

# R Basic Graphs; Focus on Decoration

Dongseong Seo, 24293621

October 21, 2017

## Introduction

From the very beginning of this class, we have encountered several plotting tasks.

I assume that many of us have struggled to figure out how to make our graphs look similar to the professor's ones.

I, **indeed**, spent several **hours** (not-minutes) to dress them up to be pretty.

However, it was kind of enjoyable to choose nicer pants and shirts for them.

Whenever I found out the best matching costume, the **delight** from it was unexplainable!

I thought I should consider my career as **a fashion coordinator** not a statistician.

However, as you guys may have experienced, this daydream has been shattered by the mighty package `ggplot2`.

This package is surely user-friendly, and the outputs of it are amazingly beautiful!

After learning `ggplot2`, it was hard for me to force myself to use R basic graphs.

This phenomenon was shared by many fellow students, and it was unstoppable.

## This is *why*

I totally agree with that using `ggplot2` seems to have more opportunities for us to generate **prettier and finer graphs**.

However, I also think that we should know the basic things first in order to prepare for unknown situations and to manage them.

Also, by doing so, we could expand our knowledge limitations and could apply **basic** concepts to **advanced** ones.

These are why I chose *R basic graphs* as my topic.

It does not mean I am going to write about various types of plot which we can generate from the basic R.

Rather than that, I am going to focus on options(**arguments**) which one can use to decorate plots.

This post would not focus on basic arguments, which we have learned during this semester, such as `xlab`, `main`, and so on.

Most of options(**arguments**) that I will introduce **can be used** in `ggplot2` with different or same invoking functions.

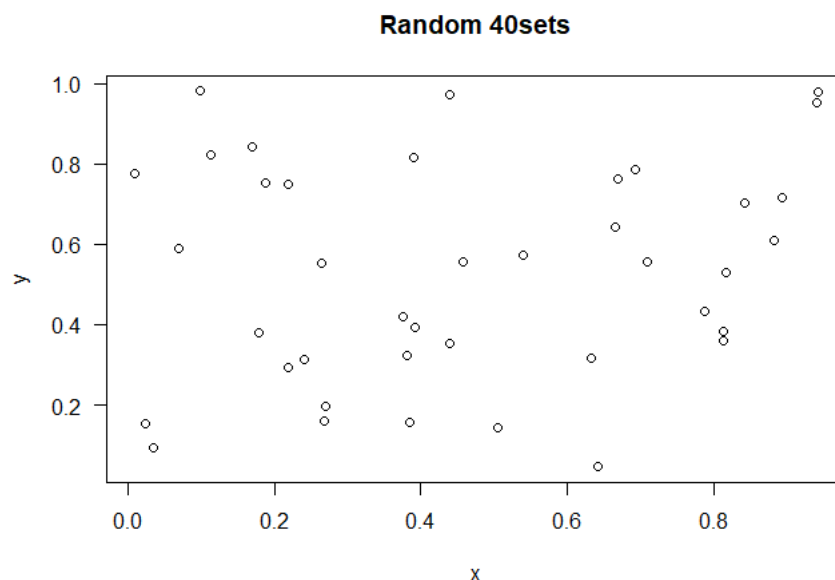
Therefore, I hope reading this post would compensate your time!

---

## Preparing a data set

First of all, let's make a data set to draw a scatter plot.

```
set.seed(12)      # Setting a seed
x <- runif(40, min = 0, max = 1)
y <- runif(40, min = 0, max = 1)
# Let's use the function runif() to pick 40 random sets of numbers from the uniform[0,1] distribution.
plot(x,y, las = 1, main = "Random 40sets")
```



OK, now we have a scatter plot.

*Keep in mind* that as we used the random number generator to produce plots, they will look different every time you knit or execute.

To avoid this, I used `set.seed()`.

Anyway, the first argument that I am going to focus is `pch`.

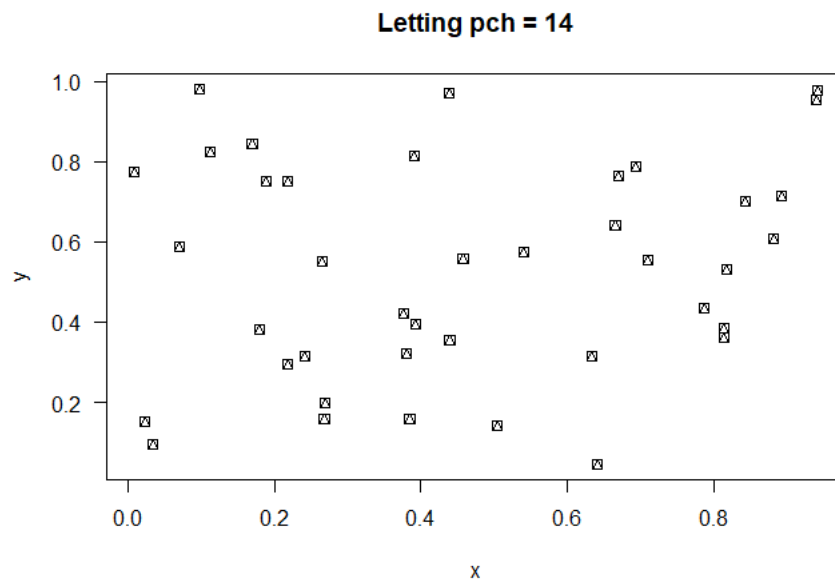
---

## pch

`pch` is basically about the *shapes* of point in a plot.

