be needed in other languages, such as in C++.

Assignment of an object to a variable can be accomplished using either = or <-, with the arrow pointing from the object to the variable. The arrow indicates the direction of the flow of information. As such, -> is also valid, despite being unconventional during normal variable declarations. Additionally, multiple variables with the same desired value can be declared and assigned values in the same line.

Assigning a new object to an object-holding variable erases all content regarding the previous object and any relevant information about it. With that said objects designed to contain multiple objects, such as vectors and matrices, result in variables that can take arguments to point to or generate new internal locations for the assignment of new objects. Only objects stored in an already existing internal location will then be erased in above described way. Printing the contents of a variable can be accomplished by merely typing the variable's name and evaluating it as R code.

```
number <- 5  
number
```

```
## [1] 5
```

```
name = "ProtoDUNE"  
name
```

```
## [1] "ProtoDUNE"
```

```
name <- "DUNE"  
name
```

```
## [1] "DUNE"
```

```
n2 <- n1 <- 4 -> n3  
n1
```

```
## [1] 4
```

```
n2
```

```
## [1] 4
```

```
n3
```

```
## [1] 4
```

Functions are prominent utilities in R. Many come in base R and do not require any additional packages to be loaded. Until stated otherwise the functions discussed ahead will be of this sort. Calling a function works similarly to printing the contents of a variable above, except that a function requires the inclusion of a container for potential arguments, or inputs. This container takes the form of closed parentheses following the name of the function, so that with no arguments will have the form `function()`. When more than one argument is used they are separated by commas, in which case it's often best practice to assign arguments explicitly to the function variables as defined in the function's documentation, giving it the form `function(var1 = arg1, var2 = arg2, ... , varn = argn)`.

Note, in functions it is necessary to use `=` when assigning objects to a function's variables, the default values of which can also be found in the documentation. This documentation can usually easily be found online. However, executing the R code `?function`, where “function” is the function name without parentheses, will provide the documentation as well. As you get comfortable with functions you will know how and when you can and cannot get away without explicitly assigning objects to function variables and which variables can be allowed to take their default values, resulting in forms such as `function(arg1,arg2,var5=arg5)`.

## Vectors and Matrices in R

Vectors can be explicitly defined element by element using the function `c()`, which assigns the enclosed comma-separated objects to each element of the vector in order. The length of the vector is determined by the number of objects, and the objects in a vector must each be of the same data type. Data types will be discussed later in its own section. For now we only really need to consider **numeric** and **character**, representing basic math and text objects. Below the integer sequence `{1,2,3,4}` is explicitly defined.

```
c(1,2,3,4)
```

```
## [1] 1 2 3 4
```

The objects can also be other vectors. Different objects can have the same data type.

```
c(c(1,2,3),4)
```

```
## [1] 1 2 3 4
```

Sequences like these can take advantage of built in shortcuts and functions. Here are two possible ways to generate the same sequence.

```
1:4
```

```
## [1] 1 2 3 4
```

```
seq(from = 1, to = 4)
```

```
## [1] 1 2 3 4
```

The function `seq()` allows additional arguments to specify step size (`\texttt{by=}`) or output vector length (`\texttt{length.out=}`).

```
seq(1,4, by = 0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

```
seq(1,4, length.out = 7)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

Decreasing sequences can also be created with these methods.

```
4:1
```

```
## [1] 4 3 2 1
```

```
seq(4,1)
```

```
## [1] 4 3 2 1
```

```
seq(4,1, by = -0.5) # Requires negative steps
```

```
## [1] 4.0 3.5 3.0 2.5 2.0 1.5 1.0
```

```
seq(4,1, length.out = 7)
```

```
## [1] 4.0 3.5 3.0 2.5 2.0 1.5 1.0
```

Vectors with repeating objects or patterns can be generated using `rep()`. When vectors are the objects being repeated options are available to repeat the entire vector (default) or repeat each element, as shown in the examples below.

```
rep(x = 5, times = 5)
```

```
## [1] 5 5 5 5 5
```

```
rep(c(1,5), 3) # Assumes 3 assigned to times, unless stated otherwise
```

```
## [1] 1 5 1 5 1 5
```

```
rep(c(1,5), each = 3)
```

```
## [1] 1 1 1 5 5 5
```

### Some notable operations, functions, and tricks

Here are some quick functions that one might find useful when handling vectors. Computational functions are of course only appropriate when dealing with vectors of numbers.

```
v1 <- 1:6  
v1
```

```
## [1] 1 2 3 4 5 6
```

```
length(v1) # number of elements
```

```
## [1] 6
```

```
rev(v1) # reverse
```

```
## [1] 6 5 4 3 2 1
```

```
sum(v1) # sum
```

```
## [1] 21
```

```
prod(v1) # product
```

```
## [1] 720
```

```
mean(v1) # mean
```

```
## [1] 3.5
```

```
cumsum(v1) # cumulative sum
```

```
## [1] 1 3 6 10 15 21
```

```
cumprod(v1) # cumulative product
```

```
## [1] 1 2 6 24 120 720
```

Random sampling from a vector can be performed using `sample()`. By default will randomly sample without replacement until all elements of the vector have been sampled.

```
sample(x = v1) # Default size = length(v1)
```

```
## [1] 3 2 6 5 1 4
```

```
sample(v1, size = 4, replace = T)
```

```
## [1] 5 1 4 3
```

```
sample(v1, 3)
```

```
## [1] 6 3 1
```

By default the function assumes equal probability between elements. For the sequence  $\{1, 2, 3, 4, 5, 6\}$  this would then give a probability to each of  $1/6$ . If we instead assigned a vector of probabilities such that the probability of sampling 1 was  $1/2$  and the probability of getting any one of the other numbers was  $1/10$ , theory tells us that the average value sampled with replacement would be

$$\sum_{x=1}^6 (\text{The probability of drawing } x) \times (x) = (0.5)(1) + (0.1)(2) + (0.1)(3) + (0.1)(4) + (0.1)(5) + (0.1)(6) = 2.5$$

```
0.5*1 + 0.1*2 + 0.1*3 + 0.1*4 + 0.1*5 + 0.1*6
```

```
## [1] 2.5
```

This average value can be estimated by taking the average  $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$  of some  $n$  samples  $x_i, i = 1, 2, \dots, n$ . I do this below by sampling  $n = 10000$  times with replacement and then taking the mean.

```
xs <- sample(x = v1, size = 10000, prob = c(0.5, rep(0.1, 5)), replace = T)
# The mean of the sample is
mean(xs)
```

```
## [1] 2.4872
```

This sample is rather large. It may it be in our interests to clear this variable to avoid running out of memory. This can be done with the function `rm()`.

```
rm(xs)
```

The function `exists()` can then verify for us that the variable, the name of which is passed as an argument in quotes, and its data were indeed removed.

```
exists("xs")
```

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

The function `ls()` can be used to show all the declared variables in the current environment.

```
ls()
```

```
## [1] "n1"      "n2"      "n3"      "name"    "number"  "v1"
```

Comparing values is done similarly to other languages. The logical NOT is implemented via a “!” and, with the exception of “!=”, must be placed and formatted in a way to “see” the entire comparison. The orderings for “>=” and “<=” are not a choice up to the user.

```
c(5 == 6,  
   5 != 6,  
   5 > 6,  
   5 < 6,  
   5 >= 6,  
   5 <= 6,  
   !(5 > 6),  
   !(5 < 6))
```

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

Comparing whole vectors is done element by element when applicable.

```
1:6 == c(1,5,3,4,2,6)
```

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

A single value compared will check for each element in the vector. For vectors of unequal size, the comparison will be done element by element and the smaller vector will repeat as necessary. A warning will be given if the shorter vector’s length does not evenly divide the larger vector’s length.

```
1:6 <= 3
```

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

```
1:6 > c(1,3,3)
```

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

```
1:6 == c(5,6,1,2)
```

```
## Warning in 1:6 == c(5, 6, 1, 2): longer object length is not a multiple of  
## shorter object length
```

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

Other functions exist for cumulative comparisons, producing quantitative information about potential mismatches.

```
all.equal(1:6, seq(1,6))
```

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

```
all.equal(1:6,c(1:5,7))
```

```
## [1] "Mean relative difference: 0.1666667"
```

```
all.equal(c("l1","l2","l3"),c("l1","l2","r3"))
```

```
## [1] "1 string mismatch"
```

Another way to do this involves recognizing that TRUE and FALSE revert to 1 and 0 when subjected to mathematical operations.

```
prod(c(TRUE,FALSE))
```

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

```
sum(c(TRUE,FALSE))
```

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

```
prod(c(TRUE,TRUE))
```

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

```
sum(c(TRUE,TRUE))
```

```
## [1] 2
```

```
# If all TRUE then  
prod(1:6 == seq(1,6))
```

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

```
# If any FALSE then  
prod(1:6 == c(1,2,3,4,5,5))
```

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

## Matrices

One way to create a matrix is by binding vectors together as columns or rows with `cbind()` and `rbind` respectively.



```
cbind(1:3,4:6,7:9) # column bind
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
rbind(1:3,4:6,7:9) # row bind
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
# will repeat elements if lengths unequal
```

```
cbind(1:2,3:6,7)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    7
## [2,]    2    4    7
## [3,]    1    5    7
## [4,]    2    6    7
```

```
# Will give warning if lengths not divisible.
```

```
# Lets longest vector define the relevant dimension
```

```
cbind(1:3,4:5,6:8)
```

```
## Warning in cbind(1:3, 4:5, 6:8): number of rows of result is not a multiple of
## vector length (arg 2)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    6
## [2,]    2    5    7
## [3,]    3    4    8
```

An explicit `matrix()` function also exists. It requires a supplied vector and at least one dimensional input.

```
matrix(1:8, nrow=2) # Fills columns first by default
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

```
matrix(1:8, nrow=2, byrow=T)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
```

```
matrix(1:8, ncol=4, byrow=T)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
```

```
matrix(1:8) # No dimensional inputs assumes a single column...
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
## [5,]    5
## [6,]    6
## [7,]    7
## [8,]    8
```

```
matrix(1:8, byrow=T) # ... Even if filling by row
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
## [5,]    5
## [6,]    6
## [7,]    7
## [8,]    8
```

If vector too short will just repeat with expected warnings

```
matrix(1:2, nrow=4, ncol=5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    1    1    1
## [2,]    2    2    2    2    2
## [3,]    1    1    1    1    1
## [4,]    2    2    2    2    2
```

```
matrix(0,4,4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0
```

```
matrix(1:3,nrow=4,ncol=5)
```

```
## Warning in matrix(1:3, nrow = 4, ncol = 5): data length [3] is not a sub-  
## multiple or multiple of the number of rows [4]
```

```
##      [,1] [,2] [,3] [,4] [,5]  
## [1,]    1    2    3    1    2  
## [2,]    2    3    1    2    3  
## [3,]    3    1    2    3    1  
## [4,]    1    2    3    1    2
```

A matrix will be converted back to a vector if passed as an argument through certain vector friendly functions.

