R Base Data Visualization

Basim Alsaedi

2024-05-14

Contents

Chapter 1

INTRODUCTION TO R AND DATA VISUALIZATION

1.1 Introduction

1.1.1 Overview of R

R is a powerful programming language and environment widely used for statistical computing and graphics. It provides a wide variety of statistical and graphical techniques, and it is highly extensible. R is freely available and runs on all major platforms, making it an excellent choice for data analysis and visualization.

With its extensive libraries, R enables users to manipulate data, conduct statistical analyses, and create visualizations to explore and communicate insights effectively. Whether you're a beginner or an experienced data scientist, R provides a flexible and comprehensive environment for all your data analysis needs.

1.2 Why Use R for Data Visualization?

1.2.1 Advantages of R over other programming languages and tools

- R has thousands of packages, designed, maintained, and widely used by statisticians.
- The R graphs have much more fun compared to other tools such as STATA.

- R has a rather liberal syntax, and variables don't need to be declared as they would in (for example) C++, which makes it very easy to code in.
- R is designed to make it very easy to write functions which are applied point wise to every element of a vector. This is extremely useful in statistics.
- R is powerful: if a command doesn't exist already, you can code it yourself.

1.2.2 R's extensive package ecosystem

A package is a collection (or library) of functions, datasets, and other objects. Most packages are not loaded automatically, so you have to do it yourself. **R's** extensive package ecosystem provides a vast array of tools for data analysis and visualization. These packages are contributed by a vibrant community of developers and cover almost every aspect of data science. Some of the most popular visualization packages include:

i) **ggplot2:** The package provides an intuitive syntax for creating complex and beautiful visualizations.
ii) **plotly:** This package allows one to create interactive web-based visualizations directly from R.
iii) **ggvis:** It allows the creation of web-based visualizations with reactive features using the grammar of graphics syntax.
iv) **lattice:** It is particularly useful for creating trellis plots, which allow you to visualize relationships in multivariate data.
v) **gganimate:** The package allows you to easily add animations to your visualizations, making it ideal for exploring changes in data over time.

In this course, we will focus on the base R graphics system, which provides a solid foundation for understanding how plots are constructed in R. Once you have a good grasp of the basics, you can easily transition to more specialized packages like ggplot2 and plotly to create even more sophisticated visualizations.

1.3 Installing and Configuring R and RStudio

Before installing **RStudio** in your computer, first start with **R**. **RStudio** is a front end program that lets you write **R** code, view plots, and do many other useful things. The detailed steps below show how to install R and RStudio in your computer system on both Windows, Mac and Linux operating systems.

1.3.1 Step-by-step guide on installing R

1. Download the R installer from https://cran.r-project.org/.

a). Click on the link for your operating system. Make sure the installer is for the latest R version. For example, the latest version is 4.3.3.

b). Click install R for the first time.

c). Use the download link at the top and save the file.

2. Run the installer (double click), default settings are fine.

1.3.2 Step-by-step guide on installing RStudio and Set Up

1. Wait until the R installer has finished. 2.Download RStudio installer from the official website https://posit.co/download/rstudio-desktop/.
2. After the download is complete, double-click on the installer and follow the installation steps to install it in your computer.

After successful installation, you can launch RStudio by double-clicking the RStudio icon on your desktop or from the Start menu. You can install the packages using the install.packages() function. However, for the base R visualization, you don't need to install any package which will support in plotting of graphs except where we you will be required to use data that comes with R packages. The last set up may be setting your working directory to the folder where your R scripts and data files are located. This makes it easier to access your files but it is optional.

1.4 Basic R Concepts.

1.4.1 Introduction to R syntax and basic commands

R is designed for statistical computing and graphics. In this section, we will cover some basic syntax and commands to help you get started with R.

1. R as a Calculator:

You can use R as a simple calculator. Here are some basic arithmetic operations:

```r
# Addition
2 + 3
# Subtraction
5 - 2
# Multiplication
2 * 3
# Division
6 / 2
# Exponentiation
2^3
```

2. Assigning Values to Variables:

You can store values in variables using the assignment operator <- or =.

```r
# Assigning a value to a variable
x <- 5
y <- 3
```

```
# You can also use =
z = x + y
```

3. Basic Data Types:

R supports several basic data types, including numeric, character, logical, and complex.

```
# Numeric
num <- 10
# Character
char <- "Hello, Basim!"
# Logical
logic <- TRUE
# Complex
comp <- 3 + 2i
```

4. Vectors:

A vector is a sequence of data elements of the same basic type. You can create a vector using the **c()** function.

```
# Creating a numeric vector
nums <- c(1, 2, 3, 4, 5)
# Creating a character vector
chars <- c("apple", "banana", "orange")
```

5. Indexing and Slicing:

You can access elements of a vector using square brackets [].

```
# Accessing elements of a vector
nums <- c(1, 2, 3, 4, 5)
nums[1]   # Access the first element of nums
nums[2:4] # Access the second to fourth elements of nums
nums[1,4,6] # Access the first, fourth and sixth elements of nums
```

6. Functions:

R has a large number of built-in functions, and you can also create your own functions for some tasks that can not be achieved by built-in functions.

```
# Built-in function
sqrt(16)   # Square root function
nums <- c(1, 2, 3, 4, 5)
mean(nums) # the mean of nums elements
var(nums) # the variance
sd(nums) # standard deviation
# User-defined function
add <- function(a, b) {
  return(a + b)
}
add(3, 5)
```

1.4.2 Overview of R's data types and structures essential for visualization

R provides several data types and structures that are essential for data visualization. Understanding these data types and structures is crucial for effectively analyzing and visualizing data.

1. Numeric:

Numeric data type is used to represent continuous numerical values.

```
num <- 5.6
```

2. Integer:

Integer data type is used to represent integer values.

```
int <- 10L
```

3. Character:

Character data type is used to represent text data.

```
char <- "Hi, John!"
```

4. Logical:

Logical data type is used to represent Boolean values (**TRUE** or **FALSE**).

```
logic <- TRUE
```

5. Vector:

A vector is a sequence of data elements of the same basic type. It is created using the concatenate **c()** command.

```
nums <- c(1, 2, 3, 4, 5) # A vector of numeric data type
chars <- c("apple", "banana", "orange")  # A vector of character data type
```

6. Matrix:

A matrix is a two-dimensional array with rows and columns in that order respectively, that is $(R \times C\ )$. It is created using the **matrix()** command.

```
mat <- matrix(1:12, nrow = 3, ncol = 4) # A matrix with three rows and 2 columns
```

7. Data Frame:

A data frame is a two-dimensional data structure with rows and columns, similar to a spreadsheet. It can contain several data types. It is created using the command **data.frame()** command.

```
df <- data.frame(
  Name = c("John", "Alice", "Bob"),
  Age = c(25, 30, 35),
  Height = c(175, 160, 180)
)
```

8. List:

A list is an ordered collection of objects (which may be of different types: numeric, character, etc.). It is created using the **list()** command.

```
lst <- list(
  Name = c("John", "Alice", "Bob"),
  Age = c(25, 30, 35),
  Height = c(175, 160, 180)
)
```

9. Factors:

Factors are used to represent categorical data. It is created using the **factor()** command.

```
gender <- factor(c("Male", "Female", "Male", "Female"))
loan_default<-factor(c('Yes', 'No'))
```

1.5 Practical Examples: Exercises

Exercise 1: Basic arithmetic

Create a variable called x and give it the value 15.Take the exponent of the variable and add 5 to the final result. Print the final result of x.

```
# Your code
```

Exercise 2: Vectors:

The weights of five people before and after a diet programme are given in the table below. Read the ‘before’ and ‘after’ values into two different vectors called **before** and **after**. Use **R** to evaluate the amount of weight lost for each participant. What is the average amount of weight lost?

Before	78	72	78	79	105
After	67	65	79	70	93

```
#Your code
```

Exercise 3: Matrices

Create two matrices called **A** and **X** defined below.

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{pmatrix}$$

$$X = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

Find:

i) the product of **A** and **X**.

ii) the transpose of **A**.

iii) the determinant of **A**.

iv) the diagonal of **A**.

v) the inverse of **A**.

```
#Your code
```

Exercise 4: Creating Data Frames

(a) Create a small data frame representing a database of films. It should contain the fields **title**, **director**, **year**, **country**, and at least three films.

(b) Create a second data frame of the same format as above, but containing just one new film.

(c) Merge the two data frames using rbind().

(d) Try sorting the titles using sort(): what happens?

```
#Your code
```

Exercise 5: Factors and Simple plot

Suppose we have the heights of 100 individuals, where the first are 50 male and the rest female. Generate 100 fixed random numbers from a normal distribution where the mean height of male is 170 while that of female is 160 with an equal standard deviation of 10 and call that vector as ‘height’. Create another vector called ‘sex’ with two entries ‘M’ and ‘F’ each replicated 50 times. Tell R to treat ‘sex’ as a categorical variable and name it as ‘Sex’. Plot **Sex** against **height** using **plot()** function. Which type of plot have you obtained? What happens if you try to plot **sex** against **height** instead?

```
#Your code
```

Chapter 2

R BASE GRAPHICS - STARTING WITH THE BASICS

2.1 Exploring Base Graphics in R

2.1.1 Overview of the philosophy behind R's base graphics system, including its stateful nature

2.1.1.1 Stateful Nature

R's base graphics system is stateful, meaning that plots are built up incrementally. You start with an empty plot and add elements to it one by one. This is in contrast to systems like ggplot2, which use a declarative approach where you specify the plot all at once. Stateful nature means that every new plotting command modifies the existing plot or creates a new one if none exists. This allows for a high degree of flexibility but can sometimes lead to complex and intuitive behavior.

R's base graphics system provides a set of low-level graphics primitives for creating plots. These primitives include functions for drawing points, lines, polygons, text, and more. By combining these primitives, you can create a wide variety of plots, from simple scatter plots to complex multi-panel layouts.

2.1.1.2 Philosophy

The philosophy behind R's base graphics system is to provide a flexible and powerful tool for creating a wide range of plots. The emphasis is on simplicity and ease of use, making it easy for users to quickly create informative visualizations.

However, despite that base graphics are powerful, they do have some limitations compared to more modern plotting systems like ggplot2. For instance, they lack some of the advanced features of ggplot2, such as automatic faceting and easy customization of plot themes.

2.1.2 Introduction to Core Plotting Functions in R

R provides a variety of core plotting functions that are useful for creating basic visualizations. In this section, we will explore some of the most commonly used plotting functions. The commonly used core plotting functions are **plot()**, **hist()**, **boxplot()** and **barplot()**.

- The **plot()** function is used to create scatter plots, line plots, and other types of plots. It is a versatile function that can be used to visualize relationships between two or more variables.
- The **hist()** function is used to create histograms, which are used to visualize the distribution of a single numeric variable. Note that the optional argument breaks chooses (approximately) how many bins the histogram should have, and col alters the colour of the bars.
- The **boxplot()** function is used to create box plots, which are used to visualize the distribution of a numeric variable, optionally broken down by a categorical variable.
- The **barplot()** function is used to create bar plots, which are used to visualize the distribution of a categorical variable.

2.2 Creating Basic Plots

2.2.1 Detailed instructions on using plot() for scatter plots and line graphs

In order to get the full documentation of the **plot()** function, run **?plot** in R. The syntax for the function is given below.

```
# plot(x, y = NULL, type = "p",  xlim = NULL, ylim = NULL,
#     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
#     ann = par("ann"), axes = TRUE, frame.plot = axes,
#     panel.first = NULL, panel.last = NULL, asp = NA,
#     xgap.axis = NA, ygap.axis = NA,
#     ...)
```

Arguments

- x, y: the x and y arguments provide the **x** and **y** coordinates for the plot.
- *type*: 1-character string giving the type of plot desired."p" for points, “l” for lines “b” for both points and lines, “c” for empty points joined by lines, “o” for overplotted points and lines, “s” and “S” for stair steps and “h” for

histogram-like vertical lines. Finally, "n" does not produce any points or lines.

- *xlim*, *ylim*: the **x** and **y** limits respectively.
- *log*: a character string which contains "x" if the x axis is to be logarithmic, "y" if the y axis is to be logarithmic and "xy" or "yx" if both axes are to be logarithmic.
- *main* : a main title for the plot.
- *sub*: a subtitle for the plot.
- *xlab*, *ylab* : The labels for x-axis and y-axis respectively, defaults to a description of x and y.

The commonly used graphical parameters are:

- *col* : The colors for lines and points. Multiple colors can be specified so that each point can be given its own color. If there are fewer colors than points they are recycled in the standard fashion. Lines will all be plotted in the first colour specified.

- *bg* : a vector of background colors for open plot symbols

- *pch*: a vector of plotting characters or symbols

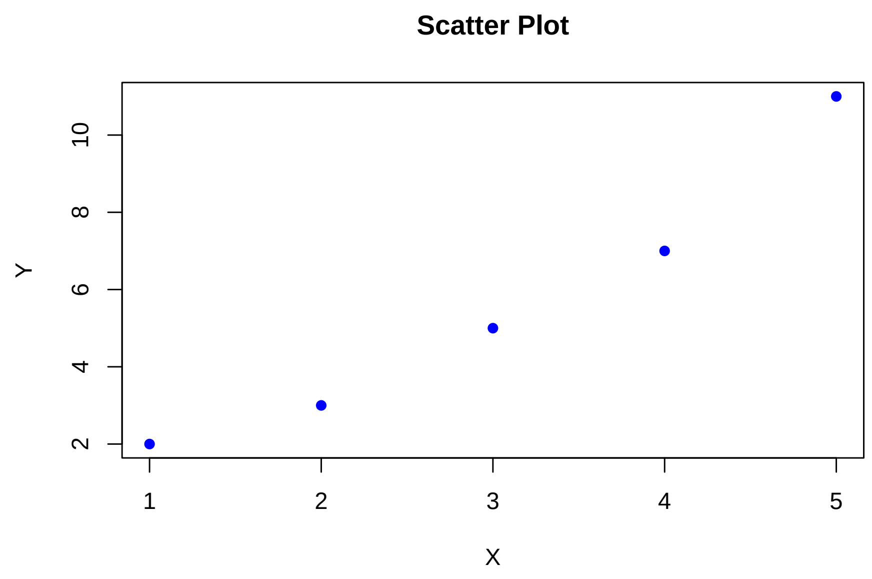
- *cex* : a numerical vector giving the amount by which plotting characters and symbols should be scaled relative to the default.

- *lty*: a vector of line types

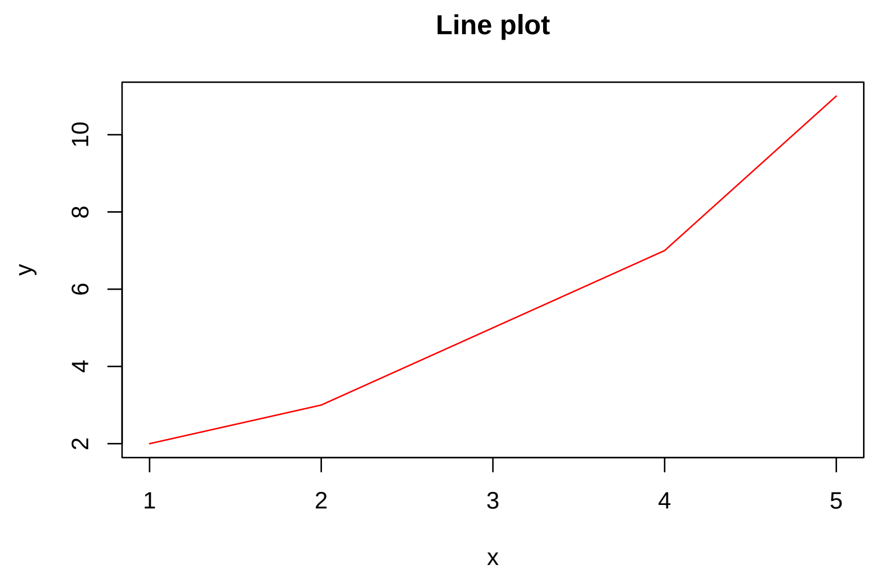
- *lwd*: a vector of line widths

Example:

```r
# Create a scatter plot
x <- c(1, 2, 3, 4, 5)
y <- c(2, 3, 5, 7, 11)
plot(x, y, type = "p", col = "blue", pch = 16, main = "Scatter Plot", xlab = "X", ylab = "Y")
```



```
# Create a line plot
plot(x, y, type = "l", col = "red", lwd = 1, main = "Line plot", xlab = NULL, ylab = NU
```

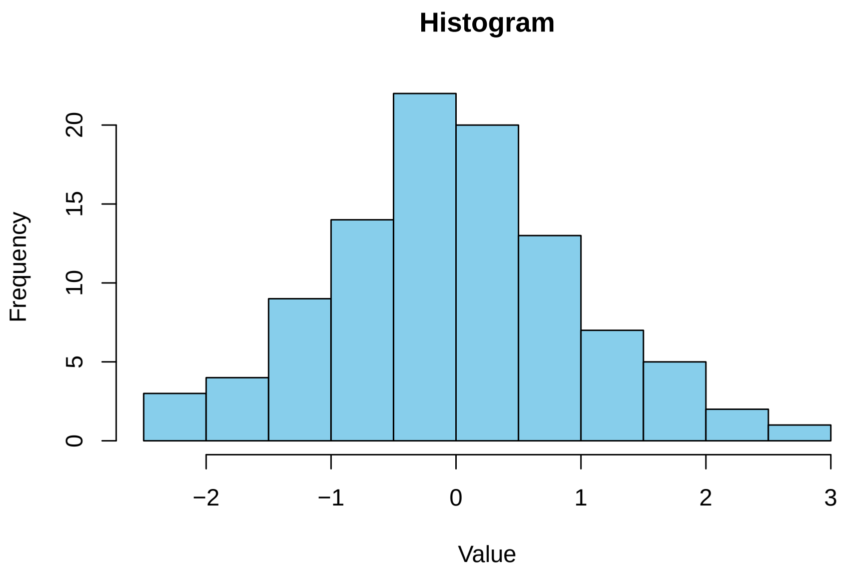


2.2.2 Using hist() to create histograms for data distribution analysis

Histograms are useful for visualizing the frequency distribution of a single variable. Similarly, you can get more documentation of the **hist()** function by running the command **?hist** in R. The generic function **hist** computes a histogram of the given data values.The default behavior for a histogram is to display frequencies on the vertical axis; probability densities can be displayed using the **freq=FALSE** option. The default title is given by **paste("Histogram of" , x)** where **x** is the name of the variable being plotted; this can be changed with the main option. The common syntax for the function is given below

```
#hist(x, breaks = "Sturges",
#     freq = NULL, probability = !freq,
#     include.lowest = TRUE, right = TRUE, fuzz = 1e-7,
#     density = NULL, angle = 45, col = "lightgray", border = NULL,
#     main = paste("Histogram of" , xname),
#     xlim = range(breaks), ylim = NULL,
#     xlab = xname, ylab,
#     axes = TRUE, plot = TRUE, labels = FALSE,
#     nclass = NULL, warn.unused = TRUE, ...)
```

```
# Create a histogram
set.seed(100)  # Fix the random numbers generated
data <- rnorm(100)  # Generate some random data
hist(data, breaks = "Sturges", col = "skyblue", main = "Histogram", xlab = "Value", ylab = "Frequ
```



2.2.3 Implementing boxplot() and barplot() to visualize data comparisons and distributions

The **boxplot()** and **barplot()** functions in R are commonly used for visualizing data comparisons and distributions.