Let's produce some interesting shapes.

```
plot(x,y, las = 1, main = "Letting pch = 14",  
     pch = 14)      # pch = 14. What would it be?
```



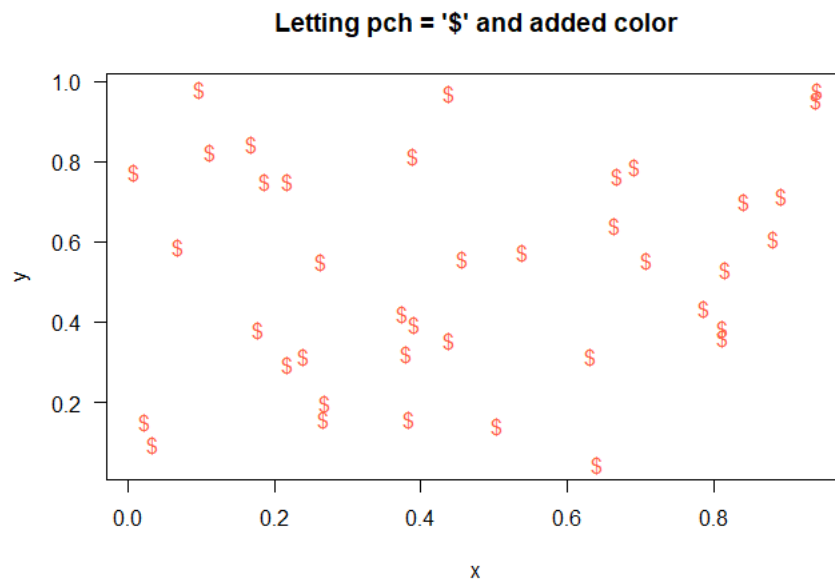
By `pch = 14`, we can see that the points' shape is changed.

In short, by letting `pch = a number` or **"a character"**, we can manipulate points' shape.

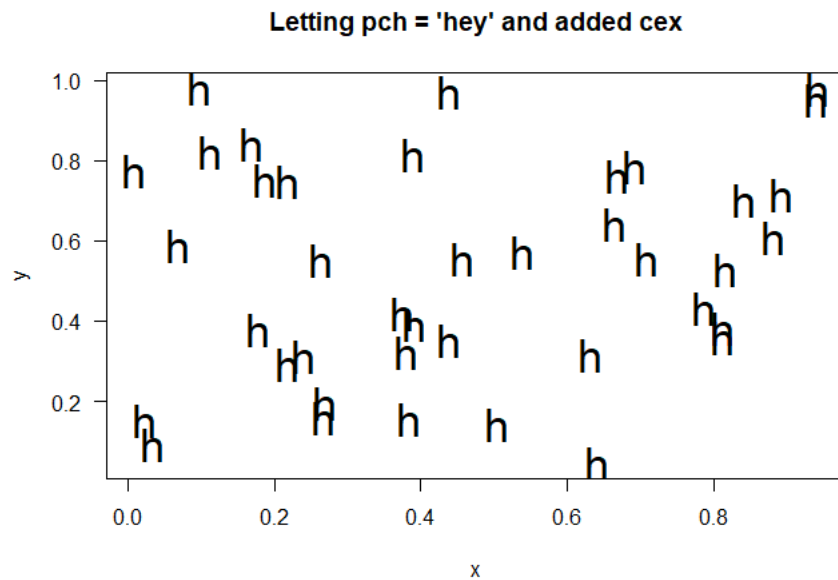
Let's try "a character" case.

- For characters, you need to use `" "` or `' '`
- Also, as you can see from the following examples, you can only use **1** character as a shape

```
plot(x,y, las = 1, main = "Letting pch = '$' and added color",  
     pch = "$",          # pch = "$"  
     col = "tomato")      # we can also add color
```



```
plot(x,y, las = 1, main = "Letting pch = 'hey' and added cex",  
     pch = 'hey',        # Only the first character, 'h', appears  
     cex = 2.14)         # cex is about the size of points
```



It's interesting isn't it?

As you may have noticed, for numbers, each number has an appointed figure to be called.

The default value of `pch` is 1 which is an unfilled circle.

[This link](#) will lead you to a magnificent world of `pch`.

By selecting and adding color, your plots would be much prettier than before.

## col

The second argument which we will explore is `col`.

When I first encountered `col`, I was kind of confused how to set the value for it.

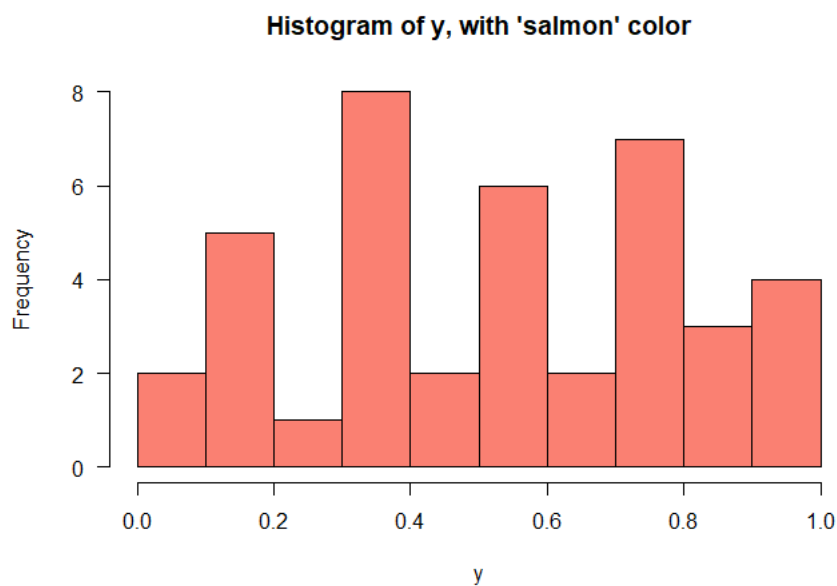
After searching on Google, [this link](#) was what I found.

As you can see, there are tremendous numbers of color from which you can choose to decorate your plots!