```
mat1 <- matrix(1:4,2)  
mat1
```

```
##      [,1] [,2]  
## [1,]    1    3  
## [2,]    2    4
```

```
c(mat1)
```

```
## [1] 1 2 3 4
```

```
rev(mat1)
```

```
## [1] 4 3 2 1
```

```
rep(mat1)
```

```
## [1] 1 2 3 4
```

Diagonal matrices can be generated with the function `diag()`.

```
diag(4) # When just a number n creates n*n identity matrix
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    1    0    0    0  
## [2,]    0    1    0    0  
## [3,]    0    0    1    0  
## [4,]    0    0    0    1
```

```
diag(2,4) # If two numbers, the first will be the elements
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    2    0    0    0  
## [2,]    0    2    0    0  
## [3,]    0    0    2    0  
## [4,]    0    0    0    2
```

```
diag(2,4) == 2*diag(4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,] TRUE TRUE TRUE TRUE
## [2,] TRUE TRUE TRUE TRUE
## [3,] TRUE TRUE TRUE TRUE
## [4,] TRUE TRUE TRUE TRUE
```

```
diag(2, nrow = 5, ncol = 4) # Does not have to be square
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    0    0    0
## [2,]    0    2    0    0
## [3,]    0    0    2    0
## [4,]    0    0    0    2
## [5,]    0    0    0    0
```

```
diag(1:4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    2    0    0
## [3,]    0    0    3    0
## [4,]    0    0    0    4
```

```
diag(c(-3,1,8))
```

```
##      [,1] [,2] [,3]
## [1,]   -3    0    0
## [2,]    0    1    0
## [3,]    0    0    8
```

Getting values from matrices by index

```
mat1 <- matrix(1:9,nrow=3)
mat1
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
mat1[1,] # indexing starts at 1
```

```
## [1] 1 4 7
```

```
mat1[,2]
```

```
## [1] 4 5 6
```

```
mat1[2,2]
```

```
## [1] 5
```

```
mat1[1:2,c(3,1)]
```

```
##      [,1] [,2]  
## [1,]    7    1  
## [2,]    8    2
```

```
#####  
#### Quick aside: paste and paste0 ####  
#####
```

```
# default sep=" "  
paste("default","separation")
```

```
## [1] "default separation"
```

```
paste("elements","separated", "by", "commas", sep = ",")
```

```
## [1] "elements,separated,by,commas"
```

```
paste("no","separation", sep = "")
```

```
## [1] "noseparation"
```

```
# paste0() is paste() with sep=""  
paste0("default ", "no ", "separation")
```

```
## [1] "default no separation"
```

```
# vectors and matrices paste element by element  
paste0(c("a1","a2","a3"),c("b1","b2","b3"))
```

```
## [1] "a1b1" "a2b2" "a3b3"
```

```
# elements repeat as necessary  
paste0("a",1:3)
```

```
## [1] "a1" "a2" "a3"
```

```
# always outputs as a vector  
paste0(matrix(1:4,nrow=2),matrix(c("a","b","c","d"),nrow=2))
```

```
## [1] "1a" "2b" "3c" "4d"
```

```
# collapse converts to length one character
paste0("a",1:3 ,collapse = ", ")
```

```
## [1] "a1, a2, a3"
```

Naming rows and columns

```
# rownames() and colnames()
rownames(mat1) <- paste0("r",1:3)
colnames(mat1) <- paste0("c",1:3)
mat1
```

```
##      c1 c2 c3
## r1   1  4  7
## r2   2  5  8
## r3   3  6  9
```

```
# Calling elements by row and column names
mat1["r1",]
```

```
## c1 c2 c3
##  1  4  7
```

```
mat1[, "c2"]
```

```
## r1 r2 r3
##  4  5  6
```

```
mat1["r2", "c1"]
```

```
## [1] 2
```

The methods for multiplying matrices and vectors aren't immediately clear. Here

```
m1 <- matrix(1,4,4)
m1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    1    1    1    1
## [3,]    1    1    1    1
## [4,]    1    1    1    1
```

```
v1 <- 1:4
v1
```

```
## [1] 1 2 3 4
```

```
m1 * v1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    2    2    2    2
## [3,]    3    3    3    3
## [4,]    4    4    4    4
```

Need to use `%*%` to do proper matrix multiplication

```
m1 %*% v1
```

```
##      [,1]
## [1,]   10
## [2,]   10
## [3,]   10
## [4,]   10
```

Inner products between vectors are natural extensions

```
sum(v1^2) # What we expect
```

```
## [1] 30
```

```
v1 %*% v1
```

```
##      [,1]
## [1,]   30
```

R makes inferences on the intended orientation of a vector based on where it is placed.

```
t(v1) %*% v1
```

```
##      [,1]
## [1,]   30
```

```
t(v1) %*% t(t(v1))
```

```
##      [,1]
## [1,]   30
```

```
v1 %*% t(v1)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16
```

```
t(t(v1)) %*% t(v1)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16
```

```
m1 %*% v1
```

```
##      [,1]
## [1,]   10
## [2,]   10
## [3,]   10
## [4,]   10
```

```
m1 %*% t(t(v1))
```

```
##      [,1]
## [1,]   10
## [2,]   10
## [3,]   10
## [4,]   10
```

```
v1 %*% m1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10   10   10   10
```

```
t(v1) %*% m1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10   10   10   10
```

```
v1 %*% m1 %*% v1
```

```
##      [,1]
## [1,]  100
```

```
t(v1) %*% m1 %*% t(t(v1))
```

```
##      [,1]
## [1,]  100
```

From above it should be clear that unless declared otherwise left-hand-side vectors (`v1 %*% ____`) are assumed to be oriented as



```
t(v1)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
```

And right-hand-sided vectors (\_\_\_ %\*% v1) are assumed to be oriented as

```
t(t(v1))
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
```

Latex is possible within R Markdown.

```
$$\begin{pmatrix}1&2&3&4\end{pmatrix}
\begin{pmatrix}2&0&0&0\\0&2&0&0\\0&0&2&0\\0&0&0&2\end{pmatrix}
\begin{pmatrix}1\\2\\3\\4\end{pmatrix}=\begin{pmatrix}1&2&3&4\end{pmatrix}
\begin{pmatrix}2\\4\\6\\8\end{pmatrix}=60$$
```

$$(1 \ 2 \ 3 \ 4) \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = (1 \ 2 \ 3 \ 4) \begin{pmatrix} 2 \\ 4 \\ 6 \\ 8 \end{pmatrix} = 60$$

The above operation is executed below.

```
1:4 %*% (2*diag(4)) %*% 1:4
```

```
##      [,1]
## [1,]    60
```

Transverse matrix

```
matrix(1:9,nrow=3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
t(matrix(1:9,nrow=3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
1:3
```

```
## [1] 1 2 3
```

```
t(1:3)
```

```
##      [,1] [,2] [,3]  
## [1,]    1    2    3
```

```
t(t(1:3))
```

```
##      [,1]  
## [1,]    1  
## [2,]    2  
## [3,]    3
```

Matrix inverses

```
matrix(c(1,2,-3,4), nrow=2)
```

```
##      [,1] [,2]  
## [1,]    1  -3  
## [2,]    2    4
```

```
solve(matrix(c(1,2,-3,4),nrow=2))
```

```
##      [,1] [,2]  
## [1,]  0.4  0.3  
## [2,] -0.2  0.1
```

```
matrix(c(1,2,-3,4),nrow=2) %*% solve(matrix(c(1,2,-3,4),nrow=2))
```

```
##      [,1]      [,2]  
## [1,]    1 -5.551115e-17  
## [2,]    0  1.000000e+00
```

Matrix determinant

```
det(matrix(c(1,2,-3,4), nrow=2))
```

```
## [1] 10
```

```
det(solve(matrix(c(1,2,-3,4), nrow=2)))
```

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

## Eigenvalues and eigenvectors

Built in functions exist for finding eigenvalues and eigenvectors of a matrix. Take the Pauli matrix

$$\sigma_1 = \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

```
mat1 <- matrix(c(0,1,1,0), nrow=2)
mat1
```

```
##      [,1] [,2]
## [1,]    0    1
## [2,]    1    0
```

The function `eigen()` produces a list of the eigenvalue and eigenvectors.

```
eigen1 <- eigen(mat1)
eigen1$values
```

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

The eigenvectors are presented as a matrix, where  $n$ th column is the eigenvector corresponding the  $n$ th eigenvalue.

```
eigen1$vectors
```

```
##      [,1]      [,2]
## [1,] 0.7071068 -0.7071068
## [2,] 0.7071068  0.7071068
```

Verification can be done easily enough.

```
mat1 %*% eigen1$vectors[,1] == eigen1$values[1]*eigen1$vectors[,1]
```

```
##      [,1]
## [1,] TRUE
## [2,] TRUE
```

```
mat1 %*% eigen1$vectors[,2] == eigen1$values[2]*eigen1$vectors[,2]
```

```
##      [,1]
## [1,] TRUE
## [2,] TRUE
```

## Data Types in R

### Logical

```
class(c(TRUE,FALSE,T,F,NA))
```

```
## [1] "logical"
```

## Numeric

```
class(c(2,4.8,-120.10,pi))
```

```
## [1] "numeric"
```

## Integer

```
class(c(3L,34L,0L))
```

```
## [1] "integer"
```

## Complex

```
class(3+2i)
```

```
## [1] "complex"
```

## Character

```
class(c("Hello","TRUE","4.8","0L","3+2i","This is a sentence."))
```

```
## [1] "character"
```

## Raw

```
charToRaw("Hello")
```

```
## [1] 48 65 6c 6c 6f
```

```
class(charToRaw("Hello"))
```

```
## [1] "raw"
```

## Mixing data types

Interesting hierarchies and conversions present themselves when mixing data types as shown below.