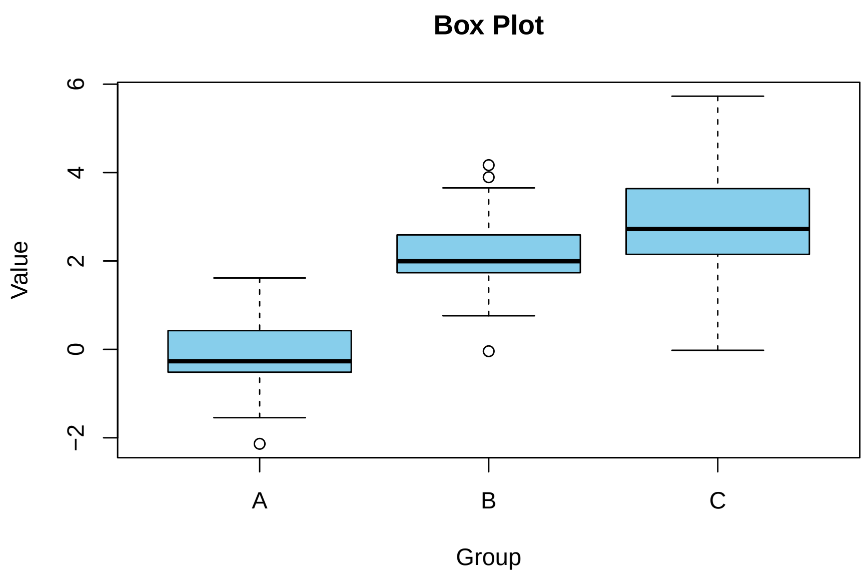
1. boxplot()

The common syntax of the function is given below.

```
#boxplot(x, data = NULL, col = "skyblue", main = NULL, xlab = NULL, ylab = #NULL)
```

Example

```
# Create a box plot
data <- data.frame(
  Group = rep(c("A", "B", "C"), each = 50),
  Value = c(rnorm(50), rnorm(50, mean = 2), rnorm(50, mean = 3))
)
boxplot(Value ~ Group, data = data, col = "skyblue", main = "Box Plot", xlab = "Group"
```



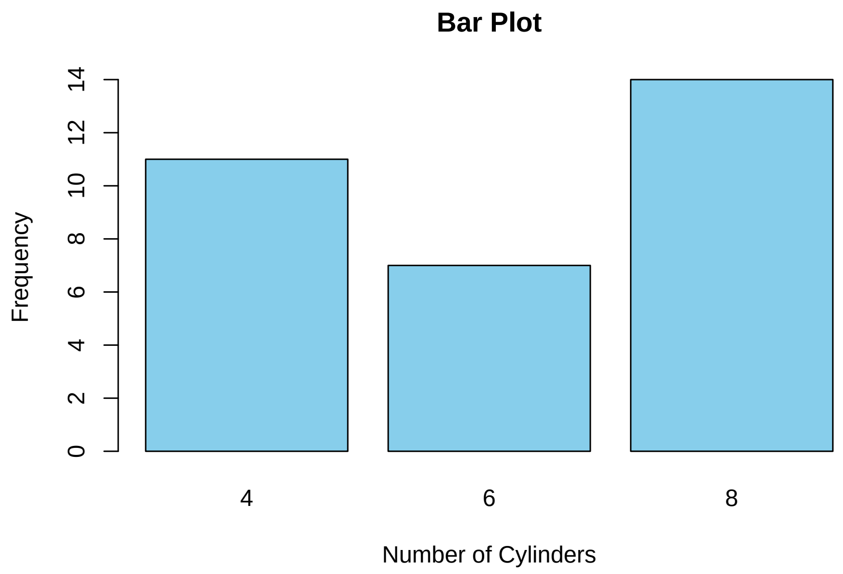
2.barplot() The syntax used for **barplot()** function is given below.

```
#barplot(x, names.arg = NULL, col = "skyblue", main = NULL, xlab = NULL, #ylab = NULL)
```

Example

```
# Create a bar plot
data <- table(mtcars$cyl)
```

```r
barplot(data, col = "skyblue", main = "Bar Plot", xlab = "Number of Cylinders", ylab = "Frequency
```

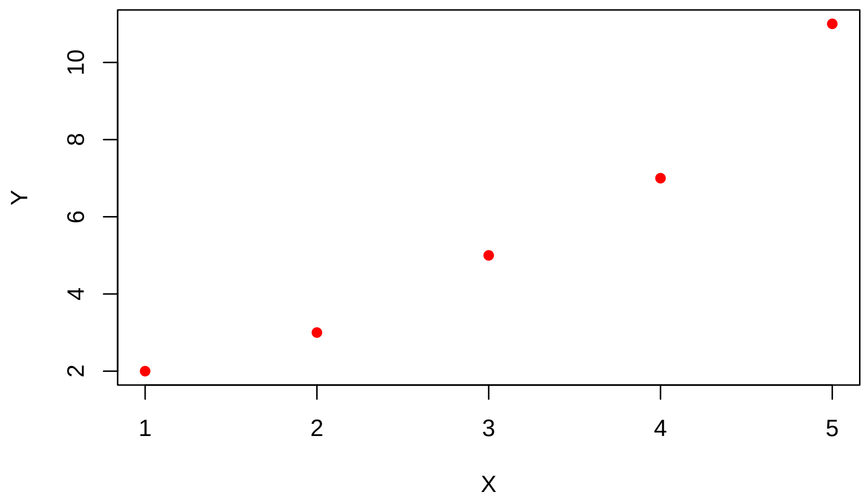


2.3 Customizing Plots

2.3.1 Basic customization options including colors, main titles, axis labels, and plot dimensions

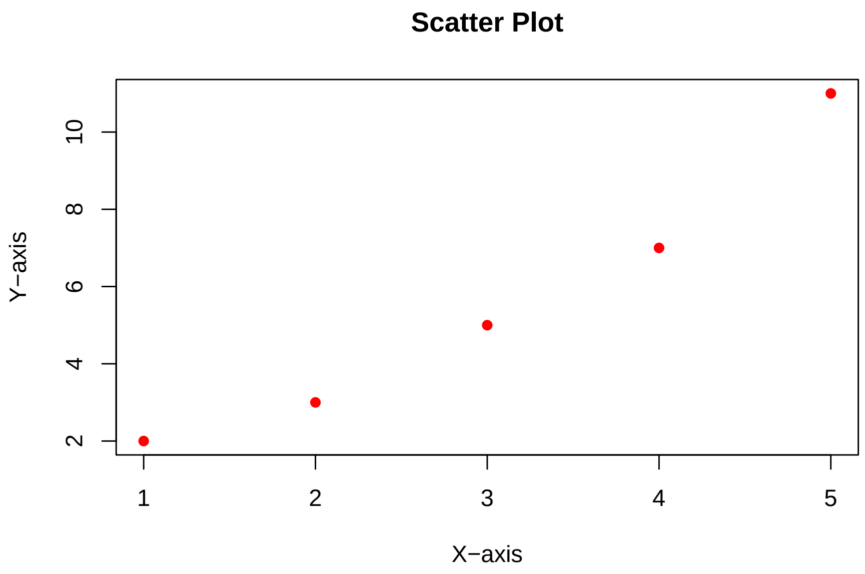
1. Changing Colors

```r
# Create a scatter plot with custom colors
x <- c(1, 2, 3, 4, 5)
y <- c(2, 3, 5, 7, 11)
plot(x, y, type = "p", col = "red", pch = 16, main = "", xlab = "X", ylab = "Y")
```



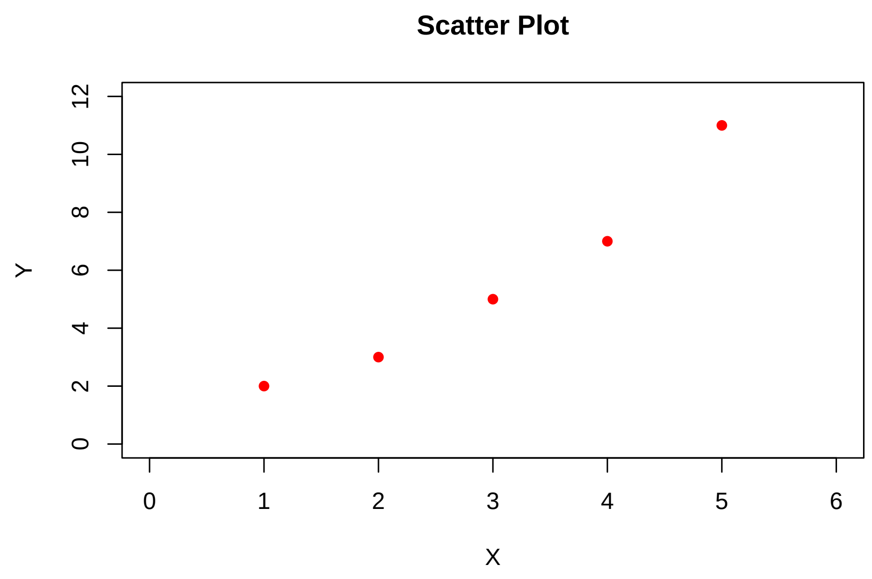
2. Adding Titles and Axis Labels

```r
# Create a scatter plot with titles and axis labels
x <- c(1, 2, 3, 4, 5)
y <- c(2, 3, 5, 7, 11)
plot(x, y, type = "p", col = "red", pch = 16, main = "Scatter Plot", xlab = "X-axis", 
```



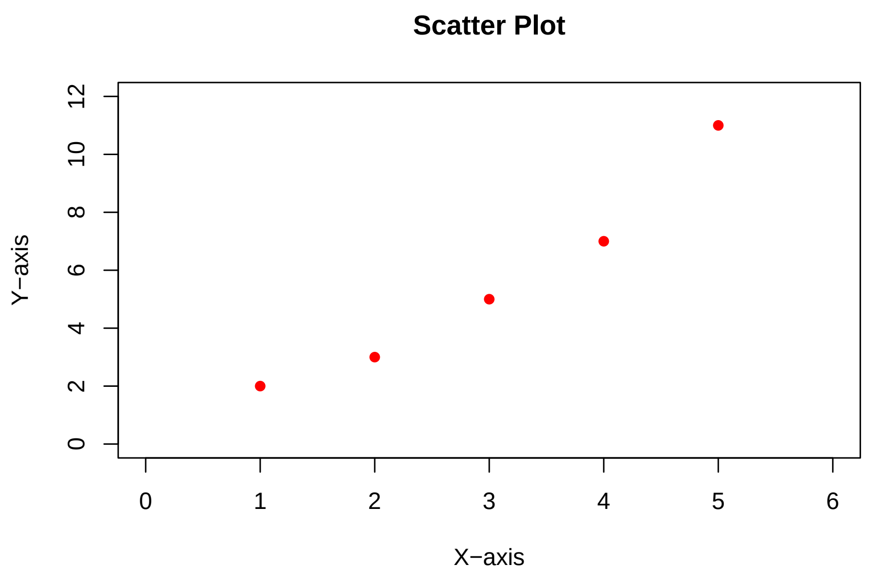
3. Changing Plot Dimensions

```
# Create a scatter plot with custom dimensions
x <- c(1, 2, 3, 4, 5)
y <- c(2, 3, 5, 7, 11)
plot(x, y, type = "p", col = "red", pch = 16, main = "Scatter Plot", xlab = "X", ylab = "Y", xlim
```



4. Combining Customization Options

```
# Create a scatter plot with custom colors, titles, axis labels, and dimensions
x <- c(1, 2, 3, 4, 5)
y <- c(2, 3, 5, 7, 11)
plot(x, y, type = "p", col = "red", pch = 16, main = "Scatter Plot", xlab = "X-axis", y
```



You can further customize your plots by adjusting parameters such as **pch** (for points), **lwd** (for lines), **lty** (for line type), **cex** (for point size), and more.

2.3.2 Tips on enhancing plot readability and aesthetic appeal

In order to have a plot with enhanced readability and have an aesthetic appeal, you need to use some of these tips:

- **Choose appropriate colors:** Use colors that complement each other and make it easy to distinguish different elements of the plot.Avoid using colors that are too bright or too similar to each other.
- **Use appropriate font sizes and styles:** Use larger font sizes for titles and axis labels to make them more prominent.
- **Add grid lines:** Adding grid lines can make it easier to read the plot and interpret the data.
- **Use appropriate plot types:** Choose the appropriate plot type for your data. For example, use a scatter plot for continuous data and a bar plot for categorical data.
- **Use consistent and intuitive labeling:** Make sure that the labels on your plot are consistent and intuitive, making it easy for readers to understand the information presented.
- **Use legends for clarity:** Use legends to explain the meaning of different colors or symbols used in the plot.

You can also add more improvements to your plot to make it look more nice and appealing.

2.4 Practical Examples: Exercises

Exercise 6

Create a dataframe of marks of 1000 students (male and female) with their respective status i.e pass if marks >= 40, fail otherwise (set a seed to 100). Additionally, let the student marks be graded using factor method in the following format:

A: 70 marks and above

B: 60-69

C: 50-59

D: 40-49

E: Below 40 marks

Create a dataframe called students with the vectors; id, gender, marks, grade and status where status is a column which shows whether the student passed or failed.You will use this dataframe to answer exercise 7,8 and 9.

```
# Your code
```

Exercise 7

Plot a box plot to show the distribution of marks by gender.Give it a title, ‘Distribution of Student Marks’. The x-axis should be labelled as ‘Gender’ while the y-axis should be labelled as ‘Marks’. Apply blue color to male and green color to female. Set cex.main and cex.lab to 1.2.

```
#Your code
```

Exercise 8

Plot a histogram of the marks where the bins should be equivalent to the classes of grading. Which grade had the highest number of students? How were the marks distributed in the class? Fill the histogram with red. Add a customized title and label the axes.

```
#Your code
```

Exercise 9

Use a well customized barpot to show the mean mark of students by gender. Which gender had the highest average score?

```
#Your code
```

Chapter 3

DATA STRUCTURES RELEVANT TO VISUALIZATION

3.1 Introduction

The two main objectives of this chapter is to understand various data structures in R and learn how to manipulate and prepare data for effective visualization.

Understanding data structures is crucial for creating effective visualizations in R. Properly organizing and structuring your data allows you to generate clear and insightful visual outputs. In this chapter, we will discuss the importance of data structure understanding in achieving effective visualizations.

3.1.1 Importance of Data Structure Understanding

- **Data Compatibility:** Different types of visualizations require different data structures. Understanding how to structure your data properly ensures compatibility with the visualization techniques you want to use.
- **Efficient Data Handling:** Well-structured data is easier to manipulate and process, leading to more efficient visualization workflows.
- **Insightful Visual Outputs:** Properly structured data enables you to generate visual outputs that effectively communicate your insights and findings.

The most commonly used data structures for visualization are vectors, matrices, data frames and lists.

3.2 Vectors, Data Frames, and Matrices

3.2.1 Definitions

Vectors are one-dimensional arrays that can hold numeric, character, or logical values. They are suitable for creating simple plots like scatter plots and line graphs. For example, a vector of numeric values can be used to represent data on the x or y-axis of a plot.

A **data frame** is a two-dimensional data structure where columns can be of different types. Data frames are highly versatile and commonly used for data analysis and visualization. They are suitable for creating a wide range of visualizations, including bar plots, histograms, and box plots. Each column in a data frame can represent a different variable, making it easy to visualize relationships between variables.

Matrices are two-dimensional arrays that contain elements of the same atomic types. Matrices are useful for creating visualizations such as heatmaps and surface plots. They are suitable for representing data that has two dimensions, such as images or spatial data.

Lists are ordered collections of objects, which can be of different types. Lists can also be nested, allowing for complex data structures. They provide flexibility in organizing and storing data for complex visualizations. They are suitable for creating custom plots or combining multiple plots into a single visualization.

Factors are used to represent categorical data in R. They are stored as integers and have labels associated with them. They are useful for creating plots that involve categorical data, such as bar plots and pie charts. They allow for easy grouping and aggregation of data based on categorical variables.

Example

```r
# Example data
set.seed(123)
numeric_vector <- rnorm(10)
character_vector <- letters[1:10]
logical_vector <- sample(c(TRUE, FALSE), 10, replace = TRUE)

# Create a list
my_list <- list(numeric_vector, character_vector, logical_vector)

# Create a matrix
my_matrix <- matrix(1:12, nrow = 4)

# Create a data frame
my_data <- data.frame(
  ID = 1:4,
  Name = c("John", "Mary", "David", "Emma"),
  Age = c(25, 30, 35, 40)
)

# Create a factor
my_factor <- factor(c("A", "B", "A", "C"))

# Print the structures
print(my_list)
```

```
## [[1]]
##  [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
##  [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197
##
## [[2]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
##
## [[3]]
##  [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE
```

```
print(my_matrix)
```

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

```
print(my_data)
```

```
##   ID  Name Age
## 1  1  John  25
## 2  2  Mary  30
## 3  3 David  35
## 4  4  Emma  40
```

```
print(my_factor)
```

```
## [1] A B A C
## Levels: A B C
```

3.2.2 Tools and Functions

Understanding various tools and functions in R is essential for manipulating data structures effectively. Here are some commonly used tools and functions:

1. str(): Displays the internal structure of R objects. For example if the object is a data frame, it shows the type of each vector in the dataframe

2. class(): Returns the class of the R object.

3. length(): Returns the number of elements in the R object. If the object is a data frame, it returns the number of columns in the data frame.

4. dim(): Returns the dimensions of an R object (for matrices and arrays).

5. attributes(): Returns the attributes of an R object.

6. as.factor(): Converts a vector to a factor.

7. data.frame(): Creates a data frame from vectors or other R objects.

8. ncol(): Returns the number of columns of a dataframe or matrix.

9. nrow(): Returns the number of rows of a data frame.

3.2.3 Step-by-Step Tutorials

3.2.3.1 Converting a Vector to a Data Frame

```
# Example vector
student_id <- 1:5
student_name <- c("John", "Alice", "Bob", "Emily", "Tom")
student_score <- c(80, 75, 90, 85, 88)

# Combine vectors into a data frame
student_data <- data.frame(ID = student_id, Name = student_name, Score = student_score

# View the data frame
print(student_data)
```

```
##   ID  Name Score
## 1  1  John    80
## 2  2 Alice    75
## 3  3   Bob    90
## 4  4 Emily    85
## 5  5   Tom    88
```

3.2.3.2 Creating and Modifying Lists for Nested Data Structures

```
# Example list
student1 <- list(Name = "John", Age = 25, Grades = c(Math = 80, Science = 85))
student2 <- list(Name = "Alice", Age = 23, Grades = c(Math = 75, Science = 78))

# Create a nested list
classroom <- list(student1, student2)

# View the nested list
print(classroom)
```

```
## [[1]]
## [[1]]$Name
## [1] "John"
##
## [[1]]$Age
## [1] 25
##
## [[1]]$Grades
##    Math Science
##      80      85
##
##
## [[2]]
## [[2]]$Name
## [1] "Alice"
##
## [[2]]$Age
## [1] 23
##
## [[2]]$Grades
##    Math Science
##      75      78
```

```
# Accessing elements of the nested list
print(classroom[[1]]$Name)  # Output: John
```

```
## [1] "John"
```

```
print(classroom[[2]]$Grades["Science"])  # Output: 78
```

```
## Science
##      78
```

```
# Modifying elements of the nested list
classroom[[1]]$Grades["Math"] <- 85

# View the modified nested list
print(classroom)
```

```
## [[1]]
## [[1]]$Name
## [1] "John"
##
## [[1]]$Age
## [1] 25
##
## [[1]]$Grades
##    Math Science
##      85      85
##
##
## [[2]]
## [[2]]$Name
## [1] "Alice"
##
## [[2]]$Age
## [1] 23
##
## [[2]]$Grades
##    Math Science
##      75      78
```

3.3 Practical Example : Exercise

Exercise 10

You have been provided with a snapshot of a fictional retail landscape, capturing essential attributes that drive retail operations and customer interactions. It includes key details such as Transaction ID, Date, Customer ID, Gender, Age, Product Category, Quantity, Price per Unit, and Total Amount. The dataset can be downloaded from Kaggle via the link https://www.kaggle.com/datasets/mohammadtalib786/retail-sales-dataset. Import the data into R and answer the following questions:

i). Using a well customized line plot, what trend do you see in the sales (Total Amount)?

ii). Using an appropriate plot covered so far, show how the total amount vary by the product category. Use a well customized plot to make it more appealing and readable.

Chapter 4

CUSTOMIZING GRAPHS IN BASE R

4.1 Introduction

The main objectives of this chapter are:

- Learn to customize and enhance graphs using R's base graphic parameters.
- Understand how to apply visual enhancements to convey information more effectively.

4.2 Graph Customization Basics

Customizing plots in R allows you to enhance the appearance and readability of the visualizations. The **par()** function in R is used to set or query graphical parameters.

i). Graphical Parameters (par())

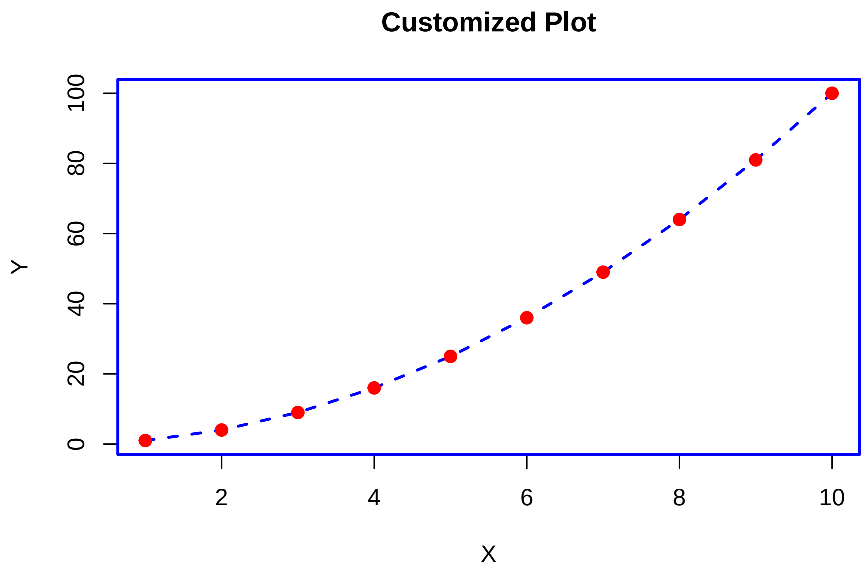
- **pch:** Sets the plotting symbol (or point character) used in plotting functions like **plot()**.
- **lty:** Sets the line type.
- **lwd:** Sets the line width.
- **col:** Sets the color of lines and points.
- **bg:** Sets the background color.
- **cex:** Sets the character size.
- **mar:** Sets the margins of the plot.

ii). Adding Custom Colors, Line Types, Point Shapes, and Plot Characters

- **Custom Colors:** You can specify custom colors using color names, hexadecimal codes, or color numbers.
- **Line Types:** Different line types are available, such as solid, dashed, dotted, etc.
- **Point Shapes:** Various point shapes are available, including circles, squares, triangles, etc.
- **Plot Characters:** Different plot characters, such as dots, crosses, stars, etc., can be used.

Example:

```r
# Sample data
x <- 1:10
y <- x^2

# Plot with customized graphical parameters
par(
  pch = 19,     # Use solid circles for points
  lty = 2,      # Use dashed line
  lwd = 2,      # Set line width to 2
  col = "blue" # Set line and point color to blue
)

plot(x, y, type = "l", main = "Customized Plot", xlab = "X", ylab = "Y")
points(x, y, col = "red") # Add points with red color
```



4.3 Enhancing Plot Aesthetics

Improving the aesthetics of plots is essential for effectively communicating your data. The following are some techniques for enhancing plot aesthetics in R:

1. Setting Axis Options:

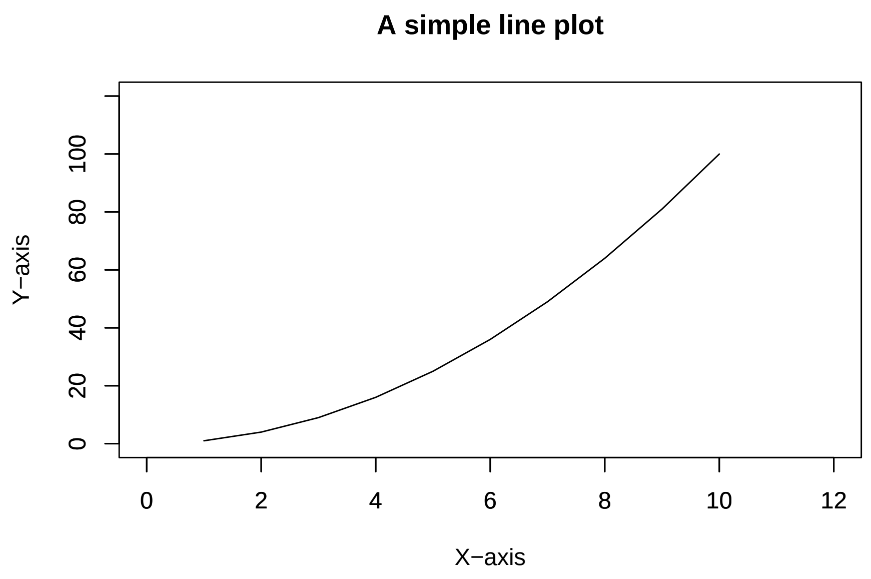
Axis Labels: Use **xlab** and **ylab** parameters to set labels for x and y axes, respectively.

Tick Marks: Use **axes = FALSE** to remove axis tick marks and add them back using axis() function.

Axis Limits: Use **xlim** and **ylim** parameters to set limits for the x and y axes, respectively.