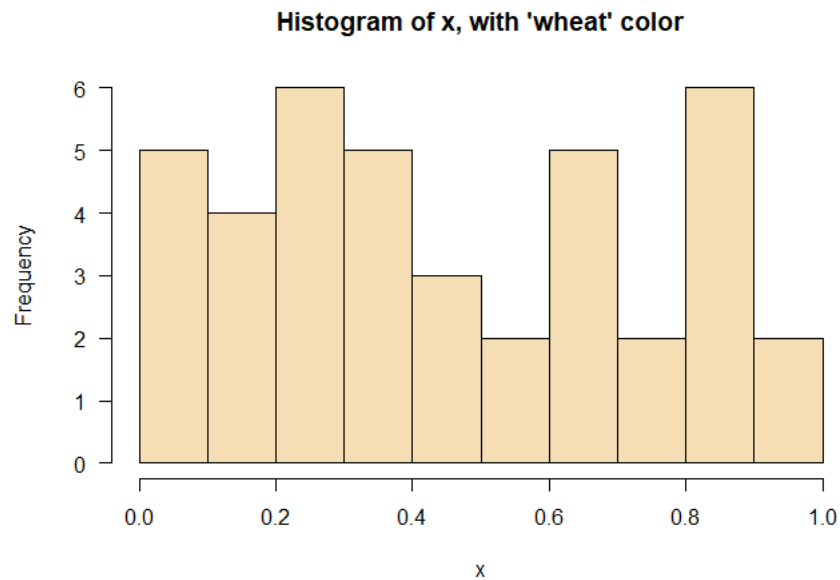
Let's play with them.

Here, I will use **histograms**, and the data set is the same as above.

```
hist(y, las = 1, main = "Histogram of y, with 'salmon' color",
     col = "salmon")           # What a name!
```



```
hist(x, las = 1, main = "Histogram of x, with 'wheat' color",
     col = "wheat")           # color name "wheat"
```



It's simple enough isn't it?

All that you need is the above link and aesthetic senses.

After printing out **the color chart** from the above link, I felt relief.

However, all of a sudden, a random number (such as `col = "#333333"`) popped up!

I could not understand what those numbers were doing.

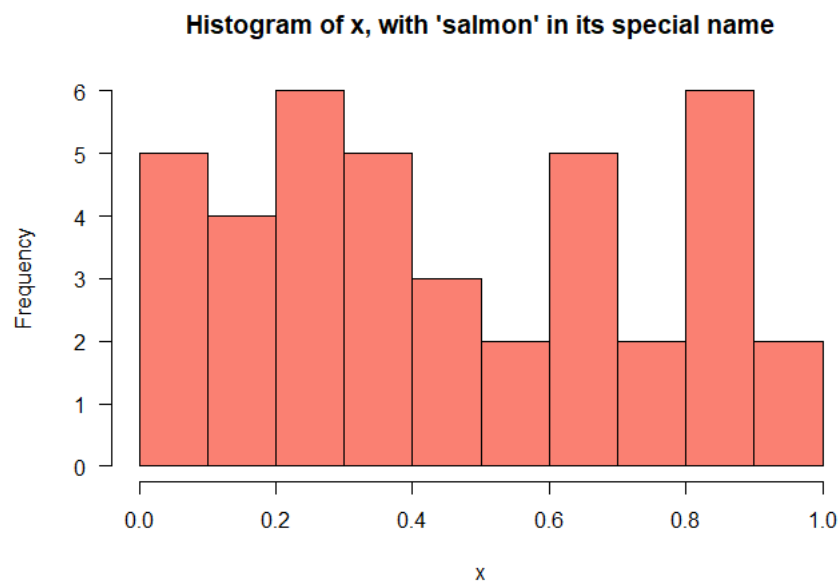
They looked even more horrendous than the color name "salmon".

Now is the time that we need to update our printout.

[This updated color chart](#) will serve this purpose!

Let's try some with our updated printout.

```
hist(x, las = 1, main = "Histogram of x, with 'salmon' in its special name",  
     col = "#FA8072")           # our salmon has a special name #FA8072
```



With our printout, this way to refer color is also simple enough!

You can just type your chosen color's digital name after # inside " " or ' '.

Up to this point, you may think that produced plots are not pretty enough.

Let's try something special.

---

## Palettes

At a glance, the word "palettes" would make you imagine something that a painter uses while painting.

Yes, you are *absolutely* right.

Like an artist, we will use **palettes packages** to generate beautifully colored plots.

The first thing to know is that there are many palettes packages in R which you can play with.

- examples: grDevices, colorRamps, RcolorBrewer, colorspace, and so on.

Among them, I will focus on the `grDevices` package.

The good news is that, you don't have to go through the installing and loading package process. In other words, `grDevices` is one of packages come with your base R installation.

In the `grDevices` package, there are 5 types of palettes which you can use:

1. `cm.colors()`
2. `topo.colors()`
3. `terrain.colors()`
4. `heat.colors()`
5. `rainbow()`

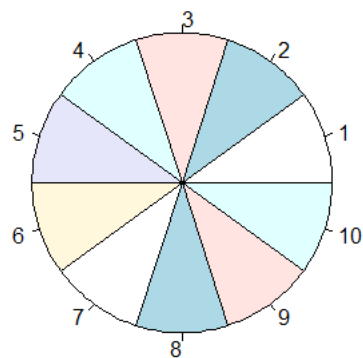
It's time to be an artist!

## `topo.colors()`

Let's take `topo.colors()` palette as an example. I will use **pie charts** as they resemble shapes of an actual palette.

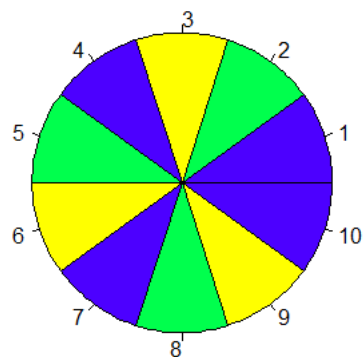
```
n <- 10
pie(rep(1,n), main = "Original pie chart") # hmm, the original pie chart looks not that bad.
```

Original pie chart



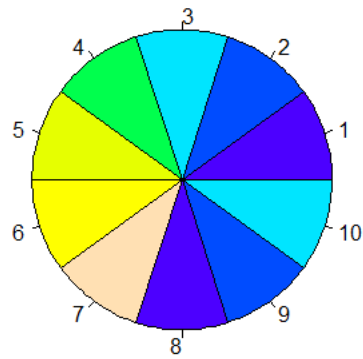
```
# But, let's add color to it.
pie(rep(1,n), main = "Pie chart with topo.colors, n = 3",
    col = topo.colors(n = 3)) # n = 3
```

Pie chart with `topo.colors`, `n = 3`



```
pie(rep(1,n), main = "Pie chart with topo.colors, n = 7",
    col = topo.colors(n = 7)) # n = 7
```

### Pie chart with topo.colors, n = 7



The argument, `n =`, inside the `topo.colors()`, represents the numbers of color picked from the `topo.colors` palette.