```
class(c(1.4,charToRaw("H"),6L,T,3+2i,"char"))
```

```
## [1] "character"
```

```
c(1.4,charToRaw("H"),6L,T,3+2i,"char")
```

```
## [1] "1.4" "48" "6" "TRUE" "3+2i" "char"
```

```
class(c(1.4,charToRaw("H"),6L,T,3+2i))
```

```
## [1] "complex"
```

```
c(1.4,charToRaw("H"),6L,T,3+2i)
```

```
## [1] 1.4+0i 72.0+0i 6.0+0i 1.0+0i 3.0+2i
```

```
class(c(1.4,charToRaw("H"),6L,T))
```

```
## [1] "numeric"
```

```
c(1.4,charToRaw("H"),6L,T)
```

```
## [1] 1.4 72.0 6.0 1.0
```

```
class(c(1.4,charToRaw("H"),6L))
```

```
## [1] "numeric"
```

```
class(c(1.4,charToRaw("H")))
```

```
## [1] "numeric"
```

```
class(c(charToRaw("H"),T))
```

```
## [1] "logical"
```

```
c(charToRaw("H"),T)
```

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

## R-Objects

We have already seen vectors and matrices. Other common objects are discussed below.

## Arrays

An array is a generalized vector and matrix type object that has an arbitrary number of dimensions. The example shown below is a  $2 \times 2 \times 2 \times 2$  array filled from the contents of the enclosed sequence. Discussion of rows and columns become difficult, so discussion of how the sequence fills the array requires a more generalized description. A particular location within this array uses a 4-dimensional index  $\{i, j, k, l\}$  and is accessed by `array1[i, j, k, l]`. In filling the array the elements start at the index  $\{1, 1, 1, 1\}$ , filling each dimension fully from left to right, resulting in the fill order  $\{2, 1, 1, 1\}$ ,  $\{1, 2, 1, 1\}$ ,  $\{2, 2, 1, 1\}$ ,  $\{1, 1, 2, 1\}$ ,  $\{2, 1, 2, 1\}$ ,  $\{1, 2, 2, 1\}$ ,  $\{2, 2, 2, 1\}$ ,  $\{1, 1, 1, 2\}$ ,  $\{2, 1, 1, 2\}$ ,  $\{1, 2, 1, 2\}$ ,  $\{2, 2, 1, 2\}$ ,  $\{1, 1, 2, 2\}$ ,  $\{2, 1, 2, 2\}$ ,  $\{1, 2, 2, 2\}$ ,  $\{2, 2, 2, 2\}$ .

```
array1 <- array(1:16, dim=c(2,2,2,2))
array1
```

```
## , , 1, 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2, 1
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
## , , 1, 2
##
##      [,1] [,2]
## [1,]    9   11
## [2,]   10   12
##
## , , 2, 2
##
##      [,1] [,2]
## [1,]   13   15
## [2,]   14   16
```

## Factors

Created from vectors, factors assign levels to items in vectors, even when not normally appropriate. By default they are determined alphabetically for characters. Factor levels are important during visualizations, by systematically determining the order in which data should be plotted, have aesthetics applied, and placed in a legend. Creating a factor from a

```
factor1 <- factor(c("red", "red", "blue", "red", "green", "blue"))
factor1
```

```
## [1] red   red   blue  red   green blue
## Levels: blue green red
```

```
# Can change order when declaring or later
factor(c("red","red","blue","red","green","blue"),
       levels = c("red","green","blue"))
```

```
## [1] red  red  blue  red  green blue
## Levels: red green blue
```

```
levels(factor1) <- c("red","green","blue")
factor1
```

```
## [1] blue  blue  red   blue  green red
## Levels: red green blue
```

## Data Frames

Data frames at first might seem similar to matrices with named columns. However, while data frames support a lot of matrix-like computations they also support relational and spreadsheet-like operations, and can support heterogeneous data mixtures. This hybrid nature stems from the fact that data frames were developed at Bell Labs for the S programming language from the need to treat data as both a matrix and a table. Released to the world in 1990, data frames were inherited by S's open-source counterpart R for its release in 2000, and by 2009 the **pandas** software library managed to become open-source, bringing data frames officially to Python. The philosophy governing the Python language's nascent and continuing development has resulted in an accessibility and expansive versatility that lends Python to becoming a dominant tool in performing whatever tasks for which supportive Python libraries happen to exist. Being no exception, the growing ubiquity of Python in data science over the last decade is predicated by its collective capabilities crossing certain critical thresholds with the release of libraries like NumPy, SciPy, TensorFlow, and **pandas**, to name just a few. The pervasive role and effectiveness of the **pandas DataFrame** object in facilitating many of Python's data analysis techniques has allowed data frames to achieve a well-deserved popularity that reminds me of [Whitney Houston's](#) cover of [Dolly Parton's](#) "I Will Always Love You". Talented young data scientists working in Python and getting baptized in the black boxes of machine learning and neural networks would not be remiss in assuming that data frames were invented by Python developers.

Since we are doing particle physics I should also mention that ROOT has an [RDataFrame](#) Class that has seen improvements as recently as ROOT's latest [release](#). I do not have any experience with it yet, but there appears to be some [enthusiasm](#) about it. With that said here is an example of a manually constructed data frame comparing the largest mountain on each continent.

```
mtns <- data.frame(Name = c("Everest","Kilimanjaro","Vinson Massif",
                           "Kosciusko","Elbrus","Denali","Aconcagua"),
                  `Height (m)` = c(8850,5895,4897,2228,5642,6194,6960),
                  Continent = c("Asia","Africa","Antarctica","Australia",
                                "Europe","North America","South America"),
                  check.names = F # Allows column names with atypical characters
                  )

mtns
```

```
##           Name Height (m)  Continent
## 1    Everest    8850      Asia
## 2 Kilimanjaro    5895      Africa
## 3 Vinson Massif    4897  Antarctica
## 4   Kosciusko    2228   Australia
## 5      Elbrus    5642      Europe
```

```
## 6      Denali      6194 North America
## 7      Aconcagua    6960 South America

# Can call columns by name to produce vectors
mtns$`Height (m)`
```

```
## [1] 8850 5895 4897 2228 5642 6194 6960
```

```
# Can index like a matrix
mtns[,2]
```

```
## [1] 8850 5895 4897 2228 5642 6194 6960
```

```
mtns[1,]
```

```
##      Name Height (m) Continent
## 1 Everest      8850      Asia
```

Notice that when a column is retrieved it is isolated as a vector, but rows remain data frames. As the above data frame contains characters it would be inappropriate to perform mathematical operations on it. Below I have have recreated the earlier Pauli matrix as a data frame and performed some operations on it. I still have to tell it to be interpreted as a matrix when performing matrix multiplication, but it takes readily to eigen (spectral) decomposition as is.

```
df1 <- data.frame(matrix(c(0,1,1,0),2))
df1
```

```
##   X1 X2
## 1  0  1
## 2  1  0
```

```
solve(df1) %*% as.matrix(df1)
```

```
##   X1 X2
## X1  1  0
## X2  0  1
```

```
t(df1)
```

```
##   [,1] [,2]
## X1   0   1
## X2   1   0
```

```
eigen(df1)
```

```
## eigen() decomposition
## $values
## [1]  1 -1
##
## $vectors
##           [,1]      [,2]
## [1,] 0.7071068 -0.7071068
## [2,] 0.7071068  0.7071068
```



## Lists

Lists are versatile objects that can hold many different objects and elements, including other lists. Here is a list holding an array, a factor, and a data frame.

```
list1 <- list(array1, factor1, mtns)
list1[[2]]
```

```
## [1] blue blue red blue green red
## Levels: red green blue
```

Items in a list can be named and accessed in ways similar to the named columns of a data frame.

```
list1 <- list(array1 = array1, factor1 = factor1, mtns = mtns)
list1$factor1
```

```
## [1] blue blue red blue green red
## Levels: red green blue
```

As lists can contain other lists, named objects can be nested within layers of many other named objects. Here the Continent column of `mtns` is three named objects deep.

```
# Names can be nested
list2 <- list(array1 = array1, factor1 = factor1,
              mtns = mtns, list1 = list1)
list2$list1$mtns$Continent
```

```
## [1] "Asia"          "Africa"          "Antarctica"      "Australia"
## [5] "Europe"        "North America"  "South America"
```

## Reading in and looking at data

Checking and setting the work directory can be done explicitly as shown below. By default the working directory is whatever directory the file is saved in.

```
getwd() # Outputs the current working directory.
```

```
## [1] "/home/gilligas/DUNE/Network/data-mgmt-testing/R_Code"
```

```
#setwd("/home/gilligas/DUNE/Network/") # Change this as necessary
```

Files in the current directory can be read in explicitly with `dir()`. Paths to other directories can be passed as an argument to inspect their contents as well.

```
dir()
```

```
## [1] "RatesByAppAndSite.html" "RatesByAppAndSite.Rmd" "README.md"
## [4] "Tutorial_Files"        "Tutorial.log"         "Tutorial.pdf"
## [7] "Tutorial.Rmd"
```

```
dir("Tutorial_Files")
```

```
## [1] "data"          "images"          "README.md"
## [4] "user_generated_files"
```

Checking for the existence of a directory can be done with `dir.exists("path/to/folder")`.

```
dir.exists("Tutorial_Files/user_generated_files")
```

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

```
#####
##### QUICK ASIDE : gsub() #####
#####
```

```
# gsub() can be used to replace certain patterns in a character vector.
# The function below replaces the spaces in "Remove my spaces" with underscores.
gsub(pattern = " ", replacement = "_", x = "Remove my spaces.")
```

```
## [1] "Remove_my_spaces."
```

```
# Continuing with dir.exists()
foldername <- gsub(" ", "_", paste("folder", Sys.time()))
dir.exists(paste0("Tutorial_Files/user_generated_files/", foldername))
```

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

```
# file.exists() works similarly for files.
file.exists("Tutorial.Rmd")
```

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

```
file_unlikely_to_exist = paste0("file_", paste(sample(0:9, replace=T, 20), collapse = ""))
c(file_unlikely_to_exist, file.exists(file_unlikely_to_exist))
```

```
## [1] "file_22113192969153492249" "FALSE"
```

Creating a directory can then be performed with `dir.create("path/to/folder")`. You will be informed if a directory already exists and nothing will happen.

```
dir.create(paste0("Tutorial_Files/user_generated_files/", foldername))
dir.exists(paste0("Tutorial_Files/user_generated_files/", foldername))
```

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

```
dir("Tutorial_Files/user_generated_files/")
```

```
## [1] "folder_2021-07-19_19:35:30" "folder_2021-07-19_19:53:40"
## [3] "folder_2021-07-19_22:57:47" "folder_2021-07-19_22:59:51"
## [5] "folder_2021-07-19_23:02:17" "folder_2021-07-19_23:06:48"
## [7] "folder_2021-07-19_23:32:35"
```

```
dir.create("Tutorial_Files")
```

```
## Warning in dir.create("Tutorial_Files"): 'Tutorial_Files' already exists
```

There are many functions for reading in files. `read.csv()` alone is sufficient for the format we have here. The data should read in a data frame.