Example:

```
# Sample data
x <- 1:10
y <- x^2

# Plot with customized axis options
plot(x, y, type = "l", main = "A simple line plot", xlab = "X-axis", ylab = "Y-axis", xlim = c(0,
axis(side = 1, at = seq(0, 12, by = 2)) # Add x-axis tick marks
axis(side = 2, at = seq(0, 120, by = 20)) # Add y-axis tick marks
```



4.4 Advanced Customization Techniques

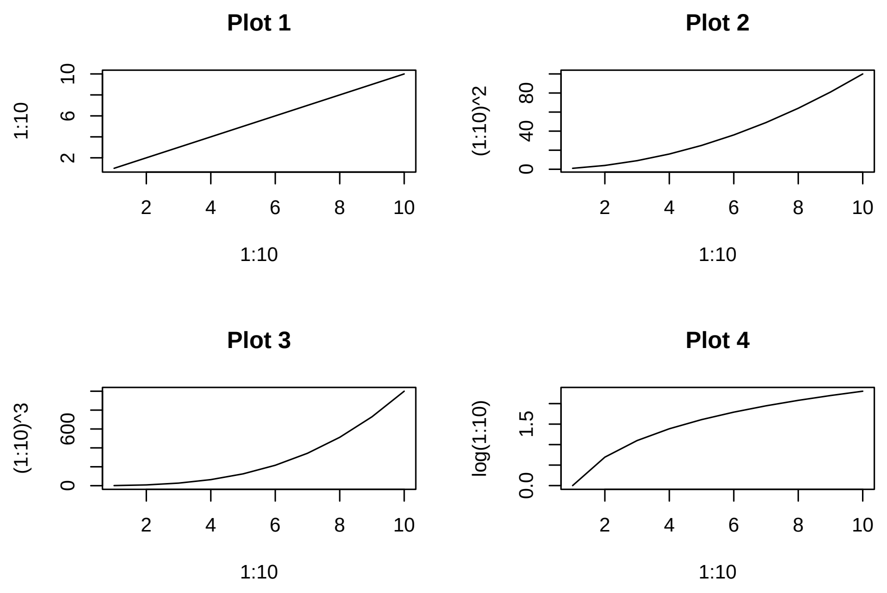
Enhancing plots in R involves more than just changing colors and labels. Here are some advanced customization techniques to make your plots more informative and visually appealing:

i). Multiple Plot Windows (mfrow and mfcol):

mfrow: Divides the plotting area into a matrix of rows and columns. Plots are filled row-wise.

mfcol: Divides the plotting area into a matrix of rows and columns. Plots are filled column-wise.

```r
# Example using mfrow
par(mfrow = c(2, 2)) # Create a 2x2 grid for plots
plot(1:10, 1:10, main = "Plot 1", type = "l")
plot(1:10, (1:10)^2, main = "Plot 2", type = "l")
plot(1:10, (1:10)^3, main = "Plot 3", type = "l")
plot(1:10, log(1:10), main = "Plot 4", type = "l")
```



ii). Customizing Plots with Grid Lines, Background Color, and Themes:

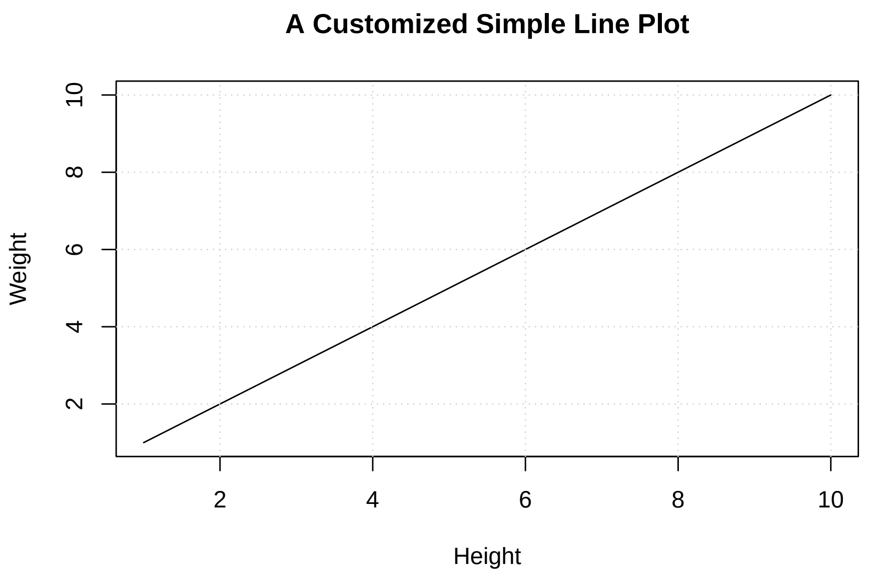
Grid Lines: Use **grid()** function to add grid lines to the plot.

Background Color: Use **par(bg = “color”)** to set background color for the plot.

Themes: Use themes from packages like ggplot2 to change the overall appearance of the plot. This will be covered under ggplot2 package chapter.

Example with Grid Lines and Background Color:

```
# Example with grid lines and background color
plot(1:10, 1:10, main = "A Customized Simple Line Plot", type = "l", xlab='Height',ylab = 'Weight
grid() # Add grid lines
```



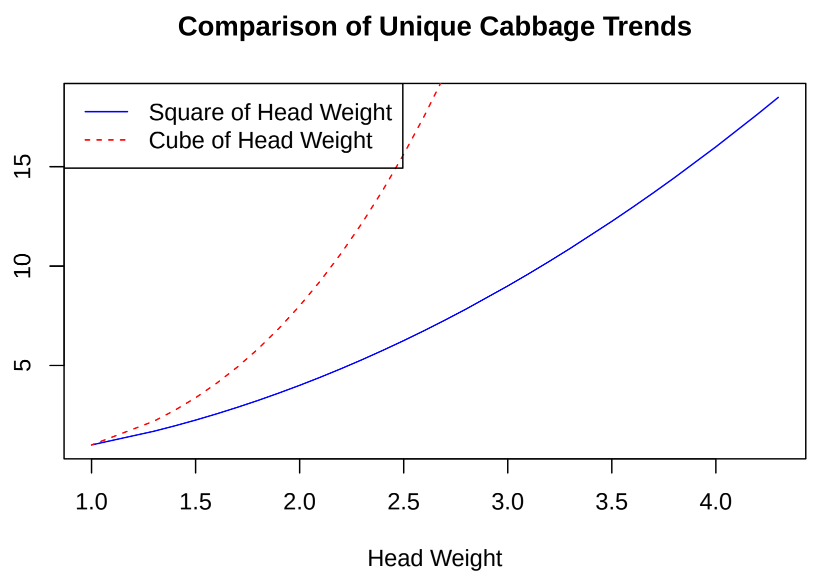
```
par(bg = "lightblue") # Set background color
```

4.5 Practical Examples: Exercise

Exercise 11

Read the **cabbages** dataset from **MASS** package. Remove the duplicated values of **"HeadWt"** and arrange the dataframe in descending order by **"HeadWt"** column. Plot a line plot of **"HeadWt"** against its square and apply blue color with the first line type. Add a second line plot of the square of cube of **"HeadWt"** , apply the red color and the line type should be the second one. Add a legend to the plot and position it at the top - left position with the names "Square of Head Weight" and "Cube of Head Weight" respectively. Maintain the same line types and colors that were used to plot. Add a title, 'Comparison of Cabbage Trends'. Label the x-axis as 'Head Weight' and y-axis should be blank

```
# Your code

# Solution
library(dplyr)
library(MASS)

cabbages <- cabbages %>%
  distinct(HeadWt, .keep_all = TRUE) %>%
  arrange(desc(HeadWt),FALSE)

plot(cabbages$HeadWt,(cabbages$HeadWt)^2,type = 'l', col='blue',
     lty=1,main = "Comparison of Unique Cabbage Trends", xlab = "Head Weight", ylab = '
lines(cabbages$HeadWt,(cabbages$HeadWt)^3,type = 'l', col='red', lty=2)
legend("topleft", legend = c("Square of Head Weight", "Cube of Head Weight "), col = c
```



Exercise 12 : Multiple Small Plots in One Graphic to Compare Different Data Sets

Create a vector **x** of a sequence of numbers from 1 to 100. Create another vector **y1** which is the square of **x**. Create another vector **y2** which is the double of **x**. Lastly, create the last vector **y3** which is equivalent to **x** raised to the power of 1.5. Create three multiple small line plots and organize them in one row with three columns. All plots will use the same independent variable in the x-axis, i.e **x**.

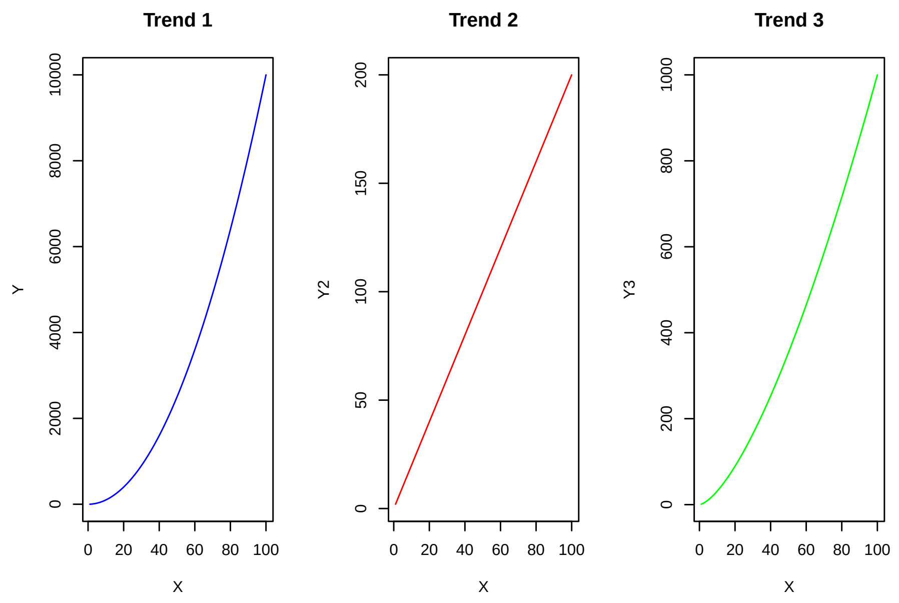
i) Plot x against y1. Give it the title,'Trend 1', x-axis should be labelled as

‘X’ while y-axis=‘Y’ and color is blue.

ii) Plot x against y2. Give it the title,‘Trend 2’, x-axis should be labelled as ‘X’ while y-axis=‘Y2’,col is red.

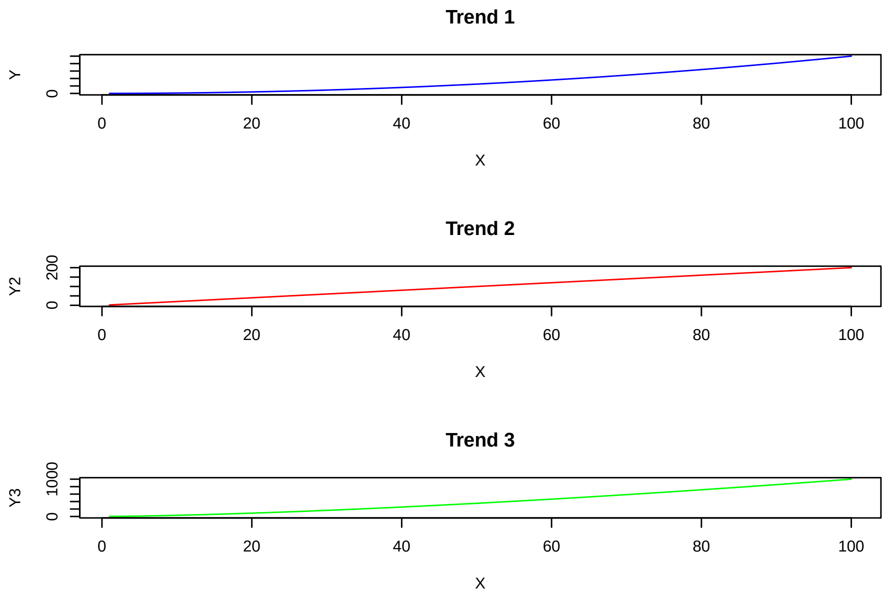
iii) Plot x against y3. Give it the title,‘Trend 3’, x-axis should be labelled as ‘X’ while y-axis=‘Y3’,col is green.

```
# Your  data

#Solution
#Create data
x <- 1:100
y1 <- x^2
y2 <- 2 * x
y3 <- x^1.5

# Create multiple small plots
par(mfrow = c(1, 3)) # 1 row, 3 columns

# Plot 1
plot(x, y1, type = "l", col = "blue", main = "Trend 1", xlab = "X", ylab = "Y")

# Plot 2
plot(x, y2, type = "l", col = "red", main = "Trend 2", xlab = "X", ylab = "Y2")

# Plot 3
plot(x, y3, type = "l", col = "green", main = "Trend 3", xlab = "X", ylab = "Y3")
```



iv) Repeat the above similar plots but arranged in three rows and one column using **mfcol()**.

```
# Your  data

#Solution
#Create data
x <- 1:100
y1 <- x^2
y2 <- 2 * x
y3 <- x^1.5

# Create multiple small plots
par(mfcol = c(3, 1)) # 3 rows, 1 column

# Plot 1
plot(x, y1, type = "l", col = "blue", main = "Trend 1", xlab = "X", ylab = "Y")

# Plot 2
plot(x, y2, type = "l", col = "red", main = "Trend 2", xlab = "X", ylab = "Y2")

# Plot 3
plot(x, y3, type = "l", col = "green", main = "Trend 3", xlab = "X", ylab = "Y3")
```


Trend 1
Y
0
0
20
40
60
80
100
X
Trend 2
Y2
0
200
X
Trend 3
Y3
0
1000
X

Chapter 5

ADVANCED BASE PLOTTING TECHNIQUES

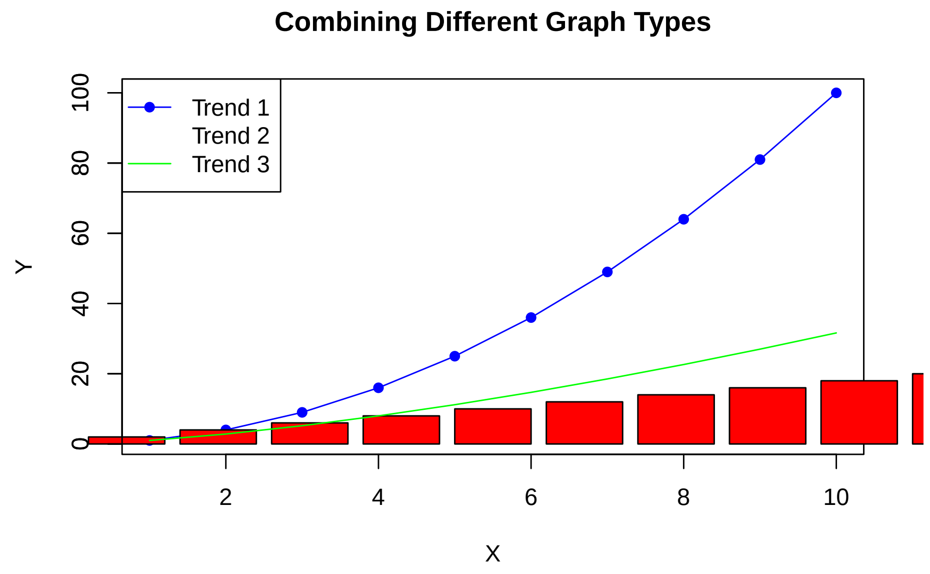
5.1 Combining Different Graph Types

We can overlay different types of plots, such as combining bar charts with line graphs, in R. Here's how this can be done along with customizing axes and scales:

5.1.1 Overlay different types of plots

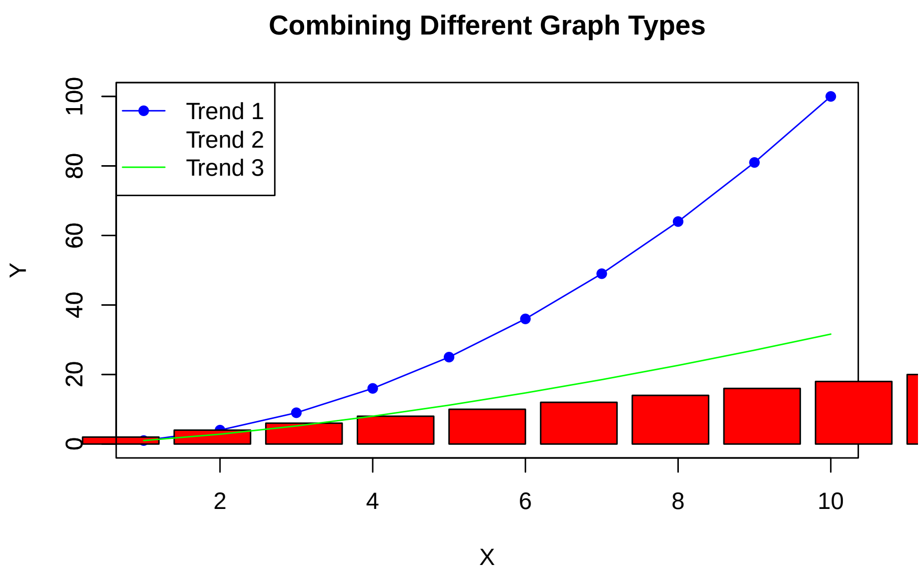
```r
# Sample data
x <- 1:10
y1 <- x^2
y2 <- 2 * x
y3 <- x^1.5

# Create a line plot
plot(x, y1, type = "l", col = "blue", main = "Combining Different Graph Types", xlab = "X", ylab

# Add points to the line plot
points(x, y1, col = "blue", pch = 16)

# Add a bar plot
barplot(y2, col = "red", add = TRUE)

# Add a second line plot
```

```r
lines(x, y3, col = "green")

# Add a legend
legend("topleft", legend = c("Trend 1", "Trend 2", "Trend 3"), col = c("blue", "red", '
```



5.1.2 Axes and scales customization to accomodate combined plots

```r
# Set custom axis limits
plot(x, y1, type = "l", col = "blue", main = "Combining Different Graph Types", xlab =

# Add points to the line plot
points(x, y1, col = "blue", pch = 16)

# Add a bar plot
barplot(y2, col = "red", add = TRUE, ylim = c(0, max(y1, y2, y3)))

# Add a second line plot
lines(x, y3, col = "green")

# Add a legend
legend("topleft", legend = c("Trend 1", "Trend 2", "Trend 3"), col = c("blue", "red", '
```

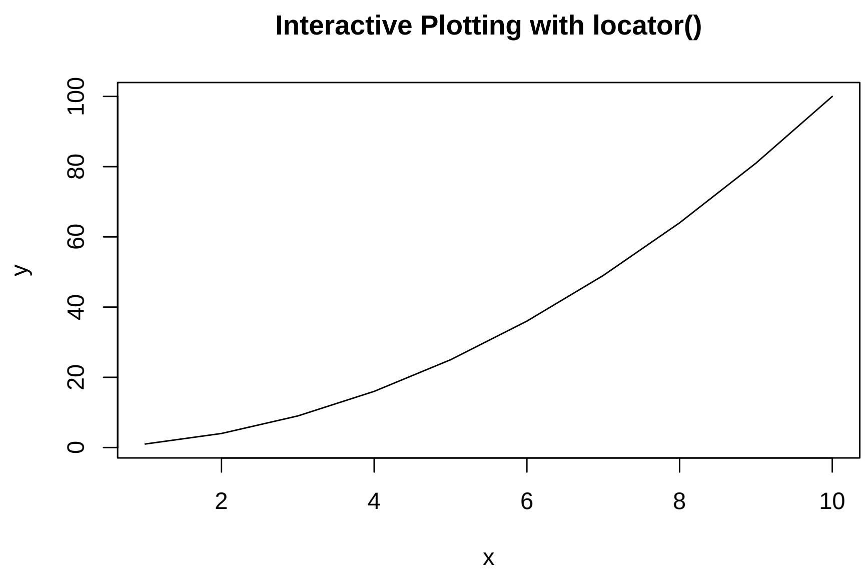


5.2 Interactive Graphics in Base R

5.2.1 Introduction to using the locator() function for interactive plotting

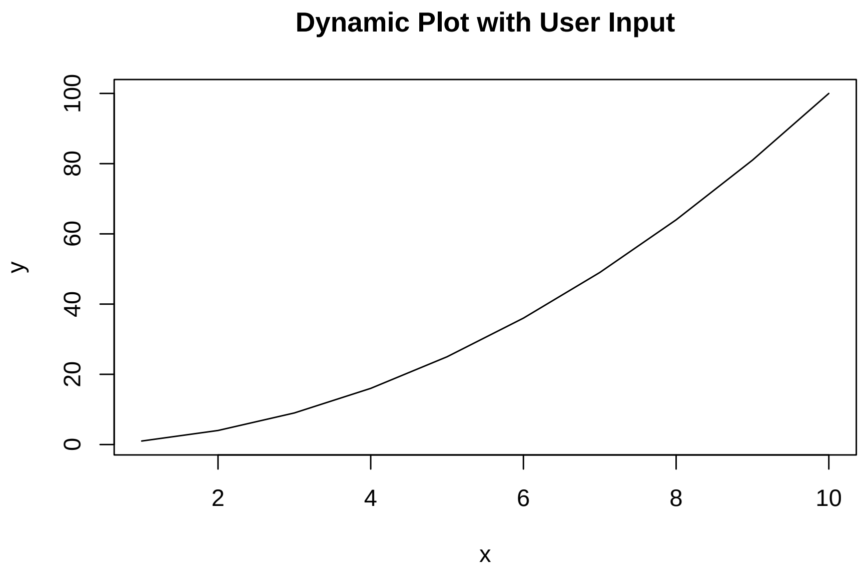
Base R provides basic functionality for creating interactive plots using the locator() function. Here's how you can use locator() for interactive plotting and create dynamic plots that respond to user input or updates in data:

```r
# Sample data
x <- 1:10
y <- x^2

# Plot data
plot(x, y, type = "l", main = "Interactive Plotting with locator()")

# Add points interactively
points(locator(5), col = "red", pch = 16)
```



5.3.3 Creating dynamic plots that respond to user input

```r
# Dynamic plot that responds to user input
plot_dynamic <- function() {
  # Sample data
  x <- 1:10
  y <- x^2

  # Plot data
  plot(x, y, type = "l", main = "Dynamic Plot with User Input")

  # Add points interactively
  points(locator(5), col = "red", pch = 16)
}

# Call the function to create dynamic plot
plot_dynamic()
```


Dynamic Plot with User Input
y
0
20
40
60
80
100
2
4
6
8
10
x

Chapter 6

PLOTTING TIME SERIES DATA

6.1 Introduction

The plotting of time series object is most likely one of the steps of the analysis of time-series data. The two main objectives of this chapter are:

- Understand specific considerations for plotting time series data in R.
- Learn to effectively visualize time-dependent data using R's base graphics.

6.1.1 Importance of time series analysis in various fields like finance, meteorology, and epidemiology

Time series analysis plays a crucial role in various fields such as finance, meteorology, epidemiology, and many others. Understanding time series data and effectively visualizing it are essential for extracting meaningful insights. In this chapter, we will explore the importance of time series analysis and the challenges associated with visualizing time series data.