Let's look at `grDevices` 5 available palettes.



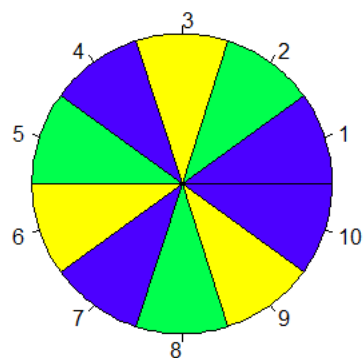
As you can notice, the `topo.colors()` palette, starts from **violet** to **apricot**.

Also, we can see a kind of **color spectrum** in between two end points, violet and apricot.

It becomes *clear* that, our input `n = some number` takes the palette, then the palette is divided by `some number`.

For example, let's consider our first pie chart again.

### Pie chart with topo.colors, n = 3



As we chose `some number = 3`, the `topo.colors()` palette is divided by 3.

What is the result?

- Yes, the *first* color is dark violet.
- Yes, the *last* color is light yellow.
- So, the *middle* color is light green!

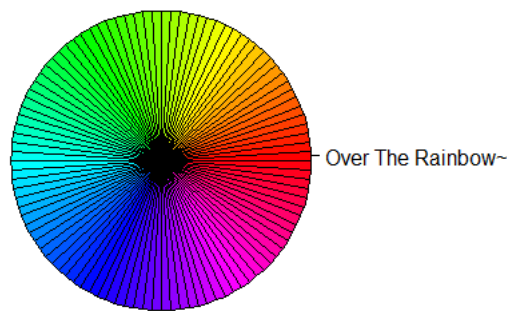
This way of mathematical thinking would violate the noble artistic world.  
However, it would give us some degrees of understanding!

## rainbow()

This time, we will try to color a pie chart with as many colors as possible from the `rainbow()` palette.

```
rainbow_n <- 100      # Let's try rainbow_n = 100
pie(rep(1, rainbow_n), labels = "Over The Rainbow~",
    main = "rainbow palette with n = 100",
    col = rainbow(n = rainbow_n) # we are using the rainbow() palette
)
```

rainbow palette with n = 100



Isn't it pretty?

Before moving on, there is something we need to know.

`rainbow()` takes more arguments than other 4 palettes in `grDevices`.

For other 4 palettes, there is one more common argument `alpha =`.

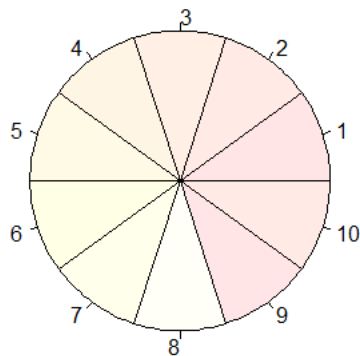
It takes a number from 0 to 1 (`rainbow()` also has the `alpha` argument).

As we have learned, `alpha =` represents transparency. As number goes closer to 0, transparency increases.

Here is the example.

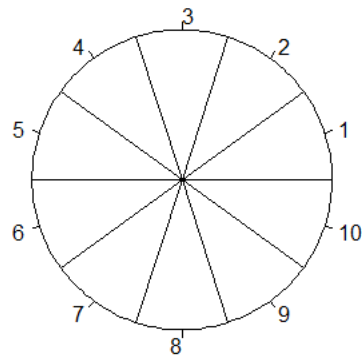
```
pie(rep(1, n), main = "heat.colors palette with alpha = 0.1",
    col = heat.colors(n = 8, alpha = 0.1)) # alpha is 0.1
```

heat.colors palette with alpha = 0.1



```
pie(rep(1,n), main = "heat.colors palette with alpha = 0",
    col = heat.colors(n = 8, alpha = 0)) # alpha = 0, the extreme case.
```

### heat.colors palette with alpha = 0

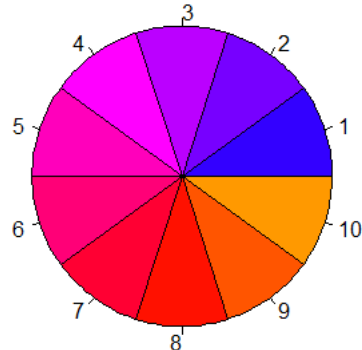


The additional argument `alpha =` can surely add beauty to your plots

Now, let's consider some of available arguments for `rainbow()`

```
pie(rep(1,n), main = "rainbow palette with start = 0.7 and end = 0.1",
    col = rainbow(n = 10, start = 0.7, end = 0.1))
```

### rainbow palette with start = 0.7 and end = 0.1



```
# We can set where to start in a rainbow palette.
# Also, we can set where to end in a rainbow palette.
```

As we can see from the above example, we can use arguments `start =` and `end =` to set where to start and end in the `rainbow()` palette.

Both arguments take number from 0 to 1 likely to `alpha =`.

As you may have noticed, the number `0.5` represents the *middle* color, blue-ish, in the `rainbow()` palette.

By the same logic, `0.1` would be orange-ish.

This can be explained by the **proportional** distance.

With the given `rainbow()` palette, we can choose a start point and an end point by letting the length of the entire palette as 1.

Then, we can choose color by using **proportional** distance.

As you know, I always provide an useful link at the end of one topic.

So, here you go. [This link](#) would give you more information about `grDevices` and also other useful coloring packages which I mentioned at the beginning.

By reading through the above link, you would be the **master of color** in R!

type



The next argument to decorate our plots is `type`.

This argument was introduced during the lecture about a week ago(?).

However, only one possible way was considered, which is `type = "l"`.

Hence, I will try to focus on other possibilities of `type`.

In order to do so, I will use scatter plots in this part.

First of all, the argument `type` is about how to represent a plot.