```
path_to_data <- "Tutorial_Files/data/"
filename <- "SUBSET_user_2021-06-01_2021-06-28.csv"
data <- read.csv(paste0(path_to_data,filename))
```

Actually looking at this data in the knitted output PDF takes some finessing. The number of columns and rows can tell us how big the object is we're dealing with.

```
# nrow() and ncol() tell us the number of rows and columns respectively
c(nrow(data), ncol(data))
```

```
## [1] 43865    19
```

Some useful functions for looking only at a little bit of a large data frame is `head()` or `tail()`, which show the first six and last six rows of data. An additional argument can be passed to either to increase or decrease the number as desired. However, given the number of columns, and the fact that we can't just ignore looking a bunch of columns means we need to split up data frame vertically and post the contents one at a time. Here are the first 5 columns of the first 10 rows.

```
head(data, 10)[,1:5]
```

```
##           disk  user      date process_id      timestamp
## 1  fndca1.fnal.gov user7 2021-06-01  15799375 2021-06-01T01:38:24.685Z
## 2  fndca1.fnal.gov user7 2021-06-01  15799345 2021-06-01T01:38:28.001Z
## 3  fndca1.fnal.gov user7 2021-06-01  15799378 2021-06-01T01:38:28.039Z
## 4  fndca1.fnal.gov user7 2021-06-01  15799297 2021-06-01T01:38:28.793Z
## 5  fndca1.fnal.gov user7 2021-06-01  15799360 2021-06-01T01:38:32.539Z
## 6  fndca1.fnal.gov user7 2021-06-01  15799361 2021-06-01T01:38:37.935Z
## 7  fndca1.fnal.gov user7 2021-06-01  15799263 2021-06-01T01:38:41.200Z
## 8  fndca1.fnal.gov user7 2021-06-01  15799350 2021-06-01T01:38:42.704Z
## 9  fndca1.fnal.gov user7 2021-06-01  15799365 2021-06-01T01:38:43.484Z
## 10 fndca1.fnal.gov user7 2021-06-01  15799347 2021-06-01T01:38:43.939Z
```

In RStudio this looks just fine. There is a built in interface for up and down and is very readable. What shows up in the knitted PDF is less so. Here is where we introduce the first package, `kableExtra`. Loading packages is accomplished with the `library()` function and the name of the package, which here is just `library(kableExtra)`. If this or a future package is not present on your computer then you have to install it via `install.packages("kableExtra")`, and replacing "kableExtra" with whatever the name of the package is you need.

```
#install.packages("kableExtra")
library(kableExtra)
```

This package grants the user powerful tools for building and manipulating potentially complex tables. I am by no means a master at it, but I can make it work. Here I have split up the first 10 rows into column groups that I know will scale well after some trial and error. Now, the formatting arguments that go into `kable()` can best be understood via demonstration. For the first group with 9 columns I will add each major item one at a time so it is clear what each is doing. Note that this function makes use of the pipe operator `%>`, which functions similar to piper operators in other languages. The operators allow users to daisy chaining statements together in a manner easy to read and plan. In this case, the output of `kable` is piped to and becomes in the input for `kable_styling()`.

```
kable(head(data, 10)[,1:9])
```

disk	user	date	process_id	timestamp	duration	file_size	username
fndca1.fnal.gov	user7	2021-06-01	15799375	2021-06-01T01:38:24.685Z	1586.5020001	931116781	username7
fndca1.fnal.gov	user7	2021-06-01	15799345	2021-06-01T01:38:28.001Z	1595.3920000	933136977	username7
fndca1.fnal.gov	user7	2021-06-01	15799378	2021-06-01T01:38:28.039Z	1636.2349999	929503075	username7
fndca1.fnal.gov	user7	2021-06-01	15799297	2021-06-01T01:38:28.793Z	0.1860001	946870800	username7
fndca1.fnal.gov	user7	2021-06-01	15799360	2021-06-01T01:38:32.539Z	1032.2809999	916277785	username7
fndca1.fnal.gov	user7	2021-06-01	15799361	2021-06-01T01:38:37.935Z	0.1849999	928435505	username7
fndca1.fnal.gov	user7	2021-06-01	15799263	2021-06-01T01:38:41.200Z	0.1870000	928064238	username7
fndca1.fnal.gov	user7	2021-06-01	15799350	2021-06-01T01:38:42.704Z	1825.8680000	939293593	username7
fndca1.fnal.gov	user7	2021-06-01	15799365	2021-06-01T01:38:43.484Z	1262.8490000	925442761	username7
fndca1.fnal.gov	user7	2021-06-01	15799347	2021-06-01T01:38:43.939Z	2270.1490002	940422386	username7

```
kable(head(data, 10)[,1:9]) %>%
  kable_styling(latex_options=c("hold_position", "scale_down"))
```

disk	user	date	process_id	timestamp	duration	file_size	username	application
fndca1.fnal.gov	user7	2021-06-01	15799375	2021-06-01T01:38:24.685Z	1586.5020001	931116781	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799345	2021-06-01T01:38:28.001Z	1595.3920000	933136977	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799378	2021-06-01T01:38:28.039Z	1636.2349999	929503075	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799297	2021-06-01T01:38:28.793Z	0.1860001	946870800	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799360	2021-06-01T01:38:32.539Z	1032.2809999	916277785	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799361	2021-06-01T01:38:37.935Z	0.1849999	928435505	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799263	2021-06-01T01:38:41.200Z	0.1870000	928064238	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799350	2021-06-01T01:38:42.704Z	1825.8680000	939293593	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799365	2021-06-01T01:38:43.484Z	1262.8490000	925442761	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799347	2021-06-01T01:38:43.939Z	2270.1490002	940422386	username7	nuescatargen

```
kable(head(data, 10)[,1:9], booktabs = T) %>%
  kable_styling(latex_options=c("hold_position", "scale_down"))
```

disk	user	date	process_id	timestamp	duration	file_size	username	application
fndca1.fnal.gov	user7	2021-06-01	15799375	2021-06-01T01:38:24.685Z	1586.5020001	931116781	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799345	2021-06-01T01:38:28.001Z	1595.3920000	933136977	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799378	2021-06-01T01:38:28.039Z	1636.2349999	929503075	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799297	2021-06-01T01:38:28.793Z	0.1860001	946870800	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799360	2021-06-01T01:38:32.539Z	1032.2809999	916277785	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799361	2021-06-01T01:38:37.935Z	0.1849999	928435505	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799263	2021-06-01T01:38:41.200Z	0.1870000	928064238	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799350	2021-06-01T01:38:42.704Z	1825.8680000	939293593	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799365	2021-06-01T01:38:43.484Z	1262.8490000	925442761	username7	nuescatargen
fndca1.fnal.gov	user7	2021-06-01	15799347	2021-06-01T01:38:43.939Z	2270.1490002	940422386	username7	nuescatargen

```
kable(head(data, 10)[,1:9], booktabs = T, linesep = "") %>%
  kable_styling(latex_options=c("hold_position", "scale_down"))
```

disk	user	date	process_id	timestamp	duration	file_size	username	application
fnlcal.fnal.gov	user7	2021-06-01	15799375	2021-06-01T01:38:24.685Z	1586.5020001	931116781	username7	nuescatargen
fnlcal.fnal.gov	user7	2021-06-01	15799345	2021-06-01T01:38:28.001Z	1595.3920000	933136977	username7	nuescatargen
fnlcal.fnal.gov	user7	2021-06-01	15799378	2021-06-01T01:38:28.039Z	1636.2349999	929503075	username7	nuescatargen
fnlcal.fnal.gov	user7	2021-06-01	15799297	2021-06-01T01:38:28.793Z	0.1860001	946870800	username7	nuescatargen
fnlcal.fnal.gov	user7	2021-06-01	15799360	2021-06-01T01:38:32.539Z	1032.2809999	916277785	username7	nuescatargen
fnlcal.fnal.gov	user7	2021-06-01	15799361	2021-06-01T01:38:37.935Z	0.1849999	928435505	username7	nuescatargen
fnlcal.fnal.gov	user7	2021-06-01	15799263	2021-06-01T01:38:41.200Z	0.1870000	928064238	username7	nuescatargen
fnlcal.fnal.gov	user7	2021-06-01	15799350	2021-06-01T01:38:42.704Z	1825.8680000	939293593	username7	nuescatargen
fnlcal.fnal.gov	user7	2021-06-01	15799365	2021-06-01T01:38:43.484Z	1262.8490000	925442761	username7	nuescatargen
fnlcal.fnal.gov	user7	2021-06-01	15799347	2021-06-01T01:38:43.939Z	2270.1490002	940422386	username7	nuescatargen

```
kable(head(data, 10)[,10:14], booktabs = T, linesep = "") %>%
  kable_styling(latex_options=c("hold_position", "scale_down"))
```

version	final_state	site	rate	project_name
v08_41_00	consumed	us_fnal.gov	0.5868992	dpershey_mcc11_snb_monoenergetic_marley_wbkg_v2_20210531202650
v08_41_00	consumed	us_fnal.gov	0.5848951	dpershey_mcc11_snb_monoenergetic_marley_wbkg_v2_20210531202650
v08_41_00	consumed	us_fnal.gov	0.5680743	dpershey_mcc11_snb_monoenergetic_marley_wbkg_v2_20210531202650
v08_41_00	transferred	us_fnal.gov	5090.7002502	dpershey_mcc11_snb_monoenergetic_marley_wbkg_v2_20210531202650
v08_41_00	consumed	us_fnal.gov	0.8876244	dpershey_mcc11_snb_monoenergetic_marley_wbkg_v2_20210531202650
v08_41_00	transferred	us_fnal.gov	5018.5718495	dpershey_mcc11_snb_monoenergetic_marley_wbkg_v2_20210531202650
v08_41_00	transferred	us_fnal.gov	4962.9094019	dpershey_mcc11_snb_monoenergetic_marley_wbkg_v2_20210531202650
v08_41_00	consumed	us_fnal.gov	0.5144367	dpershey_mcc11_snb_monoenergetic_marley_wbkg_v2_20210531202650
v08_41_00	consumed	us_fnal.gov	0.7328214	dpershey_mcc11_snb_monoenergetic_marley_wbkg_v2_20210531202650
v08_41_00	consumed	us_fnal.gov	0.4142558	dpershey_mcc11_snb_monoenergetic_marley_wbkg_v2_20210531202650