Time series analysis is vital in various fields including:

- **Finance:** Analyzing stock prices, market trends, and forecasting.
- **Meteorology:** Studying weather patterns, climate change, and forecasting.
- **Epidemiology:** Tracking disease outbreaks, analyzing trends, and forecasting.

6.1.2 Basic concepts related to time series data and its visualization challenges.

- **Time Series Data:** Time series data consists of observations or measurements taken at different points in time.
- **Temporal Patterns:** Time series data often exhibits various temporal patterns such as trend, seasonality, and cycles.
- **Visualization Challenges:** Visualizing time series data poses challenges due to its temporal nature and the need to effectively represent temporal patterns.

6.2 Time Series Data Basics

Time series data in R is commonly represented using the **ts** class. The time series object must have a **Date** or **POSIXct/lt** column and at least one numeric column.

6.2.1 Introduction to time series objects in R (ts class).

- The **ts** class in R is a basic time series object.
- It is suitable for regularly spaced time series data.
- Time series objects are created using the **ts()** function.

Example: Creating a time series object

```r
# Sample time series data
data <- c(10, 15, 20, 25, 30)
time_series <- ts(data)

# View the time series object
print(time_series)
```

```
## Time Series:
## Start = 1
## End = 5
## Frequency = 1
## [1] 10 15 20 25 30
```

6.2.2 How to convert standard date formats into time series objects

R provides several functions to convert standard date formats into time series objects, such as **as.ts()** and **ts()**.

Example: Converting Standard Date Formats into Time Series Objects

```
# Sample data with dates
dates <- as.Date(c("2022-01-01", "2022-02-01", "2022-03-01", "2022-04-01", "2022-05-01"))
data <- c(10, 15, 20, 25, 30)

# Create a time series object
time_series <- ts(data, start = c(as.POSIXlt(dates[1])$year + 1900, as.POSIXlt(dates[1])$mon + 1)

# View the time series object
print(time_series)
```

```
##      Jan Feb Mar Apr May
## 2022  10  15  20  25  30
```

In this example, **start** indicates the start of the time series, and **frequency** indicates the number of observations per unit of time. Here, **frequency = 12** indicates monthly data. These are the basics of working with time series data objects in R.

6.3 Visualizing Time Series

Let's load and plot the **USgas** series, a **ts** object which is attached in the TSstudio package:

```
library(TSstudio)
```

```
## Warning: package 'TSstudio' was built under R version 4.3.3
```

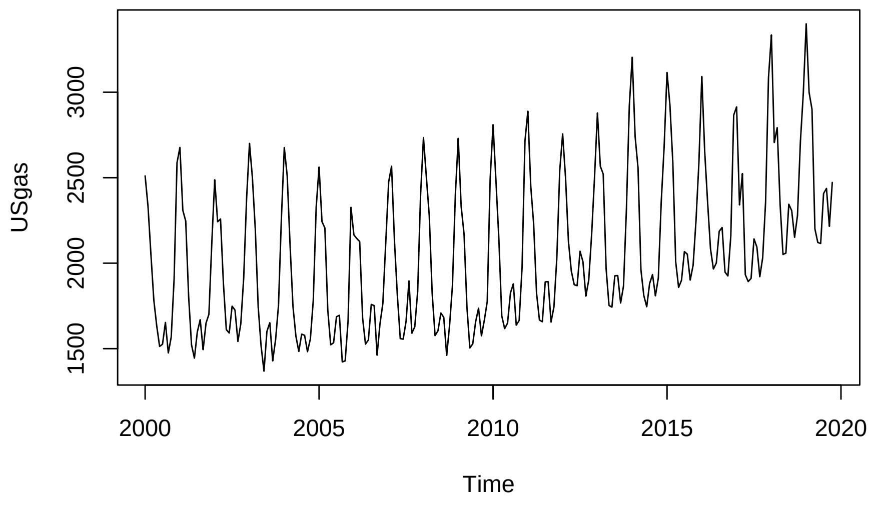
```
data(USgas)

ts_info(USgas)
```

```
##  The USgas series is a ts object with 1 variable and 238 observations
##  Frequency: 12
##  Start time: 2000 1
##  End time: 2019 10
```

6.3.1 Creating line plots for time series data

Let us plot the **USgas** series using the base R **plot()** function.

```
plot(USgas)
```



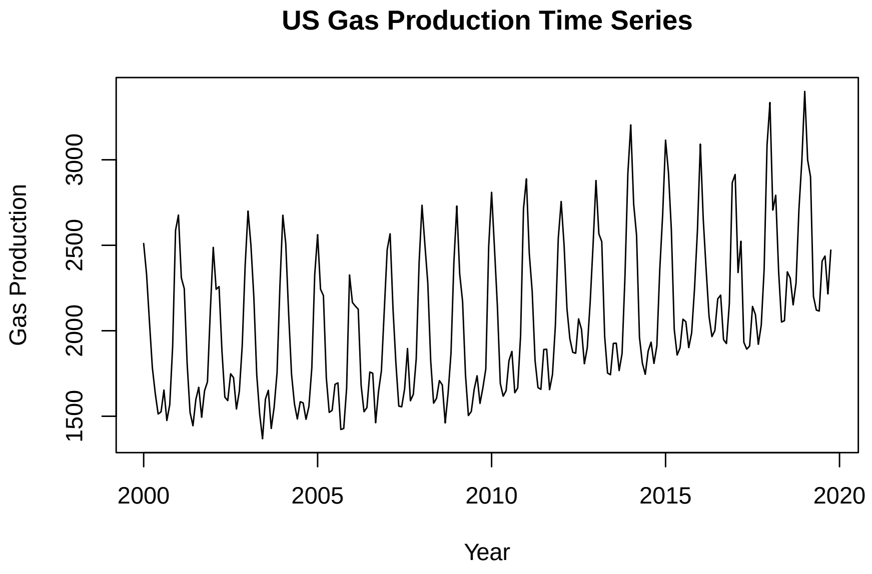
6.3.2 Techniques for highlighting trends, seasonality, and anomalies

- **Trends:** Use **lines()** function to overlay a trend line on the plot.
- **Seasonality:** Use **seasonplot()** function from the forecast package to visualize seasonal patterns.
- **Anomalies:** Use **points()** function to mark anomalies on the plot.

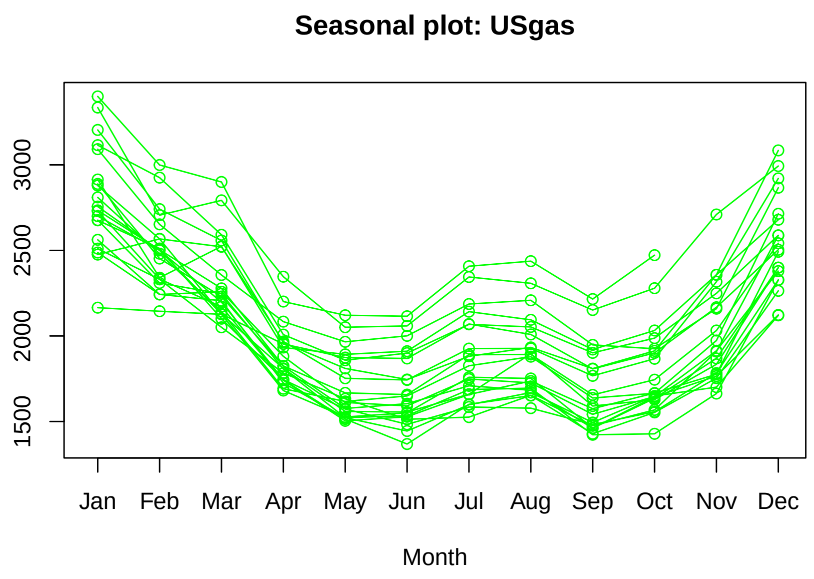
Example: Let's visualize the **USgas** time series data and highlight trends, seasonality, and anomalies:

```
# Install and load the necessary packages
#install.packages("forecast")
library(forecast)

# Load the USgas time series object
data("USgas")

# Convert the ts object into a data frame
USgas_df <- data.frame(year = time(USgas), production = as.numeric(USgas))

# Apply loess function
trend <- loess(production ~ year, data = USgas_df)
```

```
# Extract the trend values
trend_values <- predict(trend)

# Plot the USgas time series
plot(USgas, main = "US Gas Production Time Series", xlab = "Year", ylab = "Gas Production")
```



```
# Add trend line
#lines(time(USgas), trend_values, col = "blue")

# Visualize seasonal pattern
seasonplot(USgas, col = "green")

# Identify anomalies
anomalies <- which(USgas > mean(USgas) + 2 * sd(USgas) | USgas < mean(USgas) - 2 * sd(USgas))
points(time(USgas)[anomalies], USgas[anomalies], col = "red", pch = 16)
```

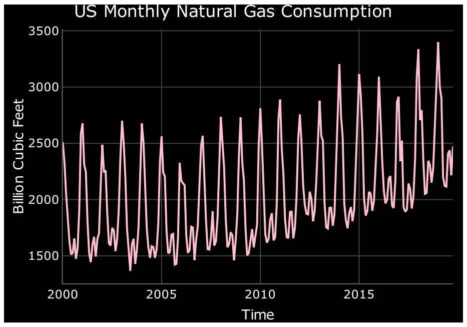


6.4 Advanced Time Series Visualization

6.4.1 Advance customization using ts_plot() function from plotly package

The **ts_plot** is a wraper of the **plotly** package plotting functions for time series objects, therefore, the output of the **ts_plot** is a **plotly** object. Advance customization of the **ts_plot** output can be done with plotly's layout function. For example, let's re plot the **USgas** series and customize the background to black:

```
library(plotly)

ts_plot(USgas,
        title = "US Monthly Natural Gas Consumption",
        Xtitle = "Time",
        Ytitle = "Billion Cubic Feet",
        color =  "pink",
        Xgrid = TRUE,
        Ygrid = TRUE) %>%
  layout(paper_bgcolor = "black",
         plot_bgcolor = "black",
         font = list(color = "white"),
         yaxis = list(linecolor = "#6b6b6b",
                      zerolinecolor = "#6b6b6b",
                      gridcolor= "#444444"),
         xaxis = list(linecolor = "#6b6b6b",
                      zerolinecolor = "#6b6b6b",
                      gridcolor= "#444444"))
```



Note that the **Xgrid** and **Ygrid** arguments, when set to **TRUE**, add the corresponding X and Y grid lines.

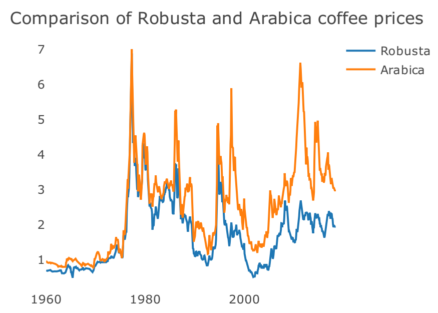
6.4.2 Plotting multiple time series on a single graph for comparative analysis

The plotting of a multiple time series object is straightforward. Let's load the **Coffee_Prices** an **mts** object that represents the monthly prices of the Robusta and Arabica coffee prices (USD per Kg.):

```
data("Coffee_Prices")

ts_info(Coffee_Prices)
```

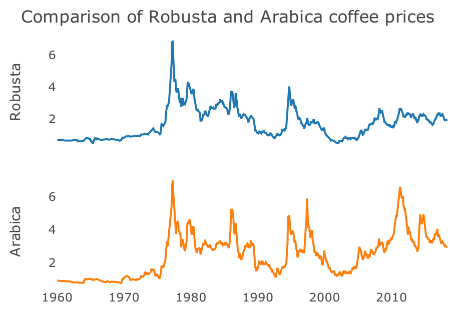
```
##  The Coffee_Prices series is a mts object with 2 variables and 701 observations
##  Frequency: 12
##  Start time: 1960 1
##  End time: 2018 5
```

```
ts_plot(Coffee_Prices,
        title = "Comparison of Robusta and Arabica coffee prices")
```



By default, the function will plot all the series in one plot. Plotting the different series on a separate plot can be done by setting the type argument to multiple:

```
ts_plot(Coffee_Prices,
        title = "Comparison of Robusta and Arabica coffee prices",
        type = "multiple")
```



Note that the **color**, **Ytitle**, and **Xtitle** arguments are not applicable when plotting multiple time series object.

6.5 Practical Examples: Exercises

Exercise 13: Stock market data

Obtain the daily closing prices for the last ten calendar years of S&P 500 Index, Dow Jones, NASDAQ100 and Russell 2000 indices from Yahoo Finance using the appropriate R package such as tidyquant or quantmod. Prepare the data and clean it by removing any missing values. Plot a time series plot of the prices of these indices in one graph and use the plot to identify any economic trends among the indices.

Exercise 14: Temperature

Obtain the temperature data from an open source such as Weather data (https://www.wunderground.com/weather/in/ahmedabad/VAAH) for Ahmedabad which is located in Western India on the banks of River Sabarmati. Obtain the average monthly temperature data from Sardar Vallabhbhai Patel International Airport Station weather station for the years 2014-2021. Decompose the time series data by splitting these components (data or the level, trend, seasonality, and noise or random components) separately into individual components. Present the decomposition of the multiplicative model of the components of the time series of the average temperature of Ahmedabad in a suitable plot. What observation did you make about the seasonality of this data?

Chapter 7

STATISTICAL GRAPHS IN R

7.1 Introduction

The two main objectives of this chapter are:

- Learn to use R's base graphics for creating statistical plots.
- Understand the application of statistical plots in real-world data analysis.

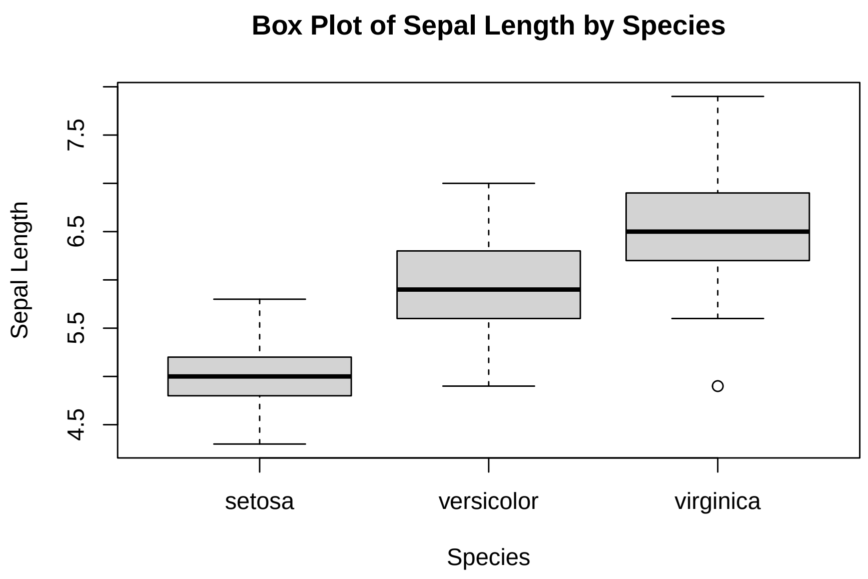
7.2 Essential Statistical Plots:

Statistical plots are essential for exploring and understanding the distribution of data. In this chapter, we will cover how to create the following essential statistical plots in R and understand the insights they provide.

7.2.1 How to create box plots, violin plots, and QQ plots.

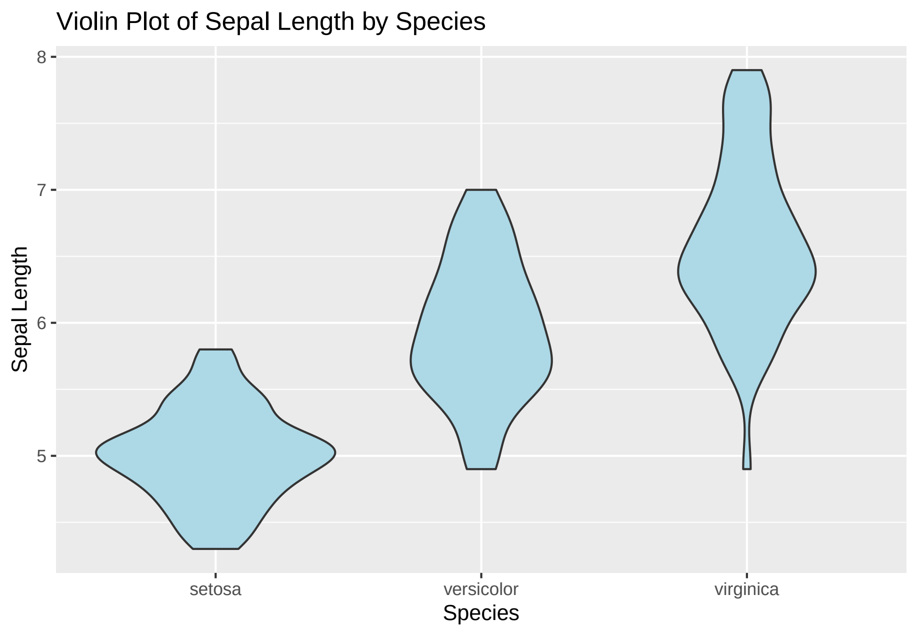
i) Box Plots:

```r
# Creating a box plot
boxplot(Sepal.Length ~ Species, data = iris,
        main = "Box Plot of Sepal Length by Species",
        xlab = "Species", ylab = "Sepal Length")
```



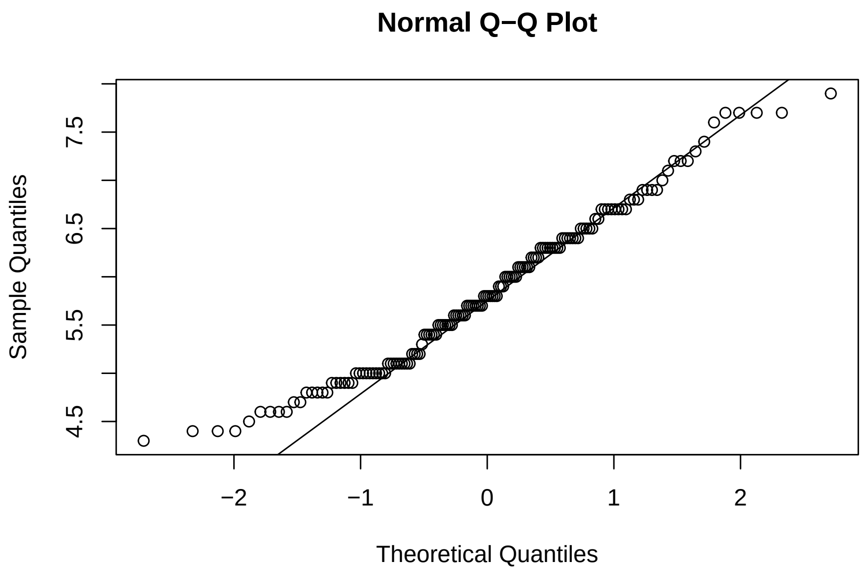
ii) Violin Plots:

```
# Creating a violin plot
library(ggplot2)
ggplot(iris, aes(x = Species, y = Sepal.Length)) +
  geom_violin(fill = "lightblue") +
  labs(title = "Violin Plot of Sepal Length by Species",
       x = "Species", y = "Sepal Length")
```



iii) QQ Plots:

```
# Creating a QQ plot
qqnorm(iris$Sepal.Length)
qqline(iris$Sepal.Length)
```



7.2.2 Understanding the data insights these plots provide.

i) Box Plots: Box plots provide a graphical summary of the distribution of a dataset. They display the median, quartiles, and potential outliers.

ii) Violin Plots: Violin plots are similar to box plots but provide a more detailed summary of the distribution. They combine a box plot with a kernel density plot.

iii) QQ Plots (Quantile-Quantile Plots): QQ plots are used to assess if a dataset follows a particular distribution. They compare the quantiles of the dataset against the quantiles of a theoretical distribution.

These statistical plots are invaluable tools for exploring and understanding the distribution of data.

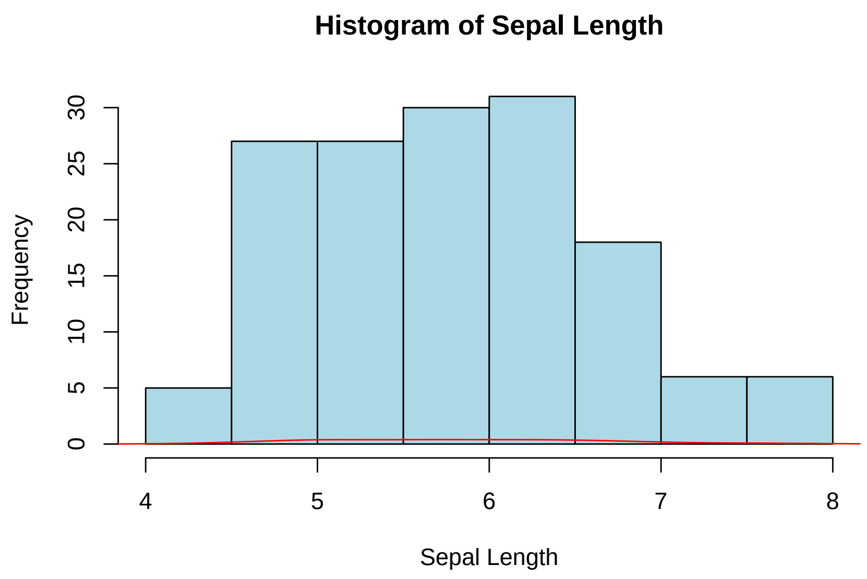
7.3 Analyzing Distributions with Histograms and Density Plots:

Histograms provide a visual representation of the distribution of numerical data. They group data into bins and display the frequency of observations in each bin.Density plots estimate the probability density function of the underlying data distribution. They provide a smooth representation of the distribution.

7.3.1 Techniques for creating and customizing histograms and density plots and best practices for displaying distribution characteristics.