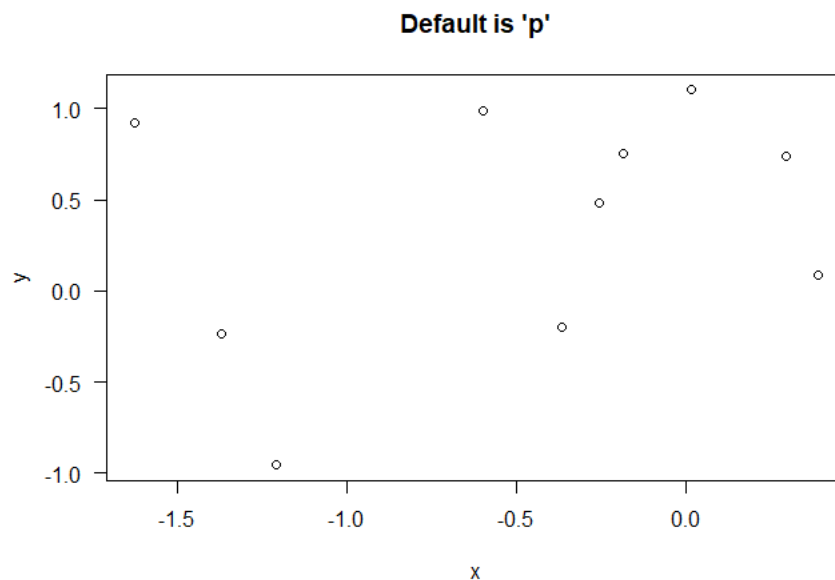
Here are possible values which it can take:

1. "p" = **P**oints (the default)
2. "l" = **L**ines
3. "b" = **B**oth points and lines
4. "c" = **C**ompletely lines alone of the type "b"
5. "o" = **O**verlay points by lines
6. "h" = **H**istogram like vertical lines
7. "n" = **N**o plotting, but basic structures(such as axes) still appear

Examples would make them clear.

Time to plot! (here, let's use `rnorm()` to generate a data set).

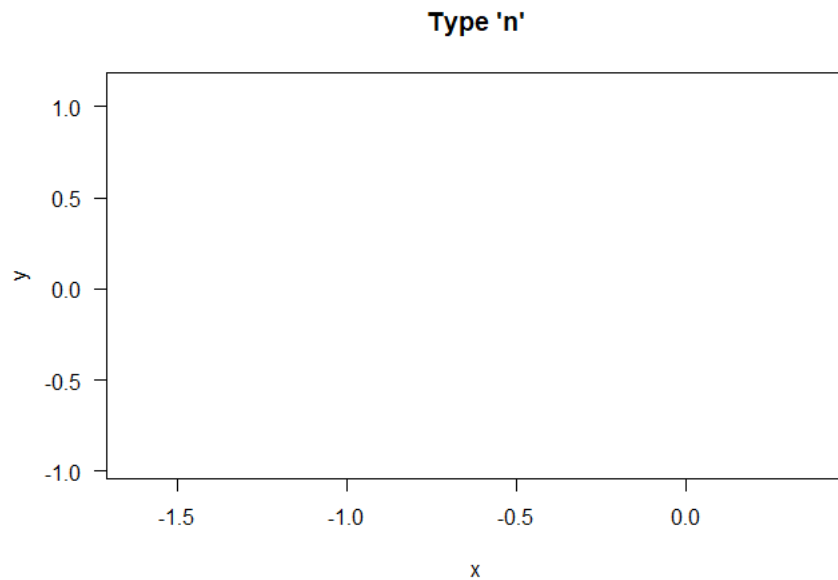
```
set.seed(10)
x_1 <- rnorm(10) # Taking random 10 numbers from a normal distribution for x.
y_1 <- rnorm(10) # Taking random 10 numbers from a normal distribution for y.
plot(x_1, y_1, main = "Default is 'p'", xlab = "x", ylab = "y",
     las = 1) # as you can see the default type is "p"oints
```



Without giving the argument `type`, we can see the default is **p**oints.

As we have tried the default one, let's try the extreme one, `type = "n"`.

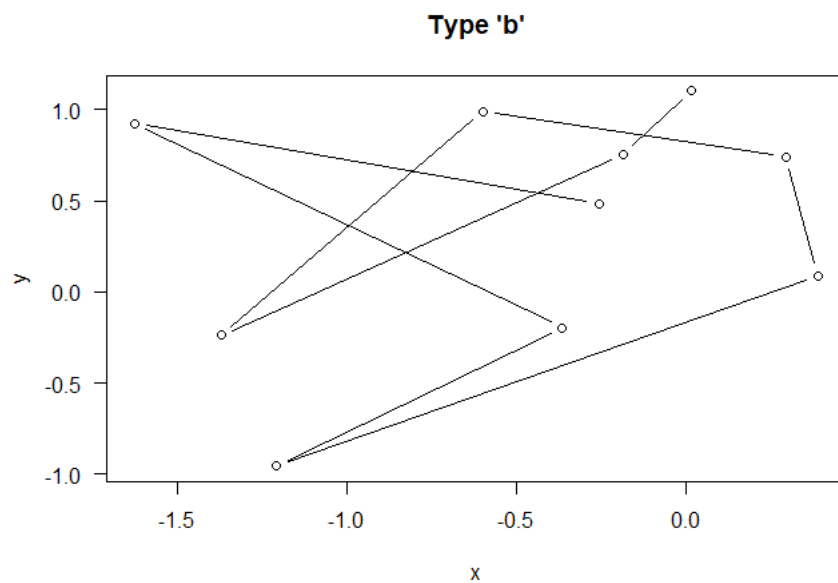
```
plot(x_1, y_1, main = "Type 'n'", xlab = "x",
     ylab = "y", las = 1,
     type = "n" # letting type = "n"
)
```



Oops, the points are gone now, but the basic structures still exist, thankfully.

The most interesting example would be comparing three types, `type = "b"`, `type = "c"` and `type = "o"`. Let's focus on the differences among them!