```
kable(head(data, 10)[,15:16], booktabs = T, linesep = "") %>%
  kable_styling(latex_options=c("hold_position", "scale_down"))
```

file_name	data_tier
mcc11_snb_singlephase_monoenergetic_marley_45.75MeV_wbkg_cb286820-068d-440b-939f-174abc26f0e4_reco.root	full-reconstructed
mcc11_snb_singlephase_monoenergetic_marley_56.75MeV_wbkg_ae1b373d-6fa7-49fc-8425-394c7bf30124_reco.root	full-reconstructed
mcc11_snb_singlephase_monoenergetic_marley_42.25MeV_wbkg_dc5f1615-2a42-49f2-9056-005e1ef79bf3_reco.root	full-reconstructed
mcc11_snb_singlephase_monoenergetic_marley_98.75MeV_wbkg_a27bfdf-d-a1be-40a0-a914-8710c9146d12_reco.root	full-reconstructed
mcc11_snb_singlephase_monoenergetic_marley_5.75MeV_wbkg_746948c0-530f-4f0e-85f4-317be6244dd6_reco.root	full-reconstructed
mcc11_snb_singlephase_monoenergetic_marley_35.75MeV_wbkg_82d1211d-99d2-4912-8188-21a55fac0b2f_reco.root	full-reconstructed
mcc11_snb_singlephase_monoenergetic_marley_38.75MeV_wbkg_3319c7bc-27a7-437a-8104-3edc515fe1be_reco.root	full-reconstructed
mcc11_snb_singlephase_monoenergetic_marley_76.75MeV_wbkg_8816f17a-73af-400f-96bc-e5af1f630319_reco.root	full-reconstructed
mcc11_snb_singlephase_monoenergetic_marley_21.25MeV_wbkg_38ff0bac-51da-4202-b94e-7154717d8589_reco.root	full-reconstructed
mcc11_snb_singlephase_monoenergetic_marley_70.75MeV_wbkg_e525a631-8eae-46b8-baea-99aea0ed91fc_reco.root	full-reconstructed

```
kable(head(data, 10)[,17:19], booktabs = T, linesep = "") %>%
  kable_styling(position = "center", latex_options=c("hold_position"))
```

You may have noticed that these tables don't turn out well in RStudio. This is because they're formatted under the hood with code that will not fully compile until the document is knitted.

The column names in our data frame can be extracted with the `names()` function. Since a data frame is in many ways also a matrix, `colnames()` would also work.

node	country	campaign
dpershey-45488713-0-fnpc19124.fnal.gov	fnal	mcc11
dpershey-45488686-0-fnpc7676.fnal.gov	fnal	mcc11
dpershey-45488646-0-fnpc7676.fnal.gov	fnal	mcc11
dpershey-45488690-0-fnpc19121.fnal.gov	fnal	mcc11
dpershey-45488735-0-fnpc19116.fnal.gov	fnal	mcc11
dpershey-45488722-0-fnpc19122.fnal.gov	fnal	mcc11
dpershey-45488712-0-fnpc19134.fnal.gov	fnal	mcc11
dpershey-45488718-0-fnpc19121.fnal.gov	fnal	mcc11
dpershey-45488693-0-fnpc19138.fnal.gov	fnal	mcc11
dpershey-45488687-0-fnpc7541.fnal.gov	fnal	mcc11

```
names(data)
```

```
## [1] "disk"      "user"      "date"      "process_id" "timestamp"
## [6] "duration"  "file_size" "username"   "application" "version"
## [11] "final_state" "site"      "rate"      "project_name" "file_name"
## [16] "data_tier" "node"      "country"   "campaign"
```

With as many data points as we are dealing with is unlikely an analyst can scan through a particle column and identify all of its unique elements. Luckily a function with an easily remembered name can be used to do this for us. Below I am selecting a column named “site” and using `unique()` to identify its unique elements, which I am then passing to the variable `sites`. What is new here is that I have wrapped the whole operation in parentheses. It is not obvious, but what this does is automatically print the content being exchanged without having to explicitly call the variable on a new line.

```
(sites <- unique(data$site))
```

```
## [1] "us_fnal.gov"      "us_colorado.edu"   "uk_brunel.ac.uk"
## [4] "ch_cern.ch"       "cz_farm.particle.cz" "nl_farm.nikhef.nl"
## [7] "uk_gridpp.rl.ac.uk" "uk_sheffield.uk"   "uk_pp.rl.ac.uk"
## [10] "nl_gina.surfsara.nl" "us_unl.edu"         "us_crush.syracuse.edu"
```

```
length(sites)
```

```
## [1] 12
```

Let’s also look at some other variables.

```
unique(data$data_tier)
```

```
## [1] "full-reconstructed" "raw"      "generated"
## [4] "simulated"          "detector-simulated" "reco-recalibrated"
```

Going forward we will need to load the package `tidyverse`. This package contains several packages that will be useful ahead. I would normally set `message=F` at the beginning of the code chunk, but for educational purposes I will allow it.

```
#install.packages("tidyverse") # This packages takes a while to download and install
library(tidyverse)
```

```
## -- Attaching packages ----- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.3      v purrr  0.3.4
## v tibble  3.1.2      v dplyr  1.0.6
## v tidyr   1.1.3      v stringr 1.4.0
## v readr   1.4.0      v forcats 0.5.1
```

```
## -- Conflicts ----- tidyverse_conflicts() --
```

```
## x dplyr::filter()      masks stats::filter()
## x dplyr::group_rows() masks kableExtra::group_rows()
## x dplyr::lag()         masks stats::lag()
```

Now maybe we are only interested in sites that are dealing with “raw” data. There are a couple ways one might do that, but here we can use a function called `filter()`. What this function does is take in our data frame and retain only rows in which a value in one or more named columns satisfy certain conditions. The output of this function will be another data frame. Since I still expect this to be rather large I have no desire to hold on to it after the code chunk, so I will remove it. Below I perform a ratio on the number of rows that contain “raw” data to the total number of arrows, I identify the unique sites, and I count them.

```
df2 <- filter(data,data_tier=="raw")
```

```
list(
  nrow(df2)/nrow(data),
  unique(df2$site),
  length(unique(df2$site))
)
```

```
## [[1]]
## [1] 0.02906645
##
## [[2]]
## [1] "uk_gridpp.rl.ac.uk"      "nl_gina.surfsara.nl"    "uk_pp.rl.ac.uk"
## [4] "uk_brunel.ac.uk"        "us_colorado.edu"        "us_fnal.gov"
## [7] "ch_cern.ch"             "uk_sheffield.uk"       "cz_farm.particle.cz"
## [10] "us_crush.syracuse.edu"
##
## [[3]]
## [1] 10
```

```
rm(df2)
```

Instead of defining `df2` and subsequently removing it I can nest the expression used to define `df2` in place of `df2` within the functions I am using above. As this can quickly become messy, `tidyverse` provides grammatical tools for piping the outputs of functions along as inputs to subsequent function. Sometimes this latter approach is inefficient or seemingly pointless, but it begins to shine as steps in data manipulation become more extensive and complicated.

```
list(

  data %>%
    filter(data_tier=="raw") %>%
    nrow()/(data %>% nrow()),

  data %>%
    filter(data_tier=="raw") %>%
    pull(site) %>% # Used to extract column by name as a vector
    unique(),

  data %>%
    filter(data_tier=="raw") %>%
    pull(site) %>%
    n_distinct()

)

## [[1]]
## [1] 0.02906645
##
## [[2]]
## [1] "uk_gridpp.rl.ac.uk"      "nl_gina.surfsara.nl"    "uk_pp.rl.ac.uk"
## [4] "uk_brunel.ac.uk"        "us_colorado.edu"        "us_fnal.gov"
## [7] "ch_cern.ch"             "uk_sheffield.uk"        "cz_farm.particle.cz"
## [10] "us_crush.syracuse.edu"
##
## [[3]]
## [1] 10
```

Here is a more extensive filter in which I then select only specific columns of interest.

```
data %>%
  filter(duration > 100,
    site == "us_fnal.gov",
    rate > 1,
    application == "neutronana") %>%
  select(c(1:3,6,9,12)) # Can use column names or numbers

##           disk user      date duration application      site
## 1 fndca1.fnal.gov user8 2021-06-27 4512.720  neutronana us_fnal.gov
## 2 fndca1.fnal.gov user8 2021-06-27 1974.668  neutronana us_fnal.gov
## 3 fndca1.fnal.gov user8 2021-06-27 1516.448  neutronana us_fnal.gov
## 4 fndca1.fnal.gov user8 2021-06-27  445.648  neutronana us_fnal.gov
## 5 fndca1.fnal.gov user8 2021-06-27  593.685  neutronana us_fnal.gov
## 6 fndca1.fnal.gov user8 2021-06-27  827.249  neutronana us_fnal.gov
```

Additional operations allow for the calculation of summary variables

```
data %>%
  filter(country == "us") %>%
  add_count(application,site) %>% # Adds column "n" that tallies enclosed unique variable combination a
```



```
group_by(application,site,n) %>% # Defines the unique row quanta over which later operations apply
summarize("duration mean" = mean(duration),
          "duration median" = median(duration),
          .groups = "keep") %>% # Creates named column with function output
mutate("(mean-median)/mean" = (`duration mean`-`duration median`)/`duration mean`) %>% # Creates new
arrange(desc(abs(`(mean-median)/mean`))), sort = T) # Sorts rows based on values of selected variable()
```

```
## # A tibble: 15 x 6
## # Groups:   application, site, n [15]
##   application site      n 'duration mean' 'duration media~ '(mean-median)/~
##   <chr>        <chr>    <int>          <dbl>          <dbl>          <dbl>
## 1 neutronana  us_colo~    58          11447.          3465.          0.697
## 2 testutils   us_unl.~    15          46960.          24746.          0.473
## 3 protonmcnorw us_colo~     5           443.           292.          0.341
## 4 checkcnn     us_colo~    23           263.           200.          0.239
## 5 testutils   us_colo~    23          20873.          16677.          0.201
## 6 twocrtpd     us_colo~    19           330.           266.          0.194
## 7 michelstudy us_colo~    17           419.           339.          0.192
## 8 protonbeama~ us_colo~     5          3195.          3781.         -0.184
## 9 reco         us_unl.~   111          51864.          44917.          0.134
## 10 demo        us_colo~     4          10162.          9664.          0.0490
## 11 neutronana  us_crus~   337          51934.          54276.         -0.0451
## 12 reco        us_colo~    99          31625.          30475.          0.0364
## 13 protonbeama~ us_unl.~    57           3048.           3129.         -0.0265
## 14 demo        us_crus~     2          18382.          18382.           0
## 15 mytestanaly~ us_colo~     2           143.           143.           0
```

useful function for cycling through a large amount of data is the function `map()`, which belongs a class of similar functions that are characterized by their outputs. For example, `\texttt{map_chr()}` creates an output that is a vector characters, and `\texttt{map_dbl()}` produces an output of numeric, or doubles. The original function `map()` has a list as an output. Here is code that creates a list with elements named after the unique disks, where each element contains a vector of the sites associated with that disk.