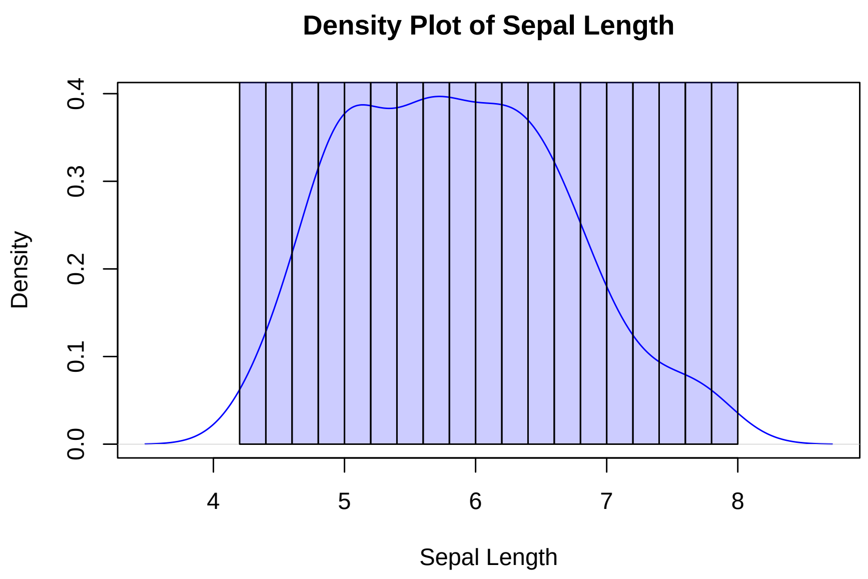
Creating and Customizing Histograms:

```r
# Creating a histogram
hist(iris$Sepal.Length,
     main = "Histogram of Sepal Length",
     xlab = "Sepal Length",
     ylab = "Frequency",
     col = "lightblue")

# Adding a density plot
lines(density(iris$Sepal.Length), col = "red")
```



Creating and Customizing Density Plots:

```r
# Creating a density plot
plot(density(iris$Sepal.Length),
     main = "Density Plot of Sepal Length",
     xlab = "Sepal Length",
     col = "blue")

# Adding a histogram
hist(iris$Sepal.Length,
     col = rgb(0,0,1,0.2),
     add = TRUE,
     breaks = 20)
```



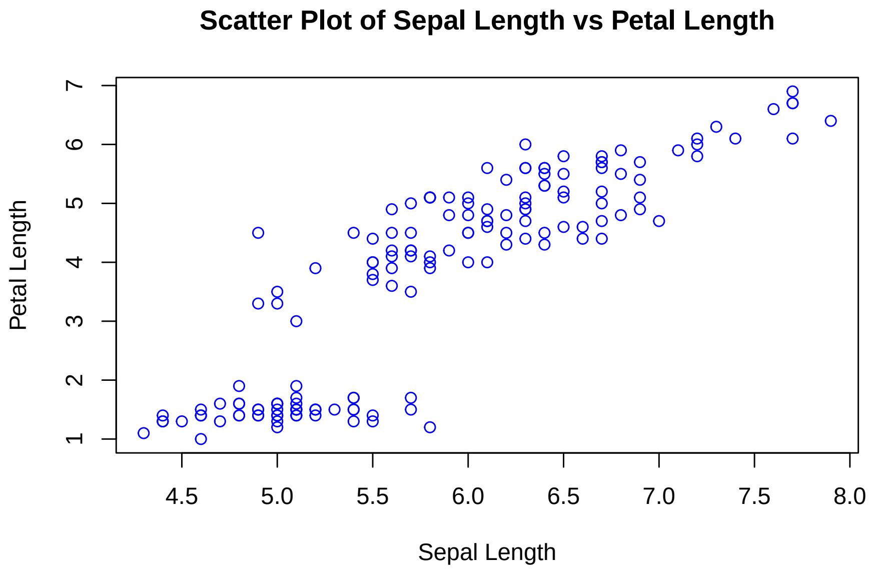
7.4 Scatter Plots and Correlation Analysis:

Scatter plots are useful for visualizing the relationship between two continuous variables.

7.4.1 Using scatter plots to visualize relationships between variables.

Scatter plots visually represent the relationship between two continuous variables. They help to identify patterns, trends, and correlations between variables.

```r
# Creating a scatter plot
plot(iris$Sepal.Length, iris$Petal.Length,
     main = "Scatter Plot of Sepal Length vs Petal Length",
     xlab = "Sepal Length", ylab = "Petal Length",
     col = "blue")
```

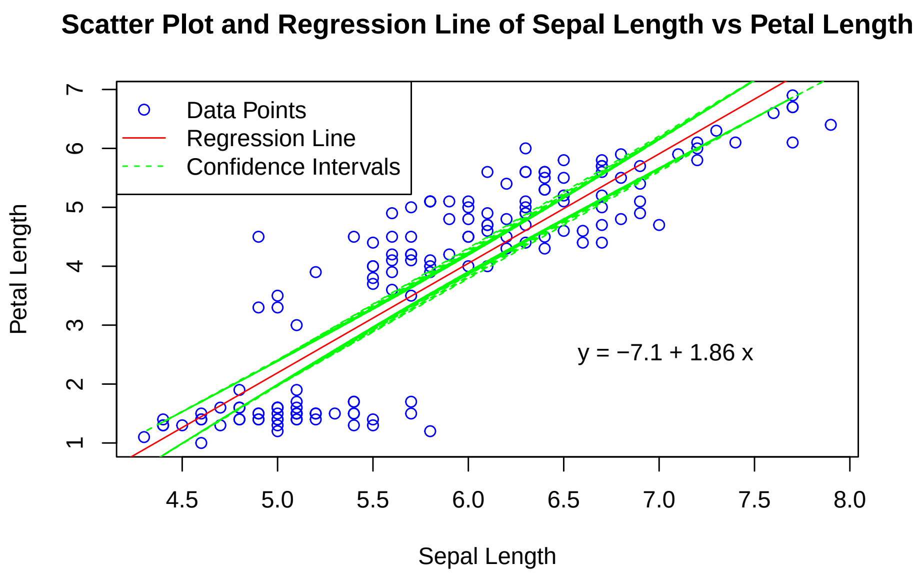


7.4.2 Introduction to adding regression lines and confidence intervals.

- Regression lines provide a summary of the relationship between variables.
- Confidence intervals help assess the uncertainty of the estimated regression line.

```r
# Creating a scatter plot
plot(iris$Sepal.Length, iris$Petal.Length,
     main = "Scatter Plot and Regression Line of Sepal Length vs Petal Length",
     xlab = "Sepal Length", ylab = "Petal Length",
     col = "blue")

# Fit linear regression model
model <- lm(Petal.Length ~ Sepal.Length, data = iris)

# Adding regression line
abline(model, col = "red")

# Adding regression equation
eq <- paste("y =", round(coef(model)[1], 2), "+", round(coef(model)[2], 2), "x")
text(6.5, 2.5, eq, pos = 4)

# Adding confidence intervals
```

```r
conf_interval <- predict(model, interval = "confidence")
lines(iris$Sepal.Length, conf_interval[, "lwr"], col = "green", lty = 2)
lines(iris$Sepal.Length, conf_interval[, "upr"], col = "green", lty = 2)
# Add legend
legend("topleft", legend = c("Data Points", "Regression Line", "Confidence Intervals"),
       col = c("blue", "red", "green"), lty = c(NA, 1, 2), lwd = c(NA, 1, 1),
       pch = c(1, NA, NA), bty = "o" # Enclose the legend in a box
       )
```



7.5 Practical Examples: Exercises

Chapter 8

INTRODUCTION TO ggplot2

8.1 Getting Started with ggplot2

8.1.1 Installation of ggplot2 and setting up the environment.

If you have not installed the **ggplot2** package in your computer, then installing the package for the first time is achieved by writing the code **install.packages("ggplot2")**. Once, the package has successfully been installed, you can set up the environment to use it in your analysis by writing the code **library("ggplot2")**. Alternatively, **ggpot2** package can be loaded by loading the **tidyverse** package which comes while attached with several packages where ggplot2 is among them.

8.1.2 Basic syntax and components of ggplot2: aesthetics, geoms, and layers

- **Aesthetics:** Mapping data variables to visual properties.
- **Geoms:** Geometric objects that represent data points, lines, shapes, etc.
- **Layers:** Building plots by adding layers of graphics.

8.2 Creating Basic Plots with ggplot2

We are using the gapminder dataset (https://www.gapminder.org/data) that has been put into an R package by Bryan (2017) so we can load it with library(gapminder). The dataset includes 1704 observations (rows) of 6 variables (columns: country, continent, year, lifeExp, pop, gdpPercap). country, continent, and year could be thought of as grouping variables, whereas lifeExp (life expectancy), pop (population), and gdpPercap (Gross Domestic Product per capita) are values. The years in this dataset span 1952 to 2007 with 5-year intervals (so a total of 12 different years). It includes 142 countries from 5 continents (Asia, Europe, Africa, Americas, Oceania). You can check that all of the numbers quoted above are correct with these lines:

```
library(tidyverse)
library(gapminder)
gapminder$year %>% unique()
```

```
##  [1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
```

```
gapminder$country %>% n_distinct()
```

```
## [1] 142
```

```
gapminder$continent %>% unique()
```

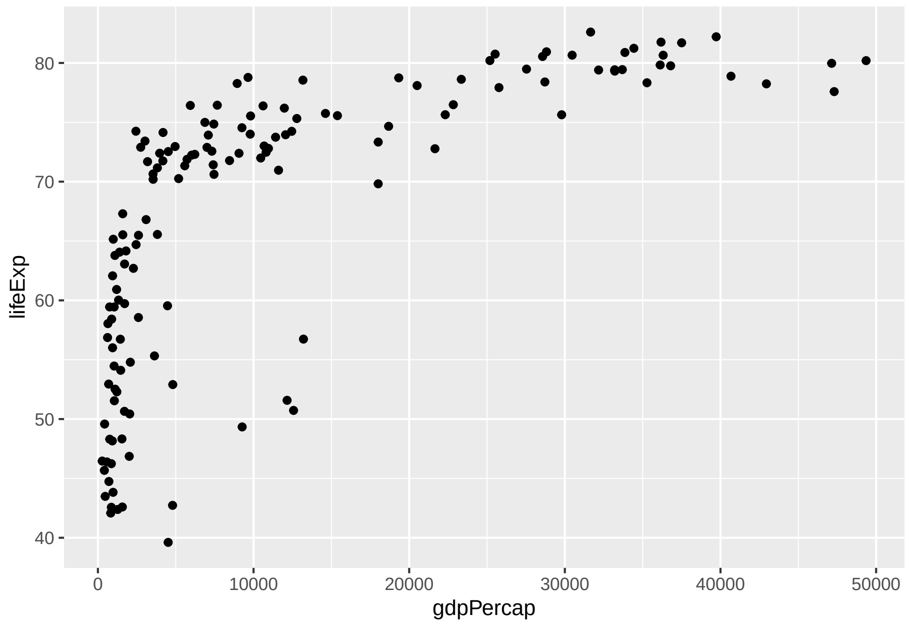
```
## [1] Asia     Europe   Africa   Americas Oceania
## Levels: Africa Americas Asia Europe Oceania
```

8.2.1 Step-by-step instructions to create scatter plots, histograms, and bar charts using ggplot2

8.2.1.1 Scatter plots/bubble plots

Let's create a new shorter tibble called gapdata2007 that only includes data for the year 2007. Let's ask ggplot() to draw a point for each observation by adding geom_point()

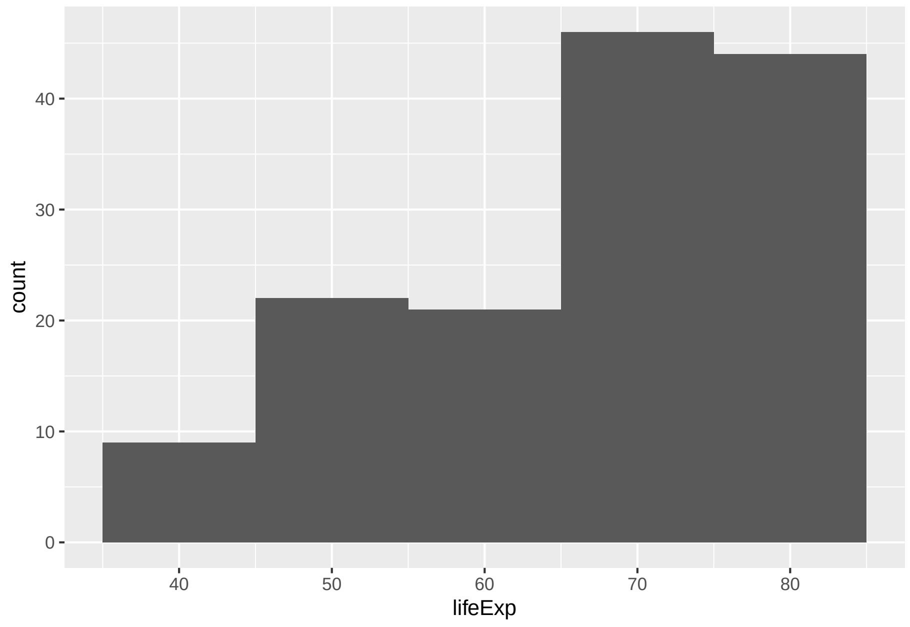
```
# Filter only the year 2007 data
gapdata2007 <- gapminder %>%
filter(year == 2007)

#Plot a simple scatter plot
gapdata2007 %>%
ggplot(aes(x = gdpPercap, y = lifeExp)) +
geom_point()
```



8.2.1.2 Histograms

A histogram displays the distribution of values within a continuous variable. In the example below, we are taking the life expectancy (aes(x = lifeExp)) and telling the histogram to count the observations up in “bins” of 10 years (geom_histogram(binwidth = 10)

```r
gapdata2007 %>%
ggplot(aes(x = lifeExp)) +
geom_histogram(binwidth = 10)
```



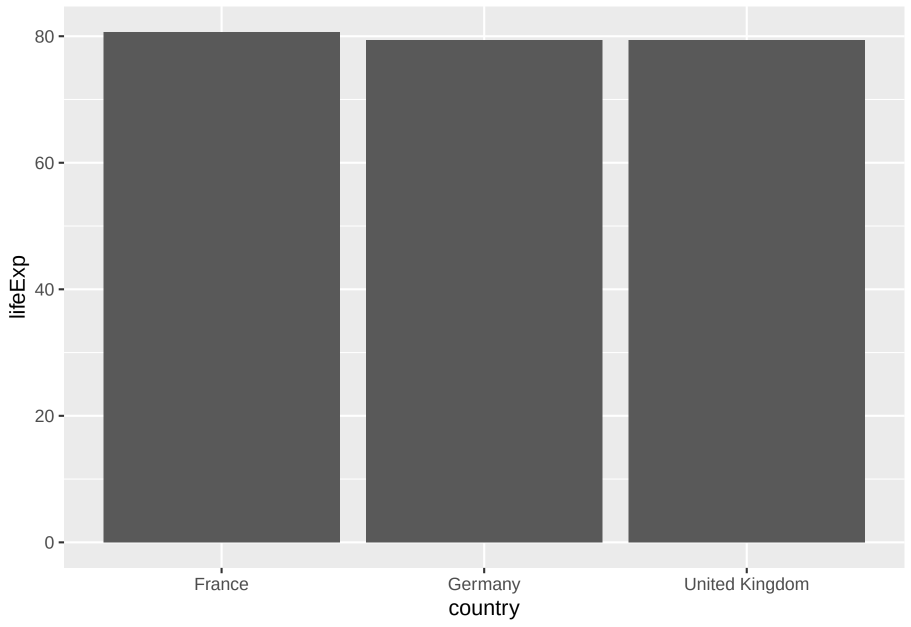
We can see that most countries in the world have a life expectancy of ~70-80 years (in 2007), and that the distribution of life expectancy globally is not normally distributed. Setting the binwidth is optional, using just geom_histogram() works well too - by default, it will divide the data into 30 bins.

8.2.1.3 Bar plots

There are two geoms for making bar plots - **geom_col()** and **geom_bar()** and the examples below will illustrate when to use which one. In short: if your data is already summarised or includes values for y (height of the bars), use **geom_col()**. If, however, you want ggplot() to count up the number of rows in your dataset, use **geom_bar()**. For example, with patient-level data (each row is a patient) you'll probably want to use geom_bar(), with data that is already somewhat aggregated, you'll use geom_col(). There is no harm in trying one, and if it doesn't work, trying the other.

Let's plot the life expectancies in 2007 in these three countries:

```
gapdata2007 %>%
filter(country %in% c("United Kingdom", "France", "Germany")) %>%
ggplot(aes(x = country, y = lifeExp)) +
geom_col()
```



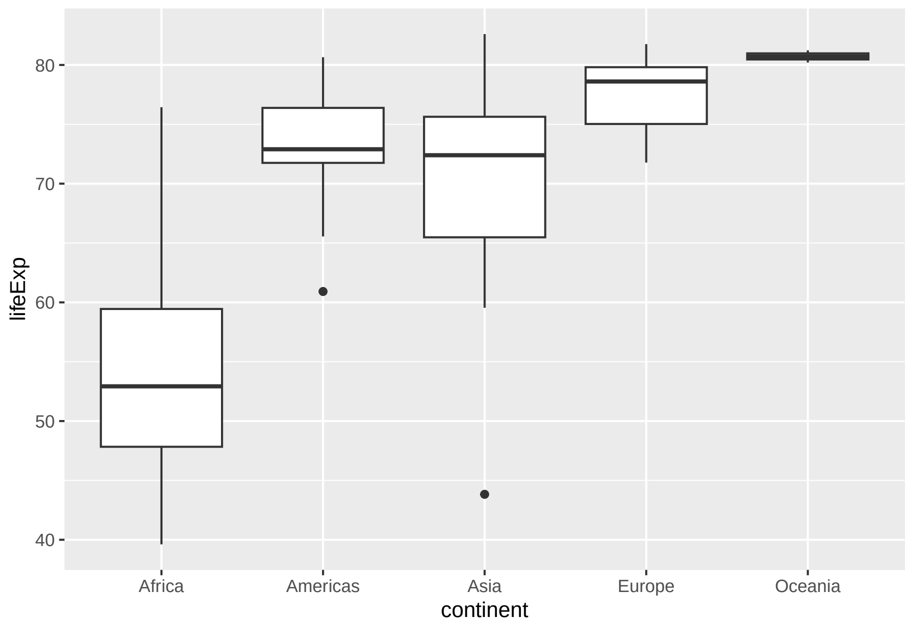
8.2.1.4 Box plots

Box plots are our go to method for quickly visualizing summary statistics of a continuous outcome variable (such as life expectancy in the gapminder dataset. Box plots include:

- the median (middle line in the box)
- inter-quartile range (IQR, top and bottom parts of the boxes - this is where 50% of your data is)
- whiskers (the black lines extending to the lowest and highest values that are still within 1.5*IQR)
- outliers (any observations out with the whiskers)

Let's pot the boxplots of life expectancies within each continent in year 2007

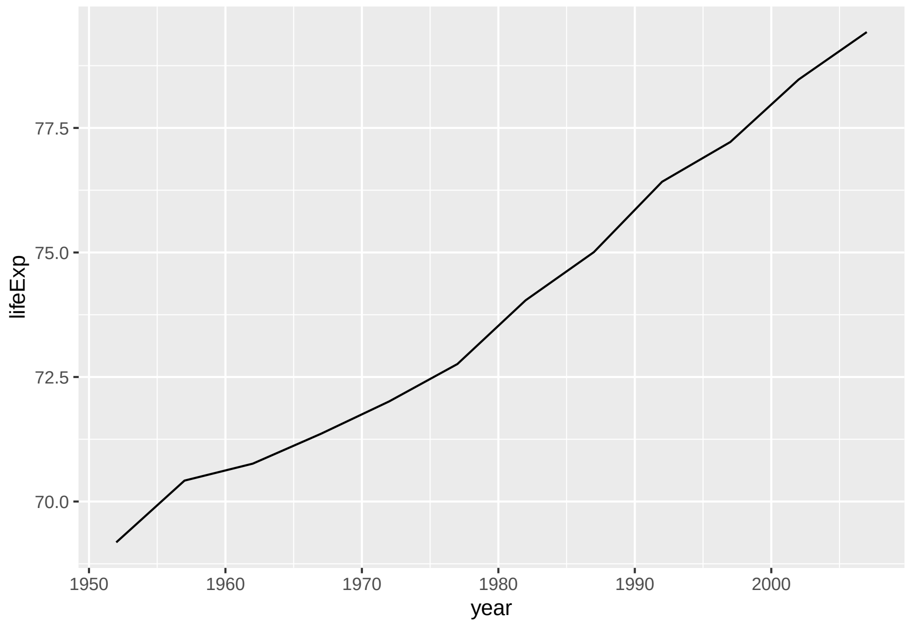
```r
gapdata2007 %>%
ggplot(aes(x = continent, y = lifeExp)) +
geom_boxplot()
```



8.2.1.5 Line plots/time series plots

Let’s plot the life expectancy in the United Kingdom over time.

```
gapdata <- gapminder
gapdata %>%
filter(country == "United Kingdom") %>%
ggplot(aes(x = year, y = lifeExp)) +
geom_line()
```



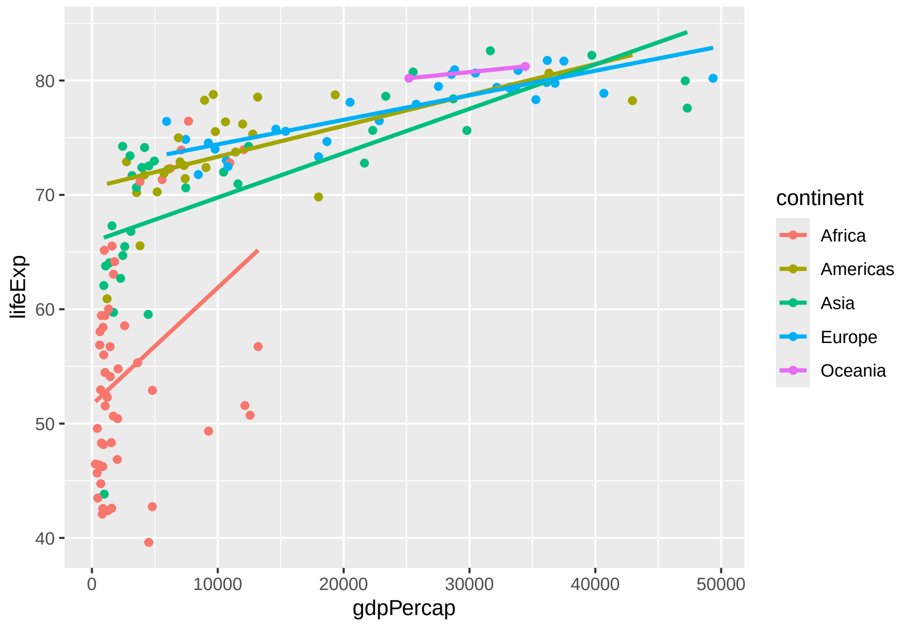
8.2.2 Introduction to adding layers, modifying aesthetics, and utilizing facets for multi-panel plots

a) Adding layers

Going back to the scatter plot (lifeExp vs gdpPercap), let's use continent to give the points some colour. We can do this by adding colour = continent inside the aes():Let's also add a regression line to the scatter plot

```
gapdata2007 %>%
ggplot(aes(x = gdpPercap, y = lifeExp, colour = continent)) +
geom_point()+
geom_smooth(method = "lm", se = FALSE)
```

```
## `geom_smooth()` using formula = 'y ~ x'
```

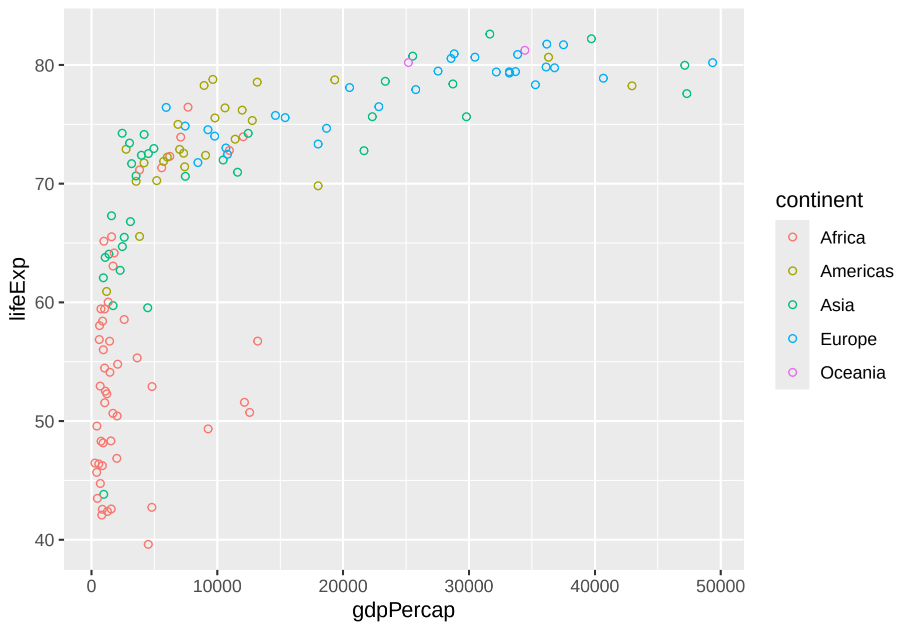


b) Modifying aesthetics

This can be achieved by specifying further variables inside **aes()** or specifying aesthetics outside **aes()**. The main aesthetics (things we can see) are: **x, y, colour, fill, shape, size,** and any of these could appear inside or outside the **aes()** function.