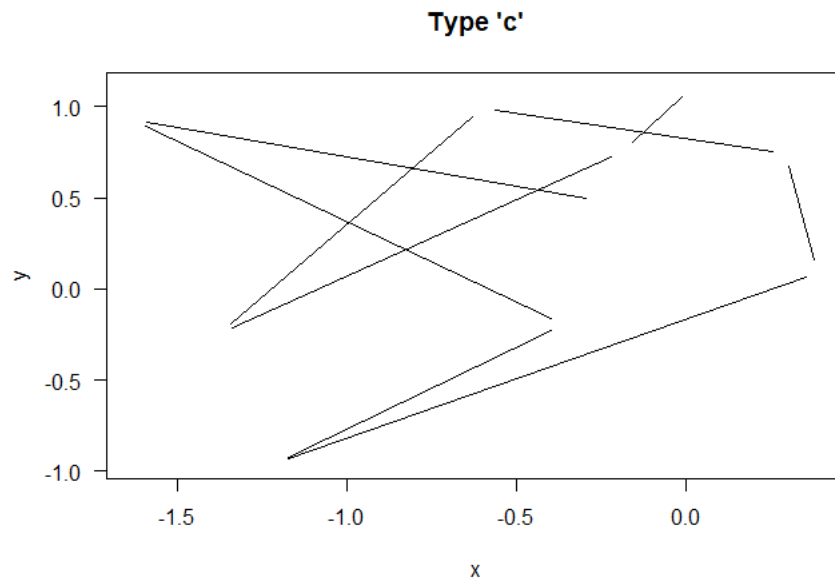
```
plot(x_1, y_1, main = "Type 'b'", xlab = "x",  
     ylab = "y", las = 1,  
     type = "b"           # letting type = "b"  
)
```



As I mentioned above, `type = "b"` represents plotting both points and lines.

OK, what about `type = "c"`?

```
plot(x_1, y_1, main = "Type 'c'", xlab = "x",  
     ylab = "y", las = 1,  
     type = "c"           # letting type = "c"  
)
```

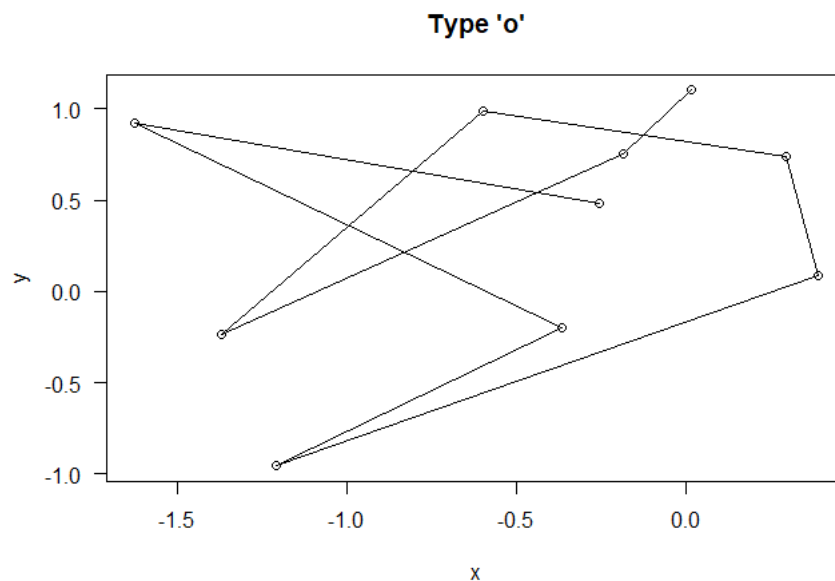


The lines are exactly the same, but points are gone now.

Only lines are left!

What about `type = "o"` ?

```
plot(x_1, y_1, main = "Type 'o'", xlab = "x",
     ylab = "y", las = 1,
     type = "o"           # letting type = "o"
)
```



Can you find the difference?

Yes, you are right.

`type = "o"` lets us plot **both** points and lines, in addition to this, the lines **overlay** points!

In terms of beauty, I prefer `type = "b"` .

However, you can choose among above three types depending on given tasks or your preference!

Before considering the last one, `type = "h"` , I want to mention one more thing, which I think interesting.

By using `type = "b"` , `type = "c"` , and `type = "o"` , we can trace plot's start point and end point.

In order to do so, let's look at the first point's coordinate, (0.0187462, 1.1017795).

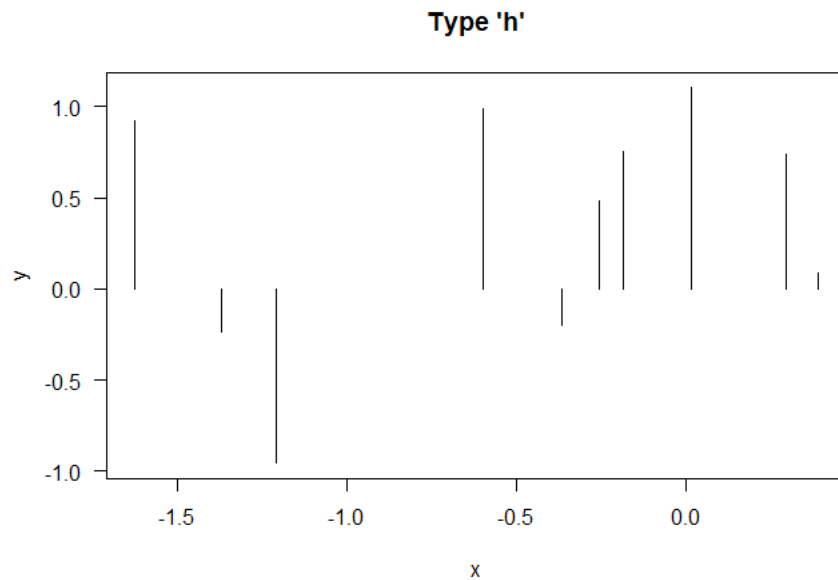
For clarification, I used an inline code above to get the first coordinate (`r x_1[1]`, `r y_1[1]` inside `` , respectively).

OK, by considering the first point's coordinate, we can find the start point.

Then, we can trace connected lines to find the end point.

One more type to cover is `type = "h"` .

```
plot(x_1, y_1, main = "Type 'h'", xlab = "x",
     ylab = "y", las = 1,
     type = "h"           # letting type = "h"
)
```



We can interpret each vertical line as a distance from the zero axis.

In other words, a vertical line starts from 0 of y-axis, and it ends where a point was located.