```
disks <- unique(data$disk)
disks_by_site <- map(disks, ~ unique(filter(data, disk==.)$site))
names(disks_by_site) <- disks
disks_by_site
```

```
## $fndca1.fnal.gov
## [1] "us_fnal.gov"          "us_colorado.edu"      "uk_brunel.ac.uk"
## [4] "ch_cern.ch"           "cz_farm.particle.cz"  "nl_farm.nikhef.nl"
## [7] "uk_gridpp.rl.ac.uk"   "uk_sheffield.uk"      "uk_pp.rl.ac.uk"
## [10] "nl_gina.surfsara.nl"  "us_unl.edu"           "us_crush.syracuse.edu"
##
## $eospublic.cern.ch
## [1] "uk_gridpp.rl.ac.uk"   "uk_pp.rl.ac.uk"       "uk_brunel.ac.uk"
## [4] "ch_cern.ch"           "cz_farm.particle.cz"  "uk_sheffield.uk"
##
## $xrootd.echo.stfc.ac.uk
## [1] "uk_brunel.ac.uk"      "uk_gridpp.rl.ac.uk"   "uk_pp.rl.ac.uk"
## [4] "uk_sheffield.uk"
```

## Color

The package `paletteer` contains the palettes of many other color packages.

```
#install.packages("paletteer")
library(paletteer)
numUsers <- length(unique(data$user))
paletteer_d("jcolors::rainbow", numUsers, type="continuous")
paletteer_d("pals::polychrome", type="discrete")
paletteer_d("pals::alphabet", numUsers, type="discrete")

#paletteer_d("Polychrome::glasbey", type="discrete")
#paletteer_d("pals::glasbey", type="discrete")

filter(palettes_d_names, package == "wesanderson")
filter(palettes_d_names, length == 32, type == "qualitative",
       package != "palettetown")

paletteer_d("wesanderson::Zissou1", 5, type="discrete")
```

<https://mokole.com/palette.html>

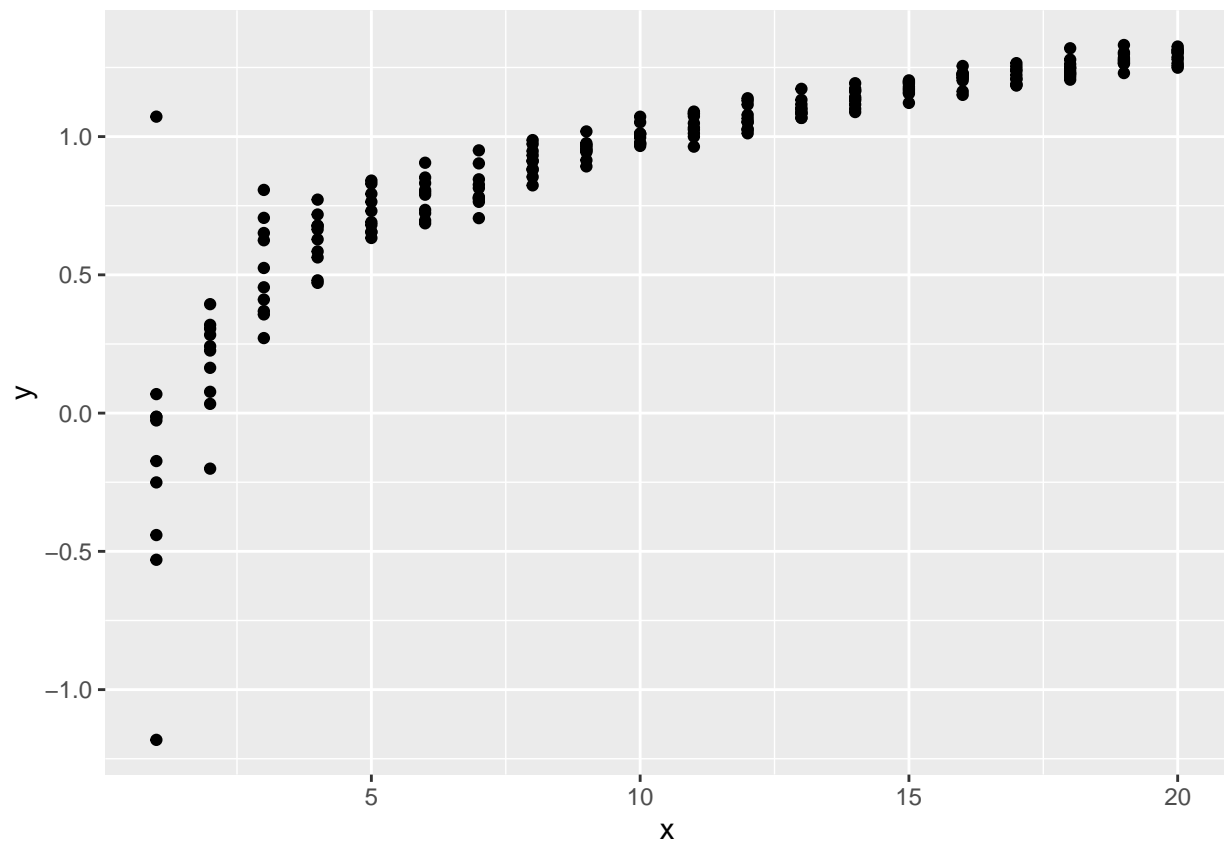
```
colors19 <- c("#808080", "#2e8b57", "#8b0000", "#808000", "#7f007f", "#ff0000", "#ff8c00", "#7cfc00",
             "#00fa9a", "#4169e1", "#00ffff", "#00bfff", "#0000ff", "#f08080", "#ff00ff", "#ffff54",
             "#dda0dd", "#ff1493", "#f5deb3")
colors25 <- c("#dcdcdc", "#8b4513", "#006400", "#808000", "#483d8b", "#008080", "#9acd32", "#00008b",
             "#8b008b", "#ff4500", "#ffa500", "#ffff00", "#00ff00", "#00fa9a", "#dc143c", "#00ffff",
             "#00bfff", "#0000ff", "#ff00ff", "#1e90ff", "#db7093", "#f0e68c", "#ff1493", "#ffa07a", "#ee82ee")
```

## ggplot

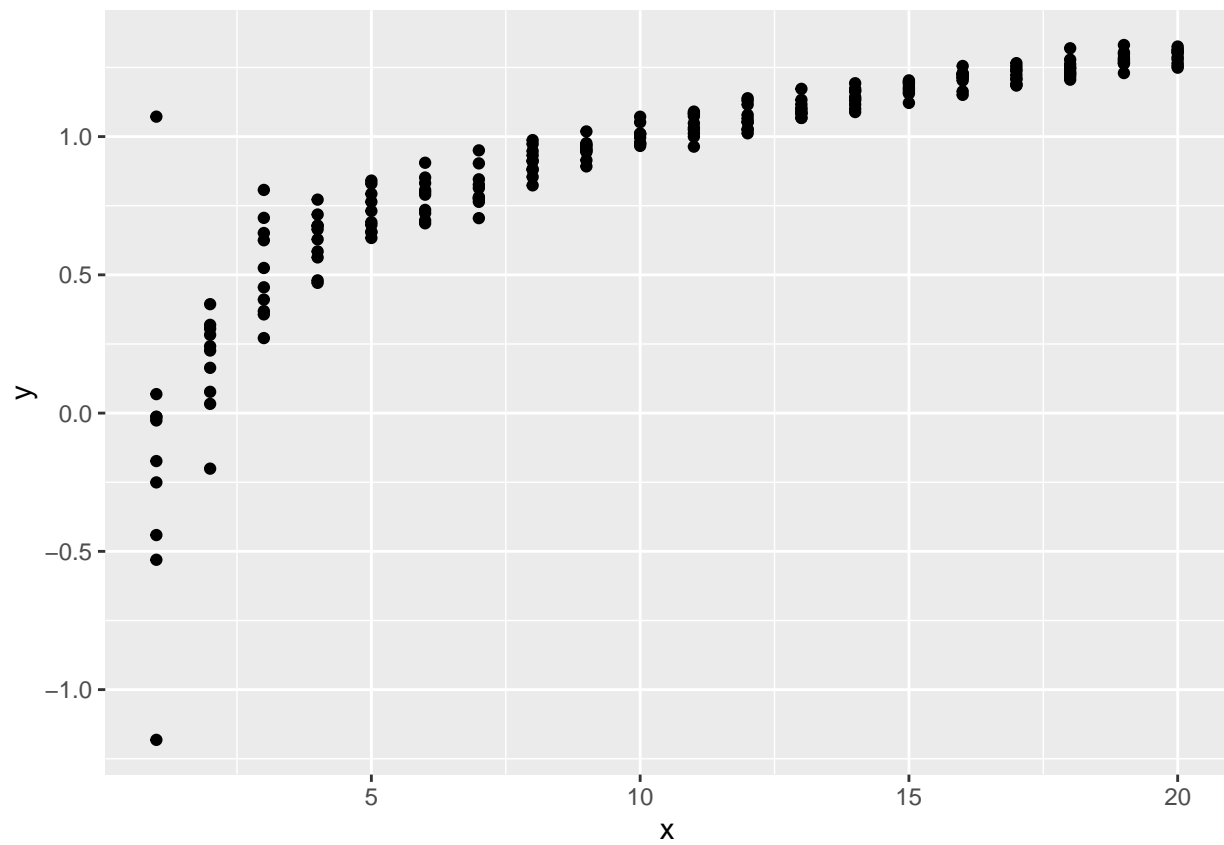
### Quick example

```
point1 <- data.frame(x = rep(1:20,10),
                    y = log10(rep(1:20,10))+rnorm(200,0,0.5)*(1/rep(1:20,10)))
line1 <- data.frame(x = seq(0.1,20,0.1),
                    y = log10(seq(0.1,20,0.1)))