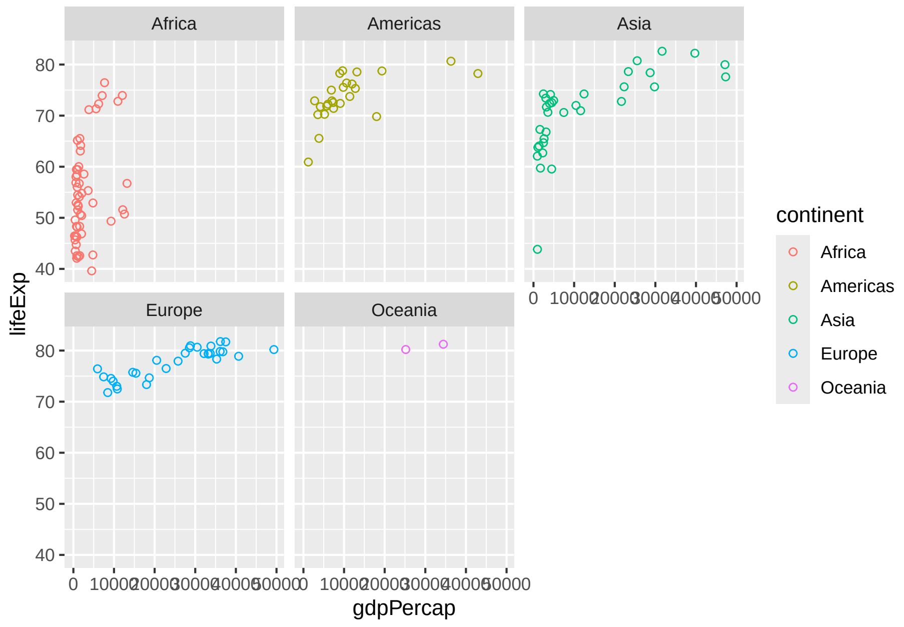
Variables (so columns of your dataset) have to be defined inside **aes()**. Whereas to apply a modification on everything, we can set an aesthetic to a constant value outside of aes().

```
gapdata2007 %>%
ggplot(aes(x = gdpPercap, y = lifeExp, colour = continent)) +
geom_point(shape = 1)
```



c) Utilizing facets for multi-panel plots Faceting is a way to efficiently create the same plot for subgroups within the dataset. For example, we can separate each continent into its own facet by adding facet_wrap(~continent) to our plot:

```
gapdata2007 %>%
ggplot(aes(x = gdpPercap, y = lifeExp, colour = continent)) +
geom_point(shape = 1) +
facet_wrap(~continent)
```



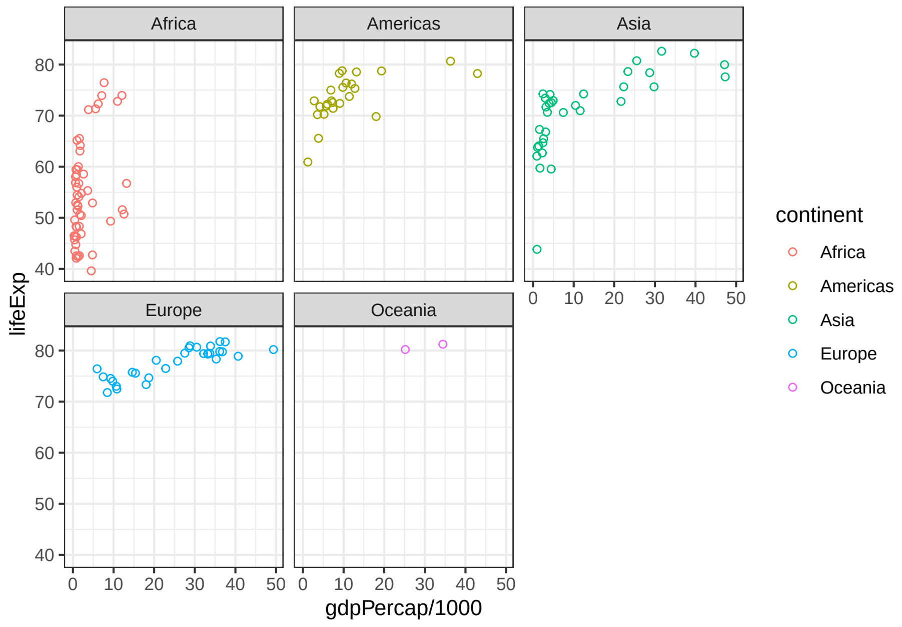
8.3 Customizing Plots in ggplot2

8.3.1 Advanced customization techniques including themes, scales, and coordinate systems.

a) Themes

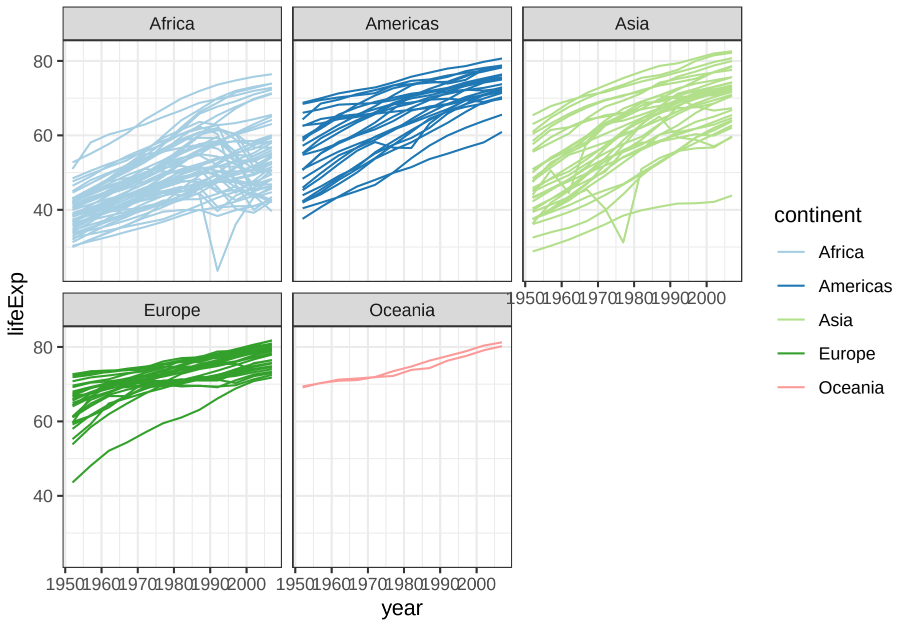
Changing themes involves changing the default background from grey to a white background. Some of the built-in ggplot themes are **theme_bw()**, **theme_dark()**, **theme_minimal()** and **theme_classic()**. We are adding **theme_bw()** ("background white") to give the plot a different look. We have also divided the gdpPercap by 1000 (making the units "thousands of dollars per capita"). Note that you can apply calculations directly on ggplot variables (so how we've done x = gdpPercap/1000 here). This is how ggplot() works - you can build a plot by adding or modifying things one by one.

```
gapdata2007 %>%
ggplot(aes(x = gdpPercap/1000, y = lifeExp, colour = continent)) +
geom_point(shape = 1) +
facet_wrap(~continent) +
theme_bw()
```



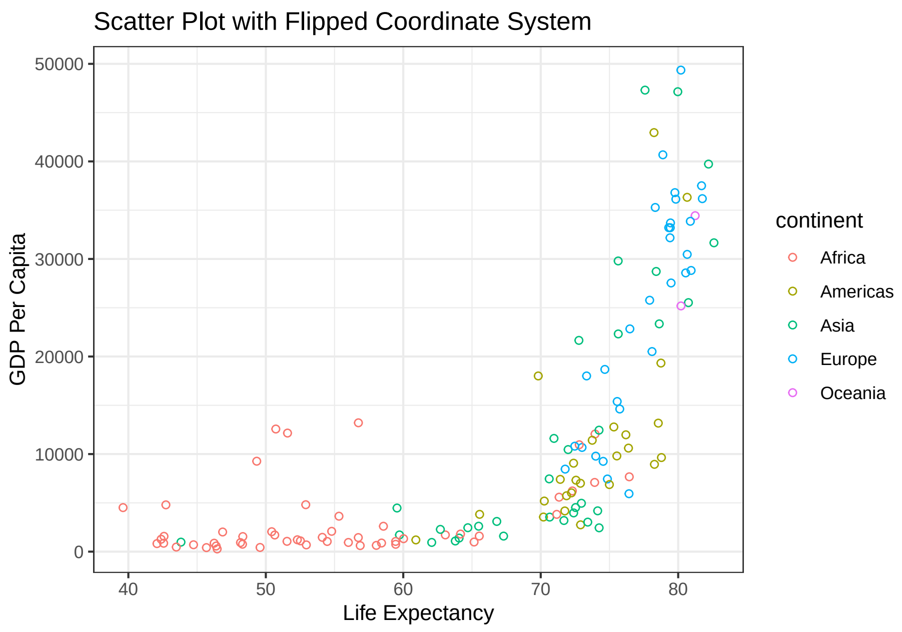
b) Scales

```
gapminder %>%
ggplot(aes(x = year, y = lifeExp, group = country, colour = continent)) +
geom_line() +
facet_wrap(~continent) +
theme_bw() +
scale_colour_brewer(palette = "Paired")
```

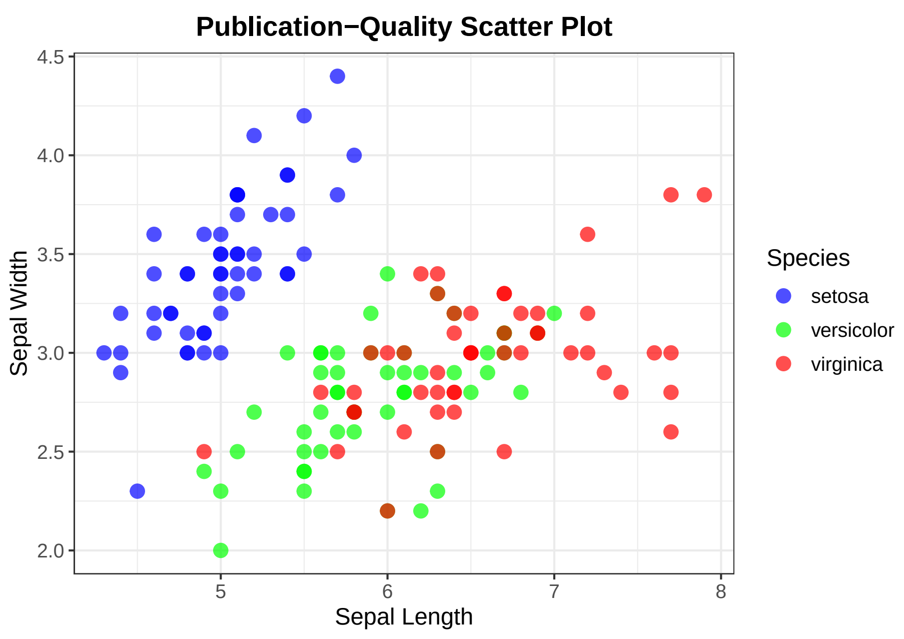


c) Coordinate systems

```
# Flipped coordinate system
gapdata2007 %>%
ggplot(aes(x = gdpPercap, y = lifeExp, colour = continent)) +
geom_point(shape = 1)+
  coord_flip() +
  labs(title = "Scatter Plot with Flipped Coordinate System",
      x = "GDP Per Capita", y = "Life Expectancy") +
  theme_bw()
```



```r
# Publication-quality scatter plot
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point(alpha = 0.7, size = 3) +
  labs(title = "Publication-Quality Scatter Plot",
       x = "Sepal Length", y = "Sepal Width") +
  theme_bw() +
  theme(
    plot.title = element_text(size = 14, face = "bold", hjust = 0.5),
    axis.title = element_text(size = 12),
    axis.text = element_text(size = 10),
    legend.title = element_text(size = 12),
    legend.text = element_text(size = 10)
  ) +
  scale_color_manual(values = c("setosa" = "blue", "versicolor" = "green", "virginica" = "red"))
```



8.4 Practical Examples: Exercise

Chapter 9

ADVANCED DATA VISUALIZATION WITH ggplot2 AND OTHER PACKAGES

9.1 Interactive Graphs with plotly

9.1.1 Converting ggplot2 charts into interactive plotly graphs

Before converting the ggplot2 chart to interactive plotly graph, you assign the plot to a vector, say, p. Then load the **plotly** package and convert the ggplot chart to interactive plotly graph using the **ggpotly()** function, for example, **ggplotly(p)**.

```r
library(plotly)
p<-ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point(alpha = 0.7, size = 3) +
  labs(title = "Publication-Quality Scatter Plot",
       x = "Sepal Length", y = "Sepal Width") +
  theme_bw() +
  theme(
    plot.title = element_text(size = 14, face = "bold", hjust = 0.5),
    axis.title = element_text(size = 12),
    axis.text = element_text(size = 10),
    legend.title = element_text(size = 12),
    legend.text = element_text(size = 10)
  ) +
  scale_color_manual(values = c("setosa" = "blue", "versicolor" = "green", "virginica"

# Convert ggplot2 to plotly
#ggplotly(p)
```

9.1.2 Customizing interactions, adding tooltips, and embedding plotly graphs in web applications.

Customizing Interactions and Adding Tooltips

```r
# Convert ggplot2 scatter plot to plotly with custom tooltip
p <- ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species,
                             text = paste("Species: ", Species))) +
  geom_point() +
  labs(title = "Interactive Scatter Plot of Sepal Length vs Sepal Width",
       x = "Sepal Length", y = "Sepal Width") +
  theme_bw()

# Convert ggplot2 to plotly with custom tooltip
#ggplotly(p, tooltip = "text")
```

Embedding plotly Graphs in Web Applications (Shiny)

```r
# Install and load Shiny package
#install.packages("shiny")
library(shiny)
```

```
## Warning: package 'shiny' was built under R version 4.3.3
```

```r
# Define UI
ui <- fluidPage(
  titlePanel("Interactive Scatter Plot"),
  plotlyOutput("scatter_plot")
)

# Define server logic
server <- function(input, output) {
  output$scatter_plot <- renderPlotly({
    ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species,
                            text = paste("Species: ", Species))) +
      geom_point() +
      labs(title = "Interactive Scatter Plot of Sepal Length vs Sepal Width",
           x = "Sepal Length", y = "Sepal Width") +
      theme_minimal() %>%
      ggplotly(tooltip = "text")
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

```
##
## Listening on http://127.0.0.1:3755
```

Interactive Scatter Plot

9.2 Building Web Applications with shiny

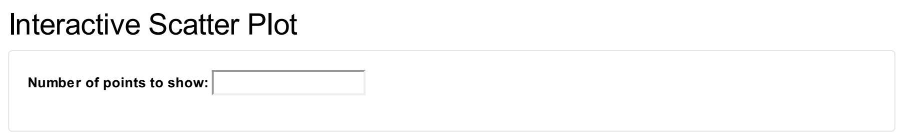
9.2.1 Introduction to shiny for building interactive web applications

Shiny allows you to turn your analyses into interactive web applications without needing to know HTML, CSS, or JavaScript. It works by allowing you to separate the user interface from the underlying R code, making it easy to create complex interactive apps.

9.2.2 Creating reactive plots and dashboards that update

```
# Load necessary packages
#library(shiny)
#library(ggplot2)

# Define UI
ui <- fluidPage(
  titlePanel("Interactive Scatter Plot"),
  sidebarLayout(
    sidebarPanel(
      sliderInput("slider", "Number of points to show:",
                  min = 10, max = nrow(iris), value = 50)
    ),
    mainPanel(
      plotOutput("scatter_plot")
    )
  )
)

# Define server logic
server <- function(input, output) {
  output$scatter_plot <- renderPlot({
    sample_rows <- sample(nrow(iris), input$slider)
    ggplot(data = iris[sample_rows, ], aes(x = Sepal.Length, y = Sepal.Width, color = S
      geom_point() +
      labs(title = "Interactive Scatter Plot of Sepal Length vs Sepal Width",
           x = "Sepal Length", y = "Sepal Width") +
      theme_minimal()
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

```
##
## Listening on http://127.0.0.1:3847
```



Interactive Scatter Plot

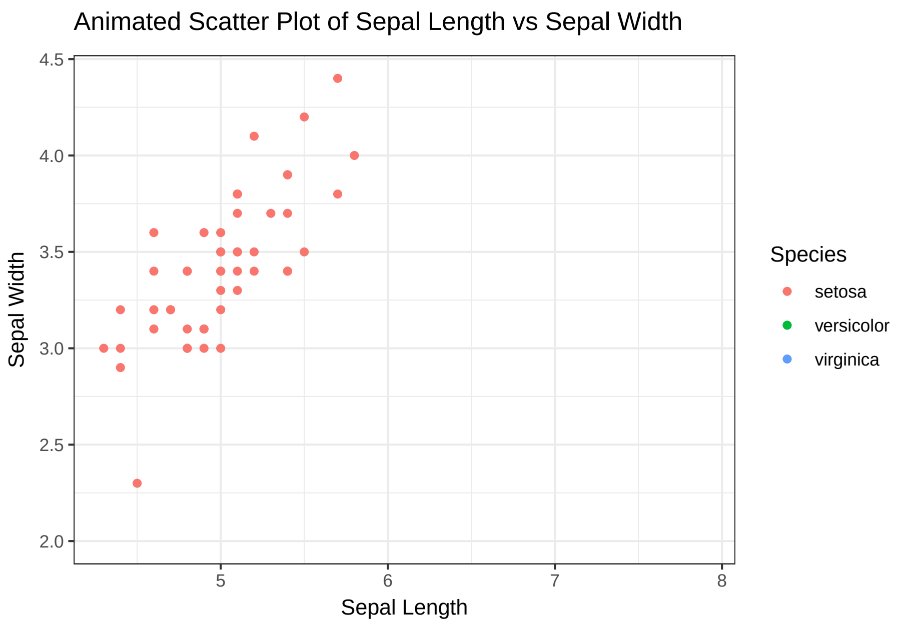
Number of points to show:

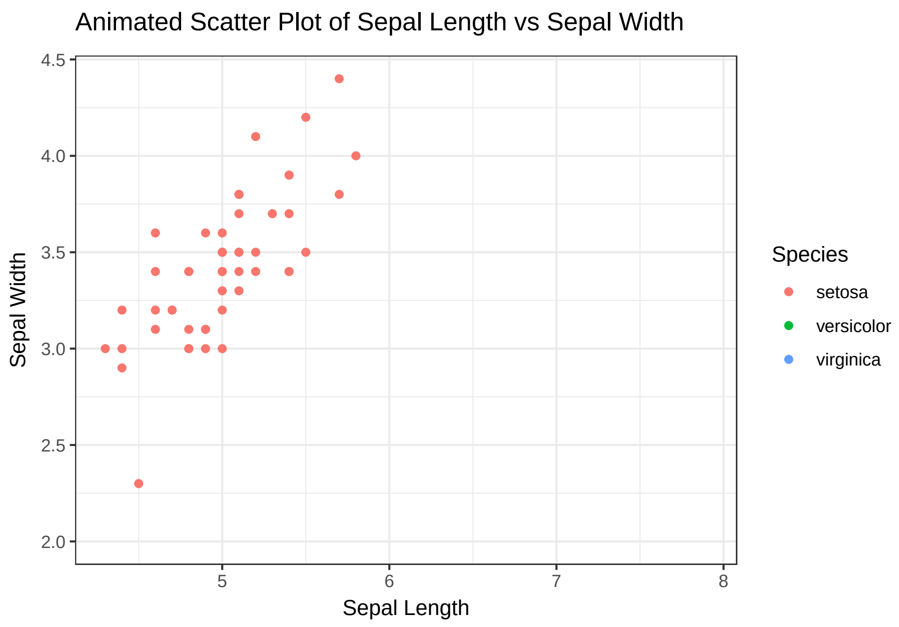
9.3 Advanced ggplot2 Extensions

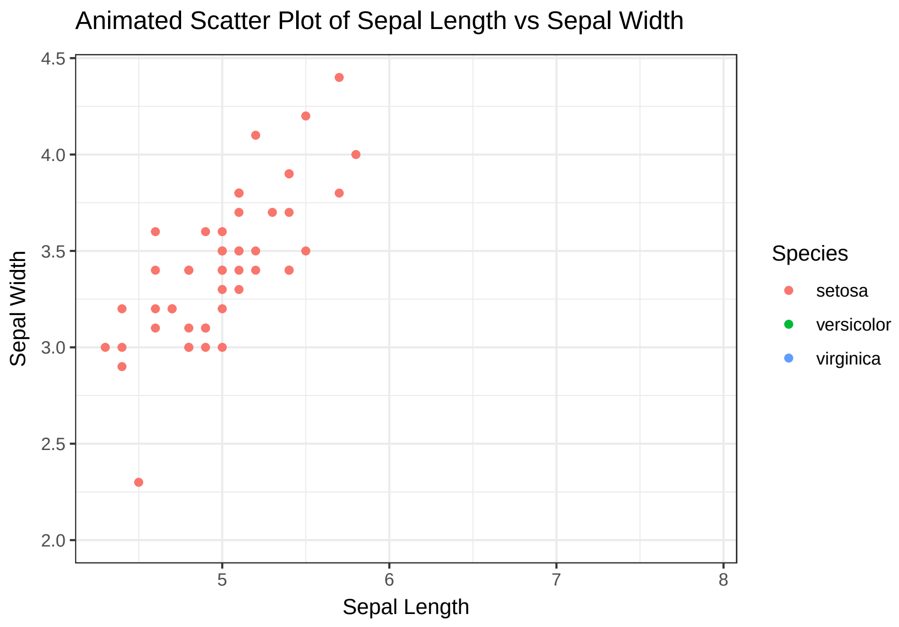
9.3.1 Explore extensions like gganimate for creating animated plots.