Here is the [reference of type = argument](#).

By choosing an appropriate type, you would be able to add more variety and beauty to plots.

### more types!

When I put a period, at the end of the final sentence above, I naively thought that I have covered all the possible types which one can use in R. Nonetheless to say, I was wrong!

**There is no end in R and data science!**

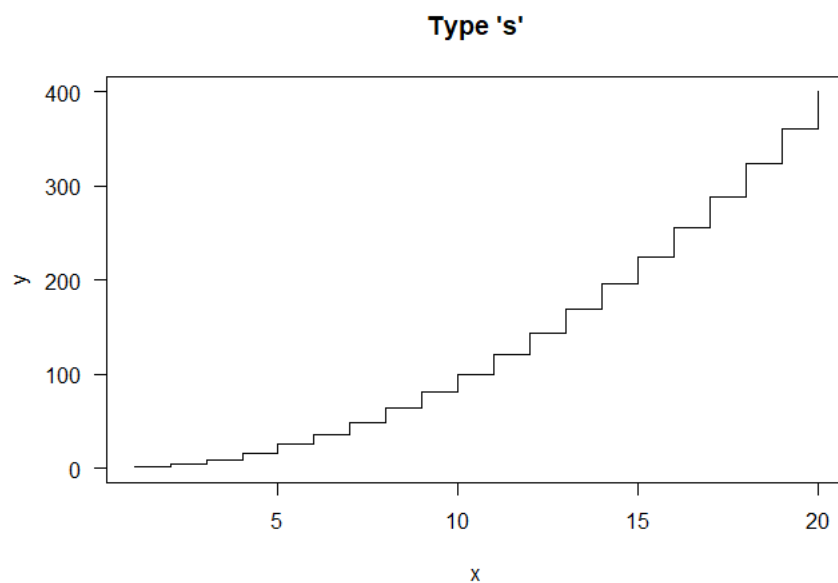
While searching for the next topic of this post, I found out that there are more available types to decorate our plots.

I found two more additional types!

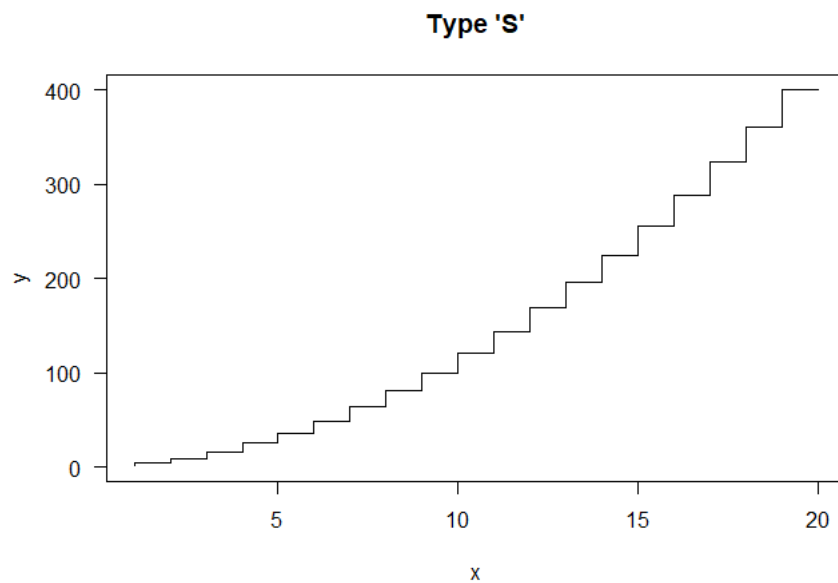
1. "s" : **S**tair steps
2. "S" : **S**tair steps, but different from the above.

Let's plot with a different data sets to see what they do!

```
x_2 <- 1:20           # new x is 1:20
y_2 <- x_2^2          # new y is x^2
plot(x_2, y_2, main = "Type 's'", xlab = "x",
     ylab = "y", las = 1,
     type = "s"       # letting type = "s"
    )
```



```
plot(x_2, y_2, main = "Type 'S' ", xlab = "x",
     ylab = "y", las = 1,
     type = "S"      # letting type = "S"
)
```



They do resemble stair steps!

Can you see the difference between `s` and `S`?

Yes, when a plot with `type = s` moves one point to another, the line moves horizontally first then vertically.

For `type = S`, the line climbs up one stair first, then it moves horizontally.

[This link](#) is the one which has taught me the lesson that there is no real end.

## lty

My final topic of this post will be the argument `lty =`.

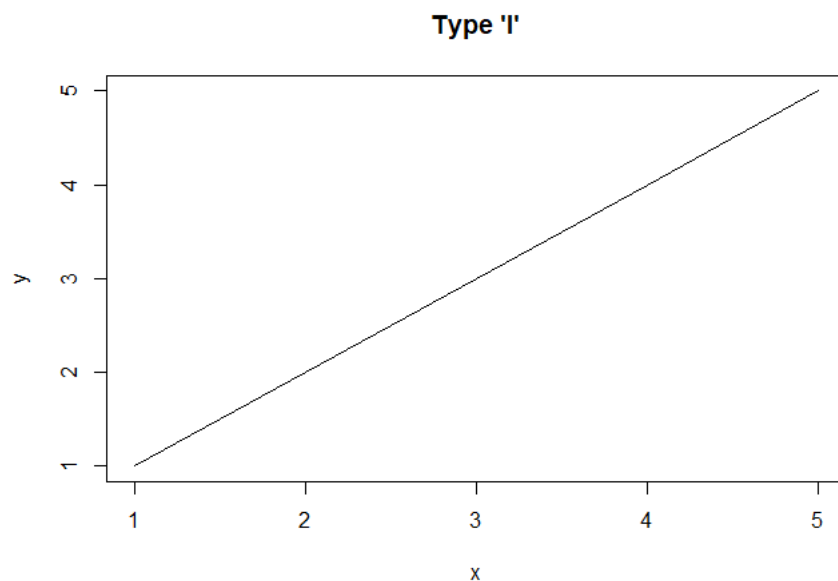
As one of the conditions of this assignment is going beyond materials covered during this semester, I could not care about `type = "l"`.