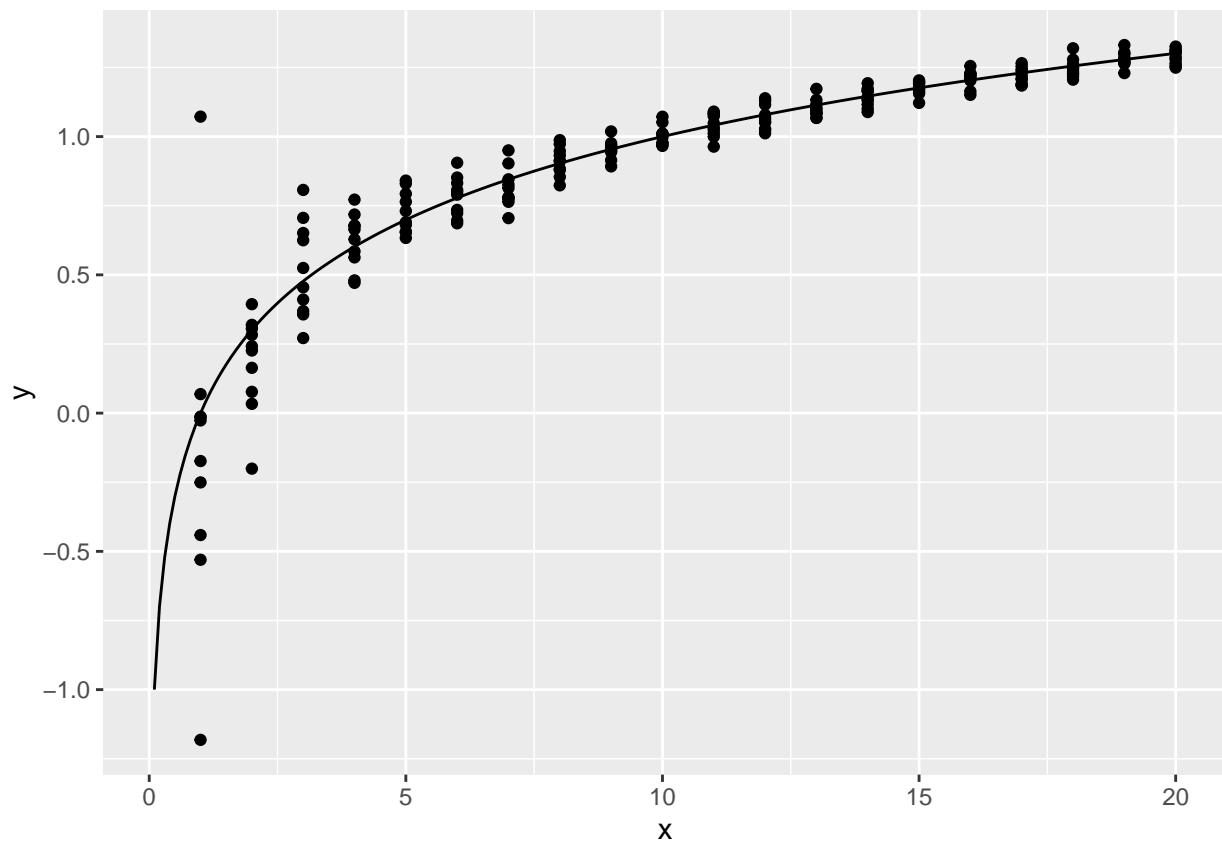
ggplot(data = point1, mapping = aes(x = x, y = y)) +
  geom_point()
```



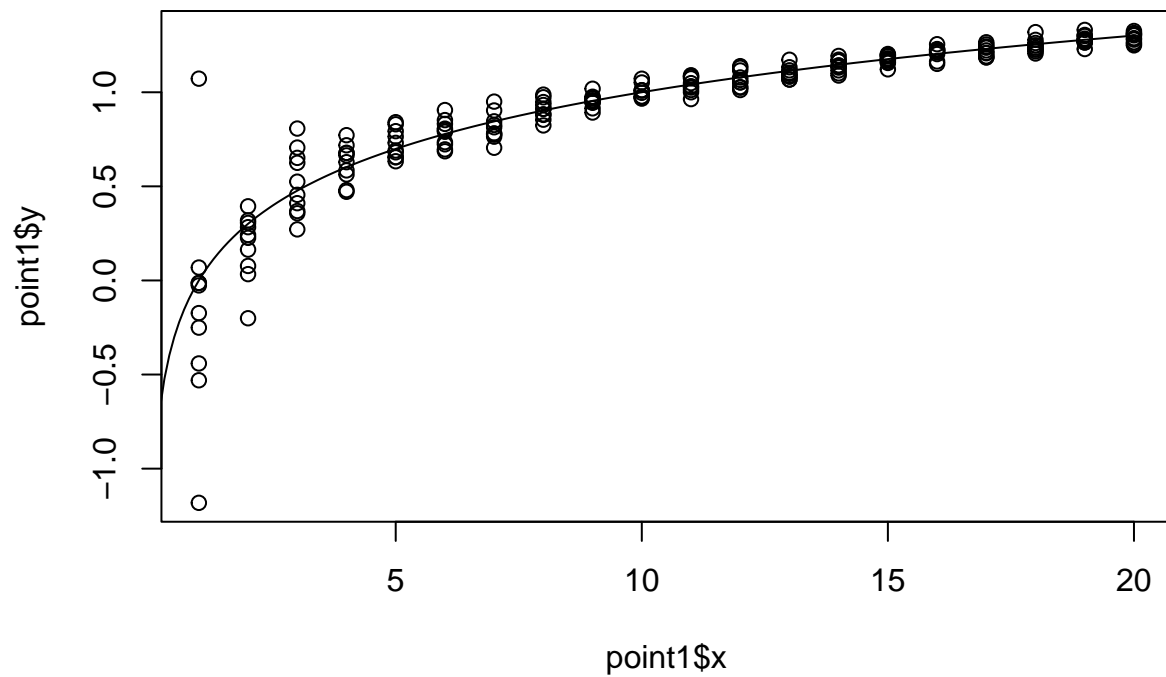
```
ggplot() +  
  geom_point(data = point1, mapping = aes(x=x, y=y))
```



```
ggplot() +  
  geom_point(data = point1, mapping = aes(x=x, y=y)) +  
  geom_line(data = line1, mapping = aes(x=x, y=y))
```

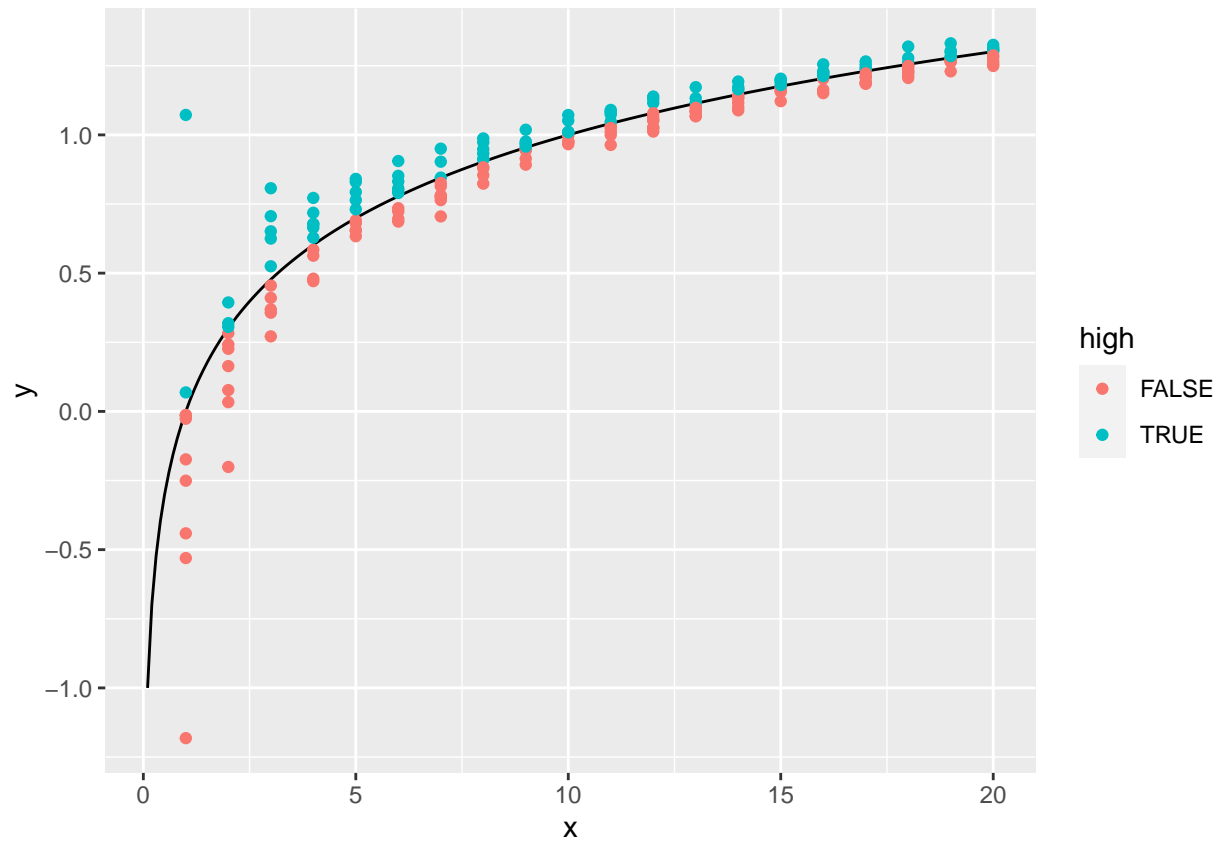