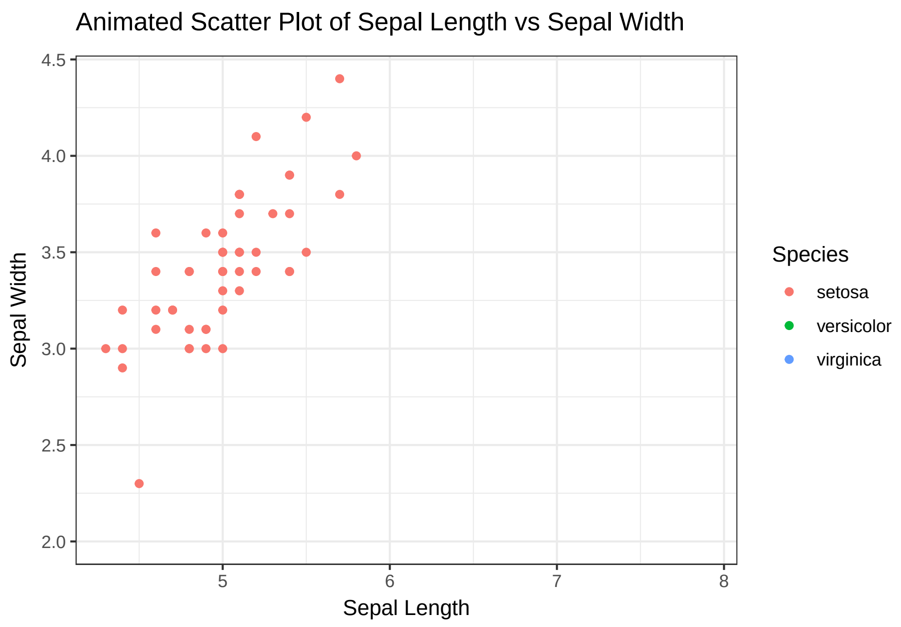
```r
library(gganimate)

#Note: This plot will be displayed after you install the gifski package
```

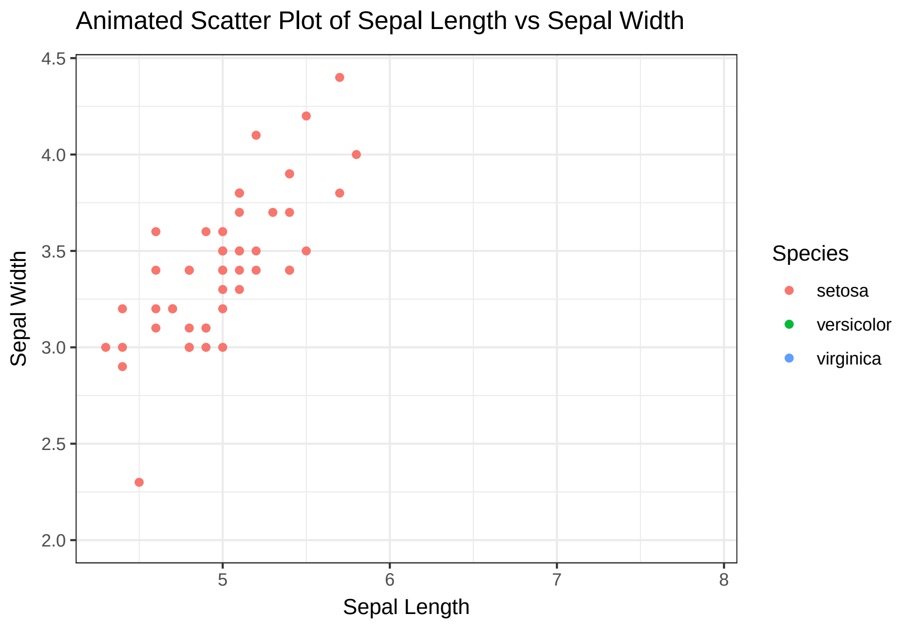
```
# Animated scatter plot
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point() +
  transition_states(Species, transition_length = 2, state_length = 1) +
  labs(title = "Animated Scatter Plot of Sepal Length vs Sepal Width",
       x = "Sepal Length", y = "Sepal Width") +
  theme_bw()
```

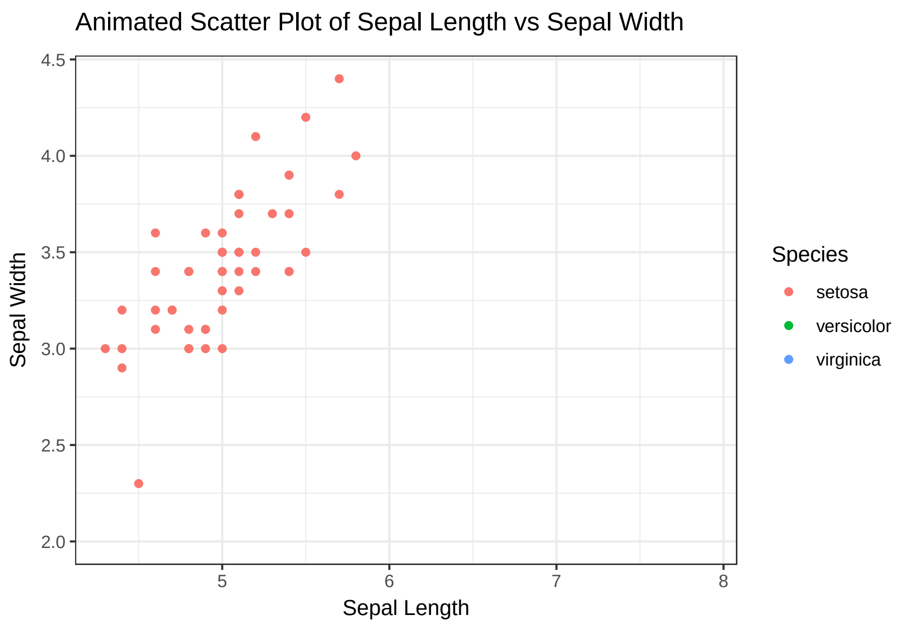


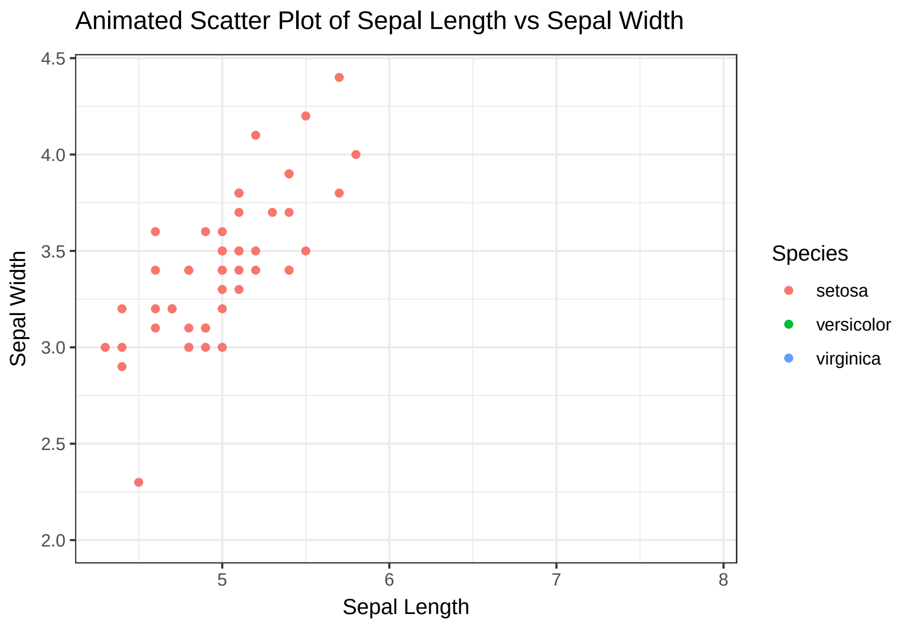

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

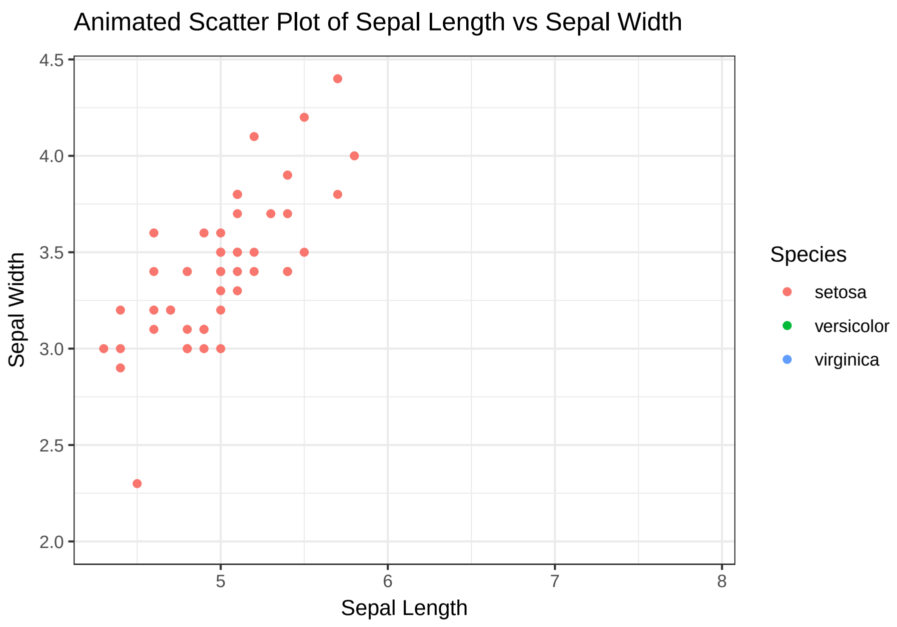

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

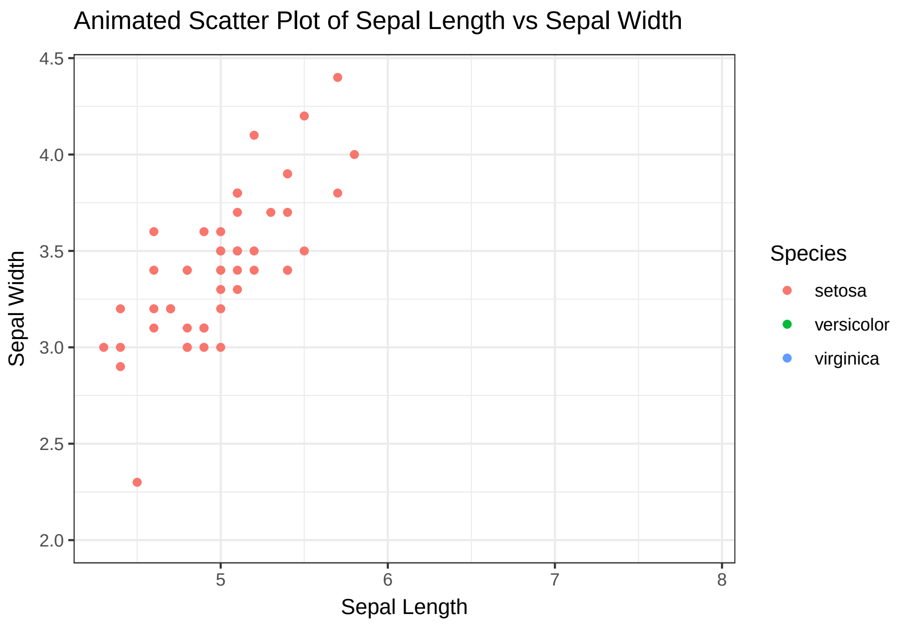

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
Species
setosa
versicolor
virginica

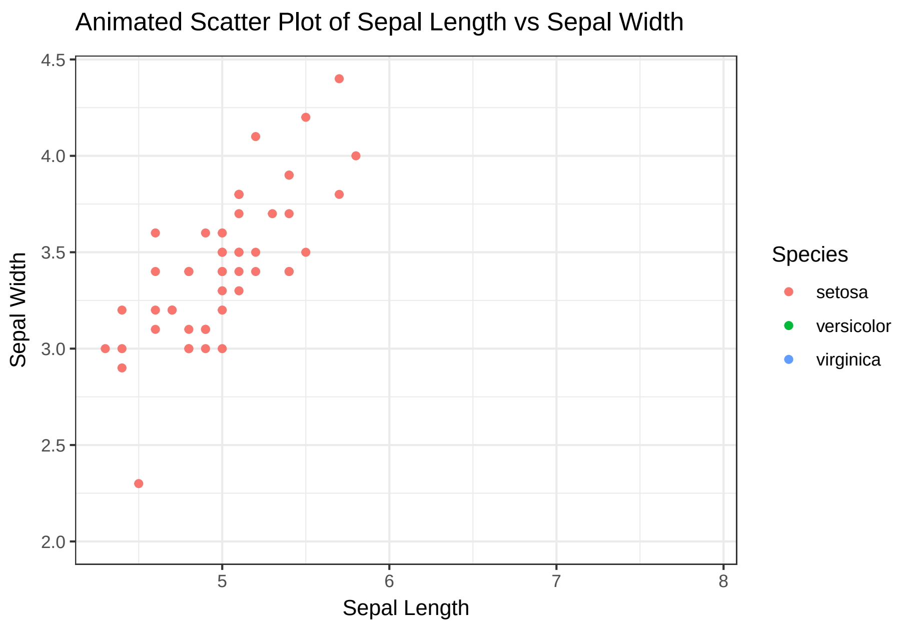

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
Species
setosa
versicolor
virginica