I felt sorry for `type = "l"`, so I am going to soothe it by giving a new friend, `lty =`.

As you know, it is time to **plot!**

Let's consider an easier data set.

```
x_3 <- 1:5 # x and y are 1:5
y_3 <- x_3
plot(x_3, y_3, xlab = "x", ylab = "y", main = "Type 'l'",
     type = "l") # type l
```









The title looks like "Type 1" but it is actually "Type l" (a lower case of L).

With this basic line graph, we are going to try some of the line types below.

The argument `lty` = with 7 types:

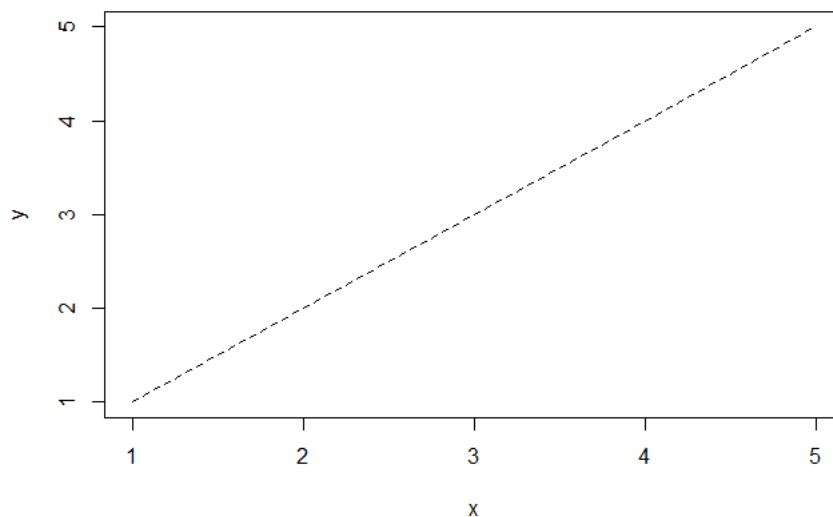
# The different line types

6. 'twodash'	
5. 'longdash'	
4. 'dotdash'	
3. 'dotted'	
2. 'dashed'	
1. 'solid'	
0. 'blank'	

Let's pick 1 cute one and 1 extreme one; 'dotted' and 'blank'.

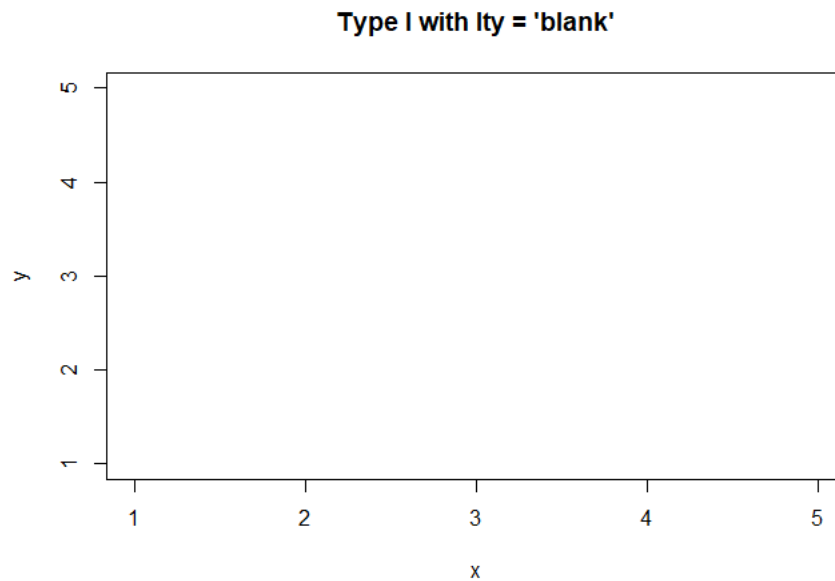
```
plot(x_3, y_3, xlab = "x", ylab = "y", main = "Type 1 with lty = 2",  
     type = "l",  
     lty = 2      # you can use either number or 'name' for lty =  
     )
```

Type 1 with lty = 2



Good, let's try the extreme one, but this time we are going to use its 'name'.

```
plot(x_3, y_3, xlab = "x", ylab = "y", main = "Type 1 with lty = 'blank'",  
     type = "l",  
     lty = 'blank'  
     )      # as we expected, blank!
```



[Here](#) is the link to the `lty =`, a friend of `type = "l"`.

## Conclusion

First of all, thank you for reading through up to this point. ~~I hope you did not just jump from the introduction to here.~~

Anyway, in this post, I started with the point that we need to know basic things before applied ones.

To deal with this purpose, I have been talking about basic R graphs (especially some of their available **arguments and options**).

Here is a brief list of **arguments and options** which have been covered:

1. `pch`
2. `col`
3. `grDevices` (one of available palette packages)
4. `type`
5. `type = "s", "S"`
6. `lty`

Although, I could not cover everything to decorate our plots in basic R, I hope the above list could provide pretty enough clothes for your plots.

Also, I hope we would consider using basic R to draw graphs, rather than going directly to the `library(ggplot2)` command.

As I mentioned in the introduction, my intention is not convincing us to use basic R rather than `ggplot2`.

Rather, this post has focused on broadening our knowledge to be competent.

Balancing knowledge between basic and applied concepts would compensate us more in the future.

Writing through this post, I have learned two important things which I want to share with you.

With mentioning these two things, I will close my post.

1. There is no **real end** in R and data science!
2. Without knowing basic things, we could not *truly* appreciate the *values* of developed **user-friendly packages**!

See you guys in the real world~

## References

1. <http://www.endmemo.com/program/R/pchsymbols.php>
2. <http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf>
3. <http://research.stowers.org/mcm/efg/R/Color/Chart/ColorChart.pdf>
4. <https://www.nceas.ucsb.edu/~frazier/RSpatialGuides/colorPaletteCheatsheet.pdf>
5. <http://www.dummies.com/programming/r/how-to-create-different-plot-types-in-r/>
6. <https://stat.ethz.ch/R-manual/R-devel/library/graphics/html/plot.html>
7. <http://www.sthda.com/english/wiki/line-types-in-r-lty>