```
plot(point1$x, point1$y)
lines(line1$x,line1$y)
```

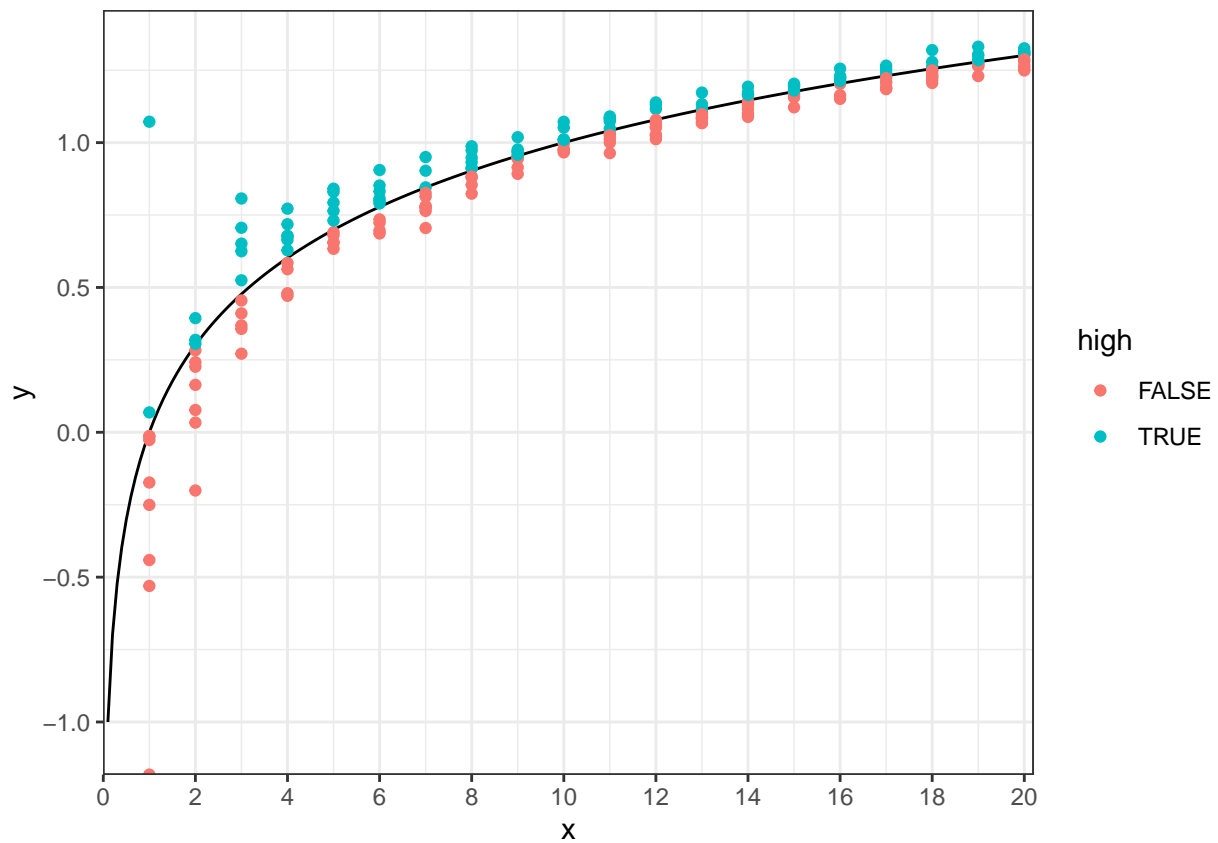


```
point1$high <- point1$y > log10(point1$x)
```

```
ggplot() +  
  geom_line(data = line1, mapping = aes(x=x, y=y)) +  
  geom_point(data = point1, mapping = aes(x=x, y=y, color=high))
```

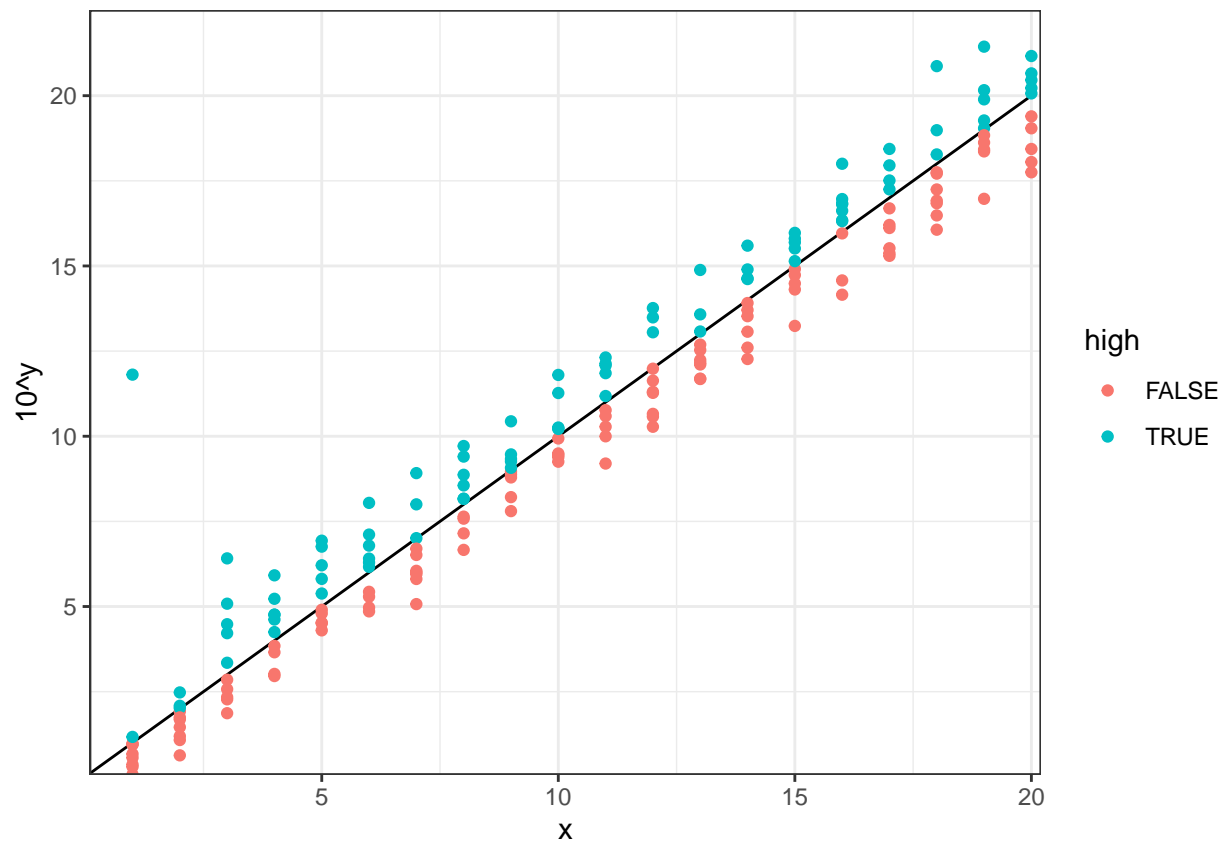


```
ggplot() +  
  geom_line(data = line1, mapping = aes(x=x, y=y)) +  
  geom_point(data = point1, mapping = aes(x=x, y=y, color=high)) +  
  scale_x_continuous(limits=c(0,20.2), expand=c(0,0), breaks = seq(0,20,2)) +  
  scale_y_continuous(expand=expansion(mult = c(0, .05))) +  
  theme_bw()
```



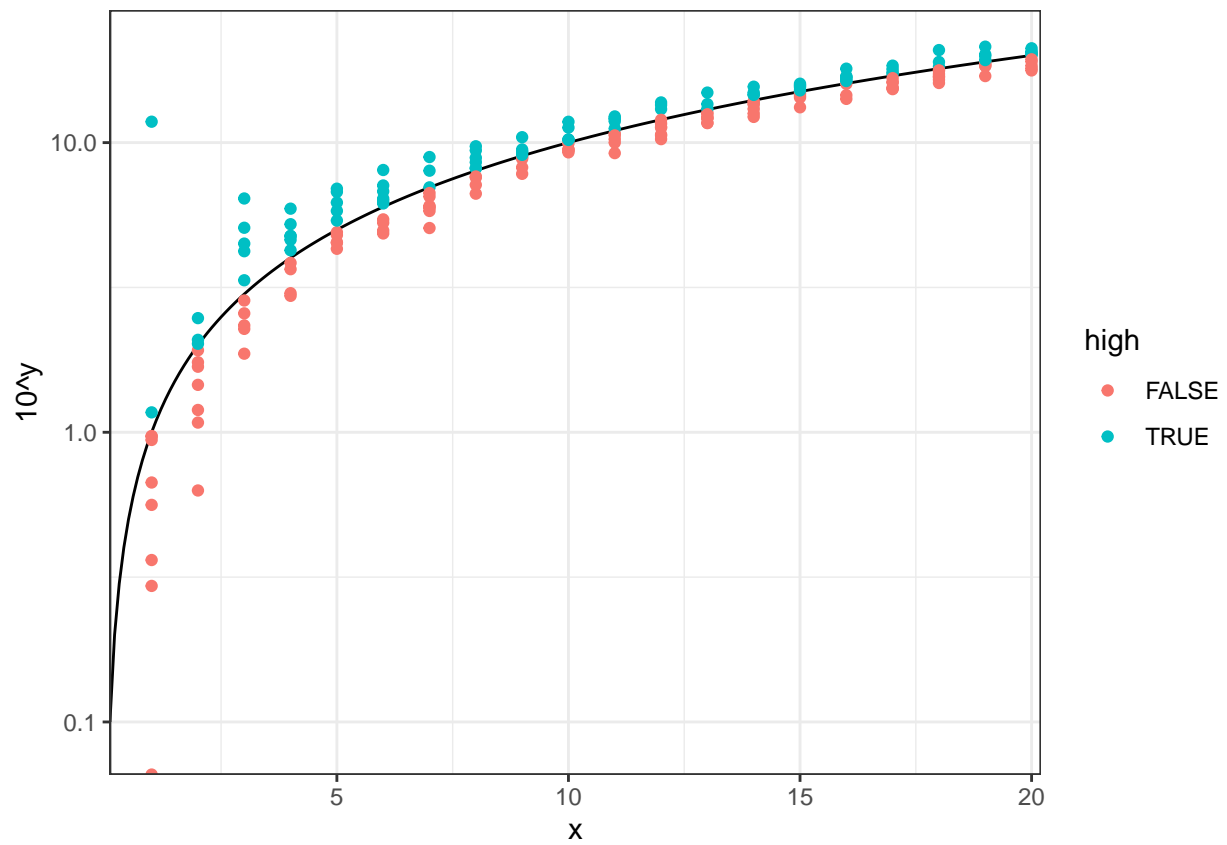
Transforming variables and axes

```
ggplot() +
  geom_line(data = line1, mapping = aes(x=x, y=10^y)) +
  geom_point(data = point1, mapping = aes(x=x, y=10^y, color=high)) +
  scale_x_continuous(expand=expansion(mult = c(0, .01))) +
  scale_y_continuous(expand=expansion(mult = c(0, .05))) +
  theme_bw()
```



```
ggplot() +
  geom_line(data = line1, mapping = aes(x=x, y=10^y)) +
  geom_point(data = point1, mapping = aes(x=x, y=10^y, color=high)) +
  scale_x_continuous(expand=expansion(mult = c(0, .01))) +
  scale_y_log10(expand=expansion(mult = c(0, .05))) +
  theme_bw()
```





Looking at our data

Dealing with time

Assigning ggplot to variable

Saving plots as pngs

```

```