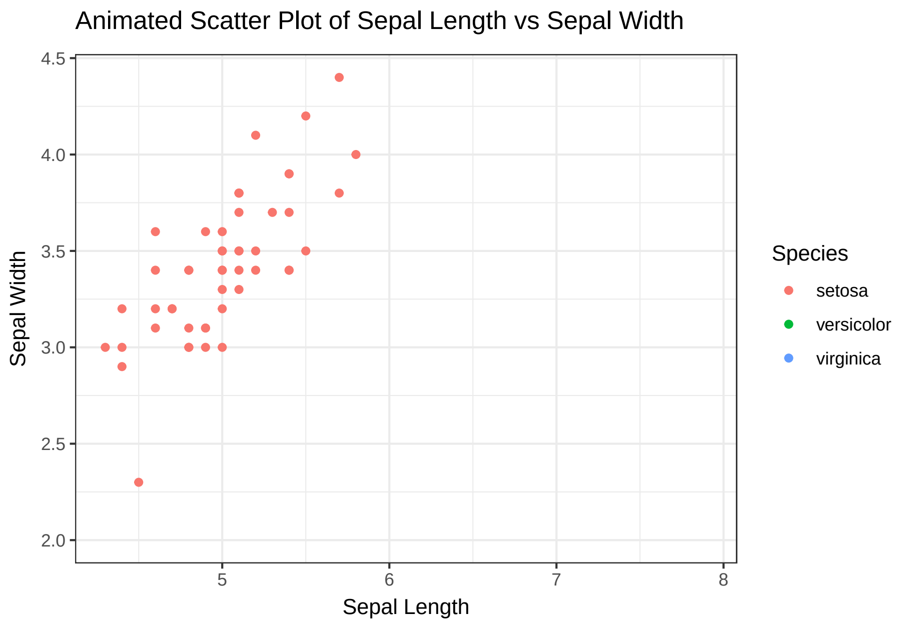

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

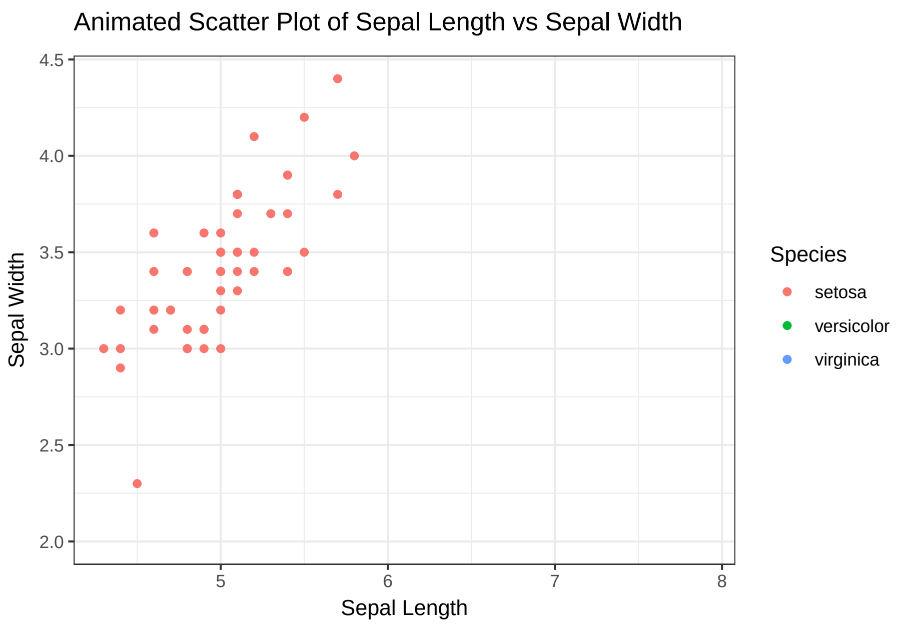

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

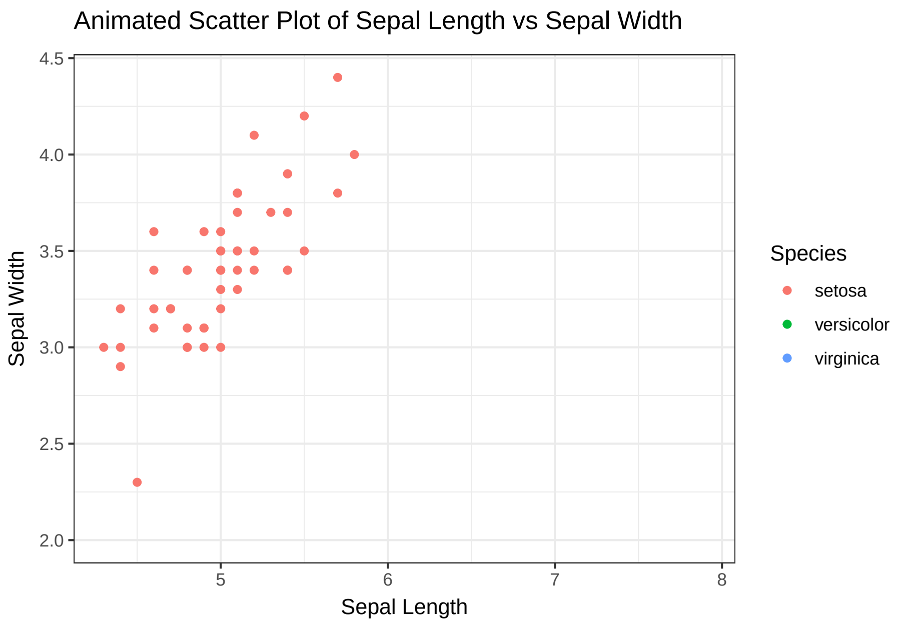

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

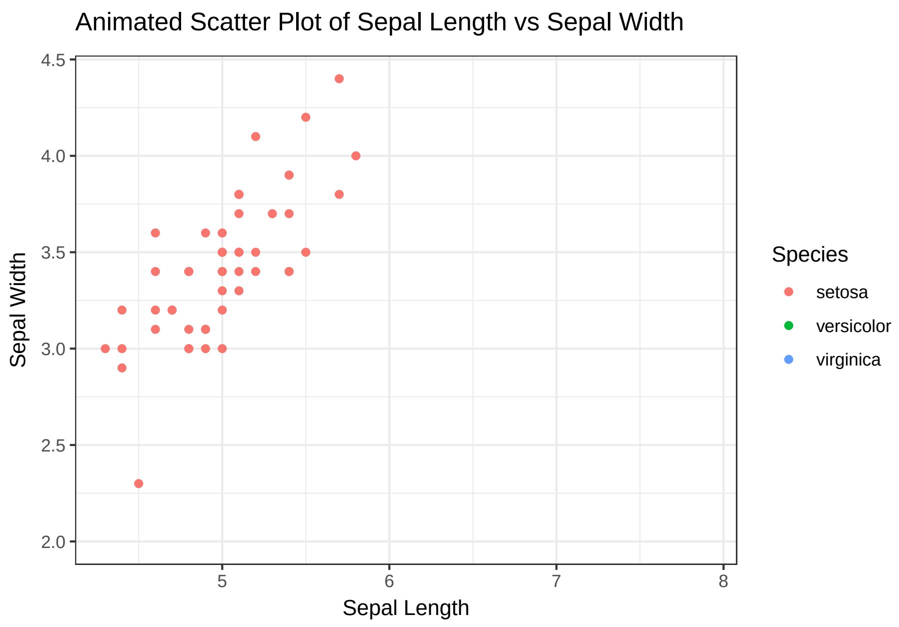

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

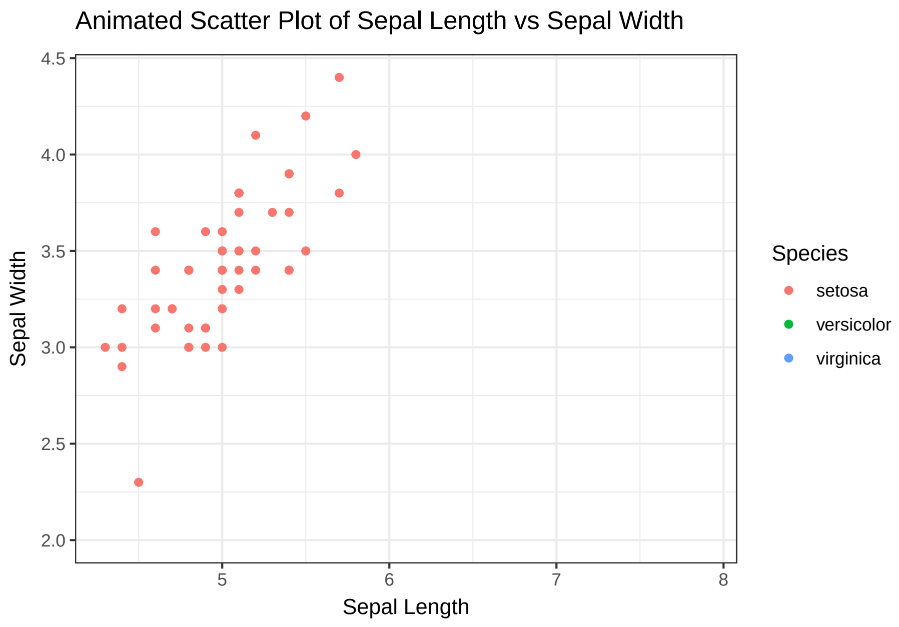

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

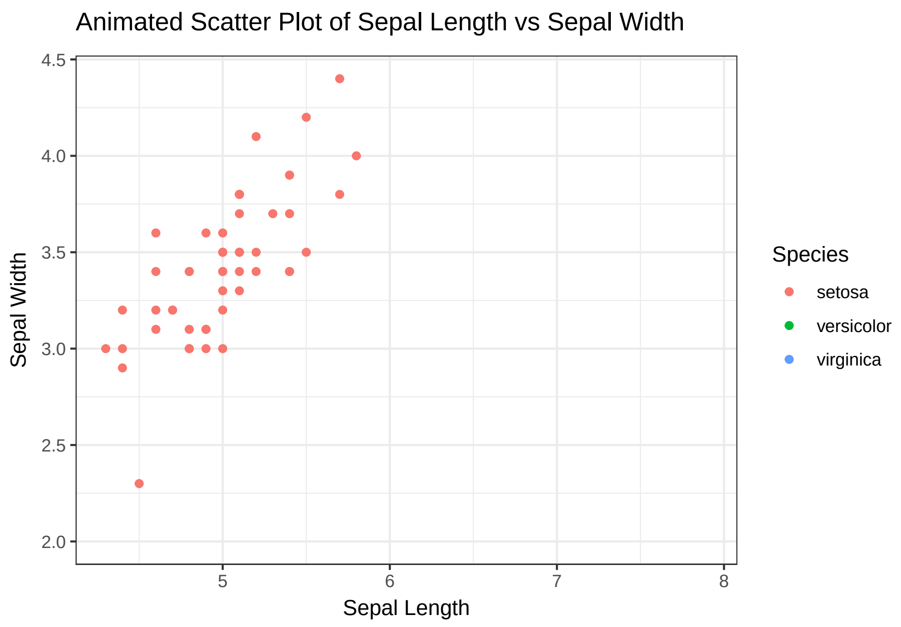

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

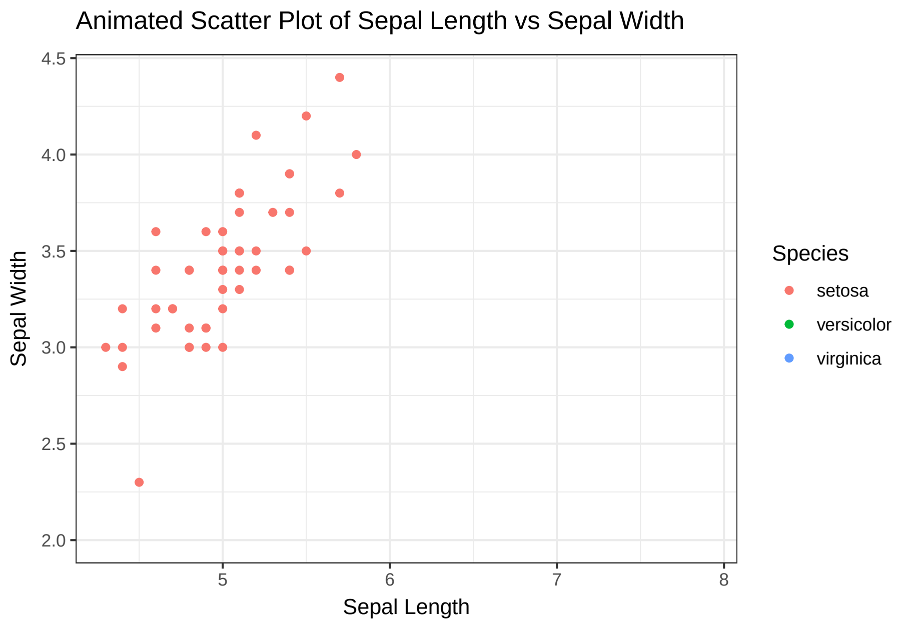

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

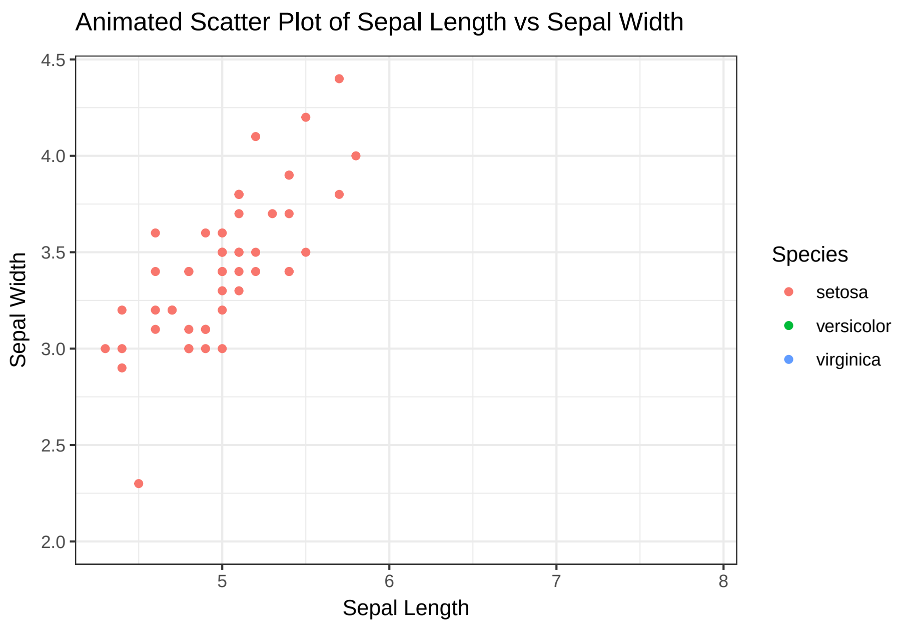

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

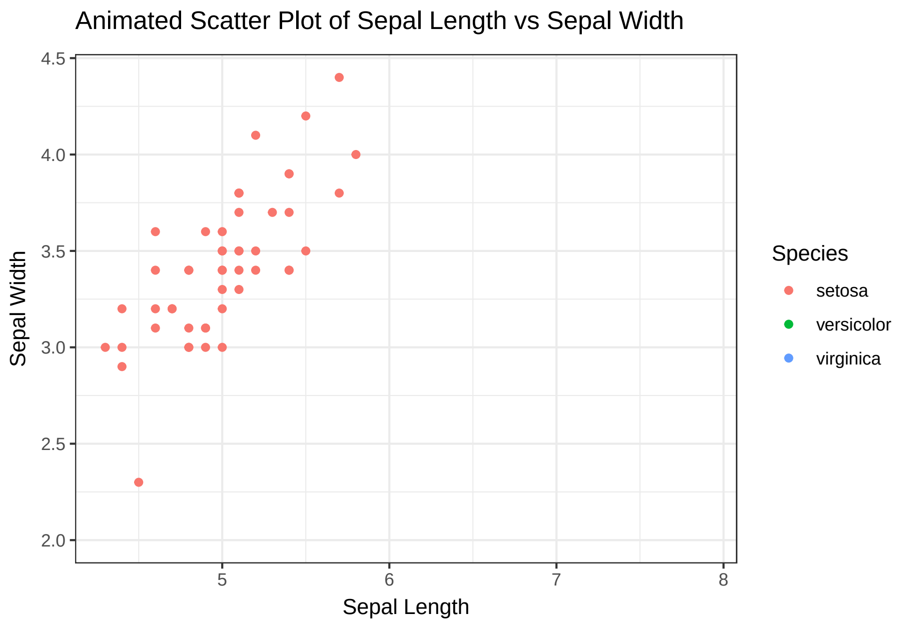

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

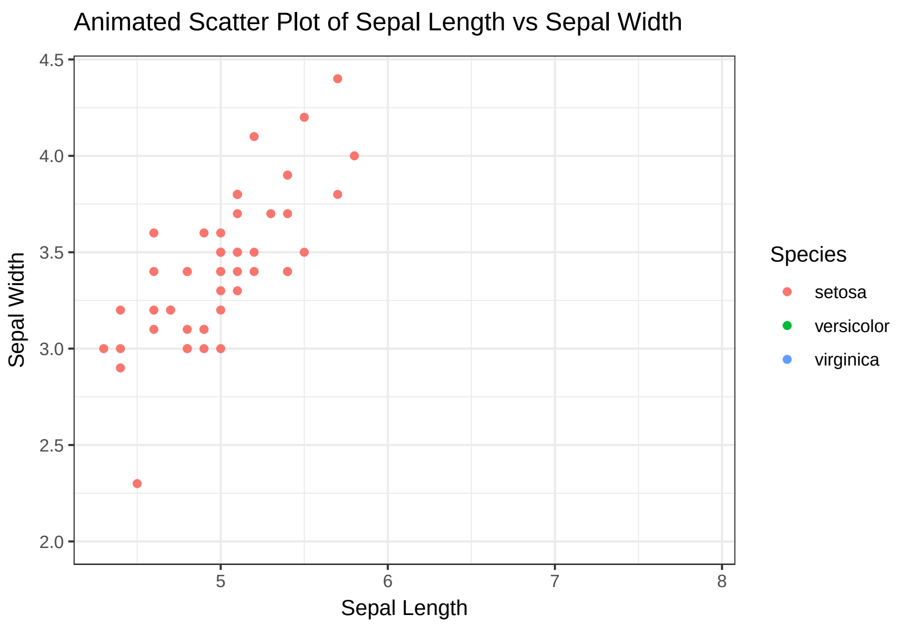

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

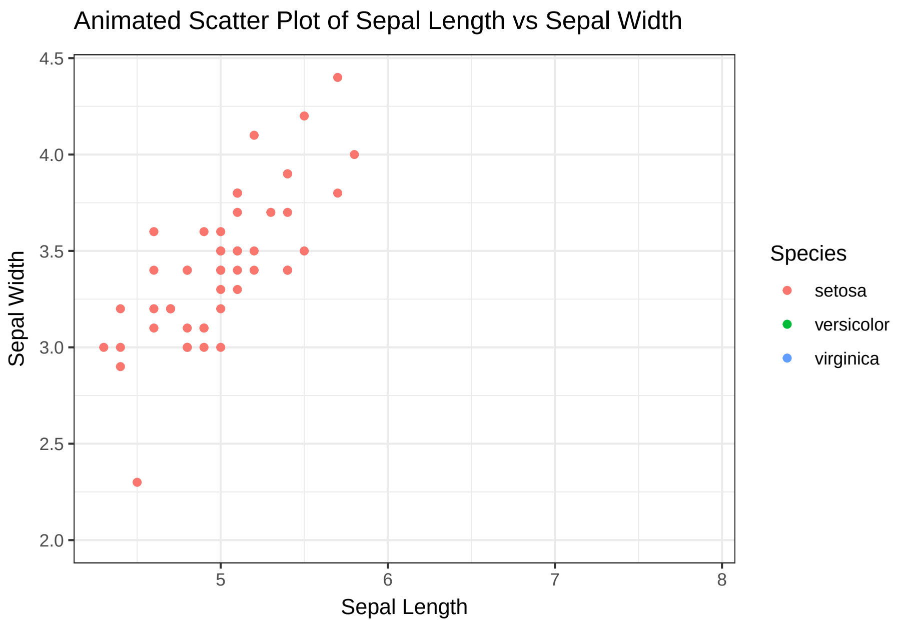

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

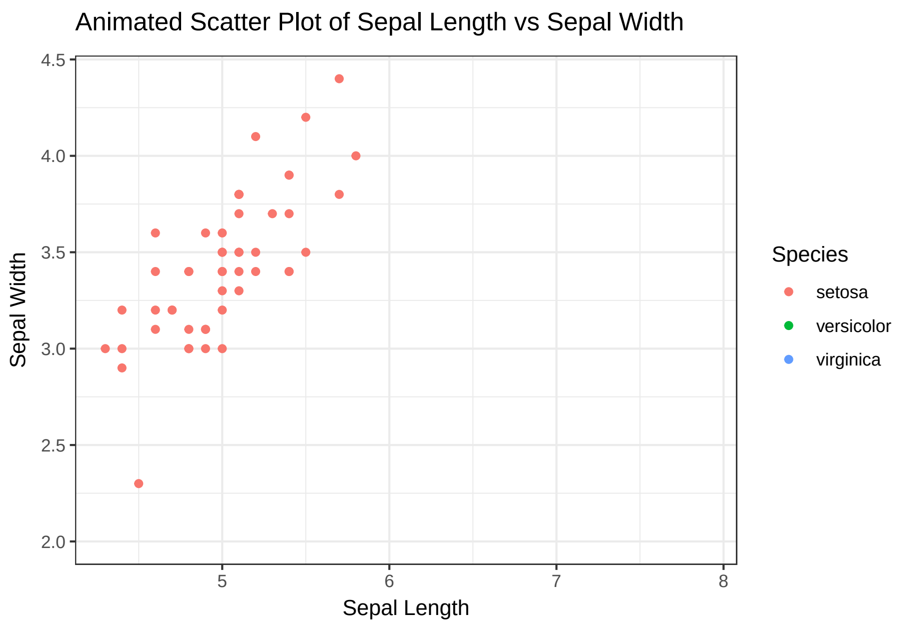

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

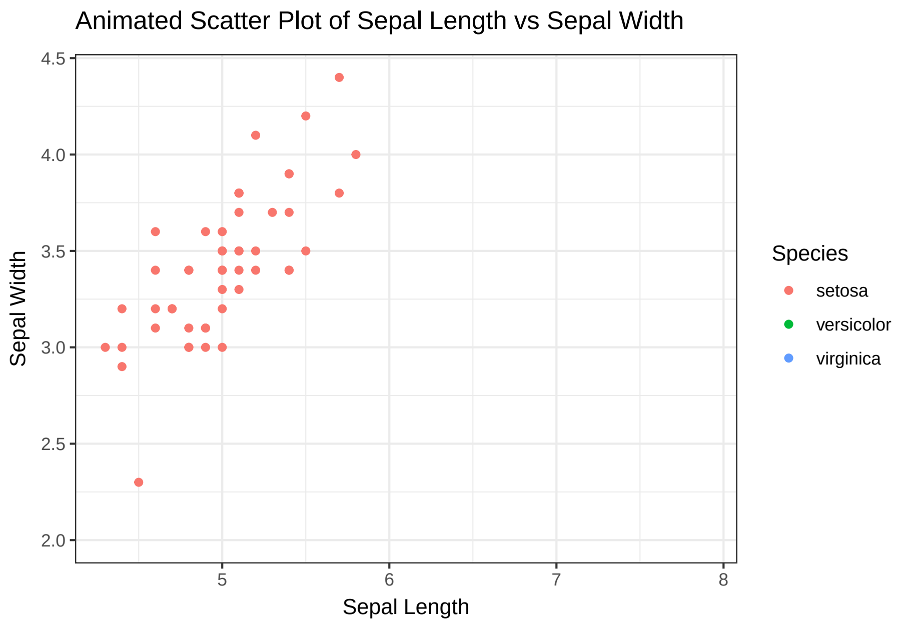

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

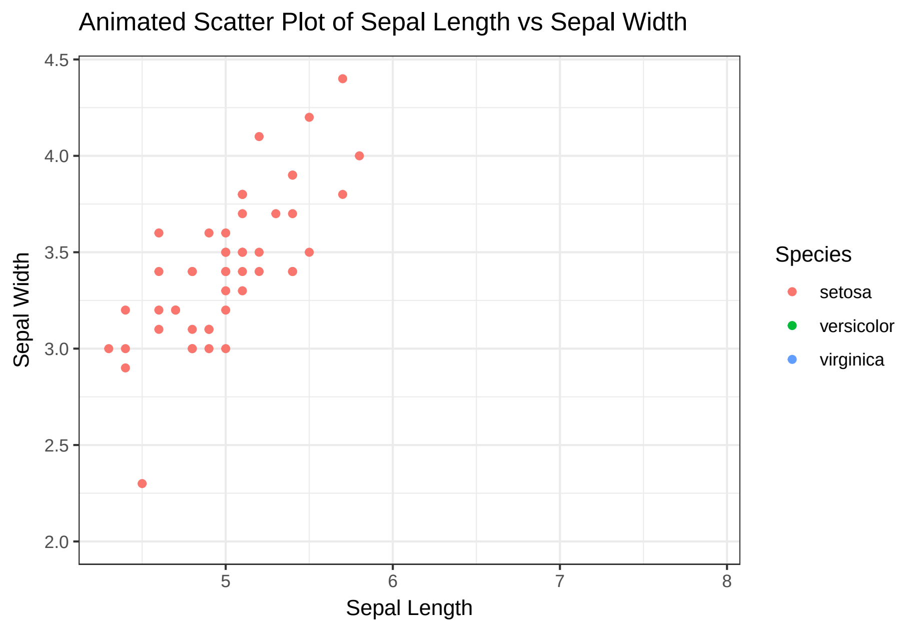

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

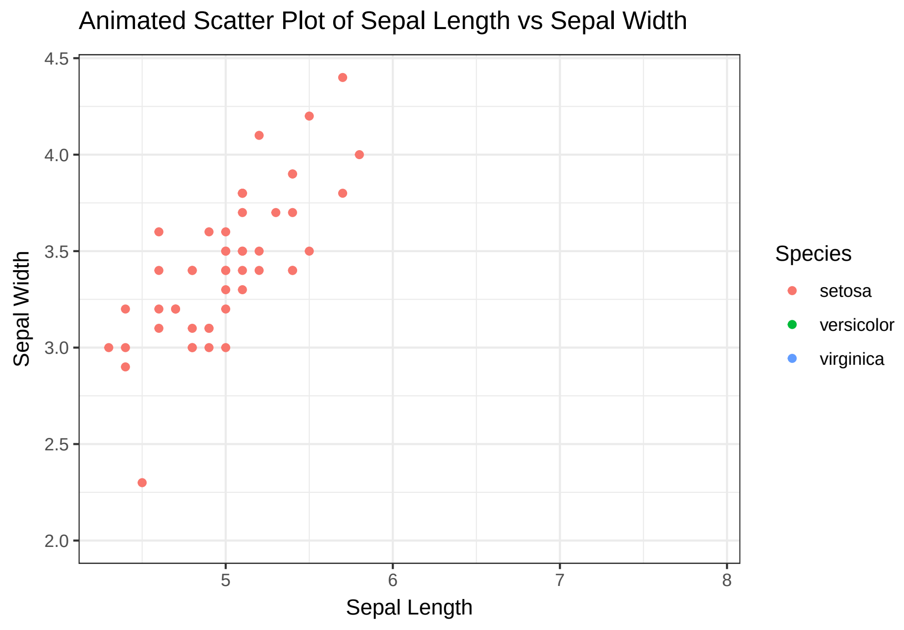

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

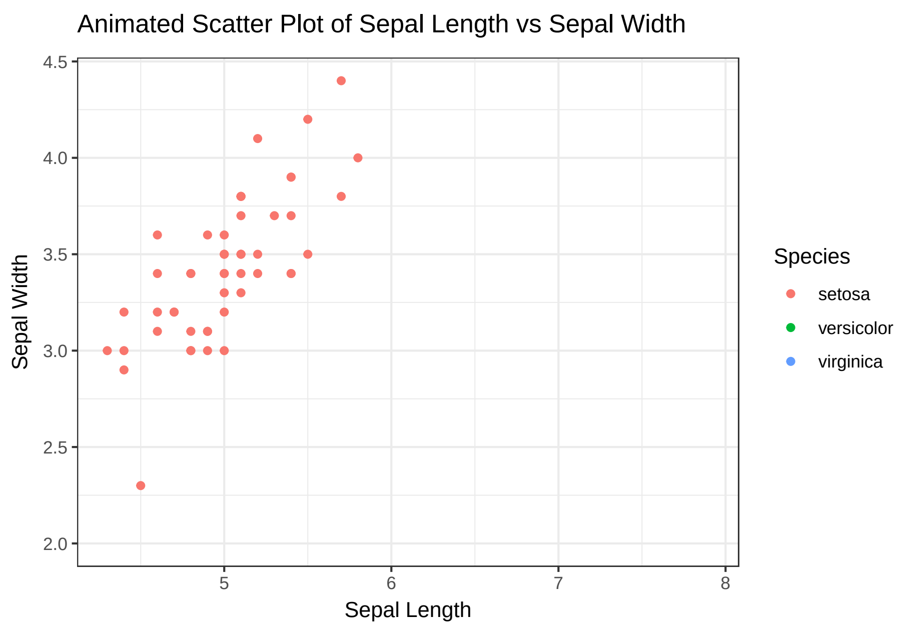

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

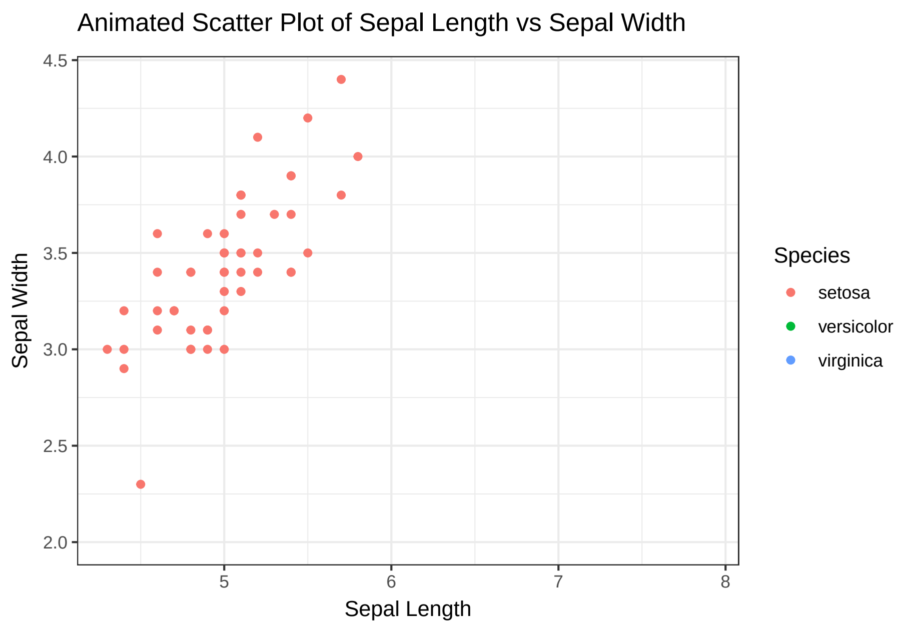

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

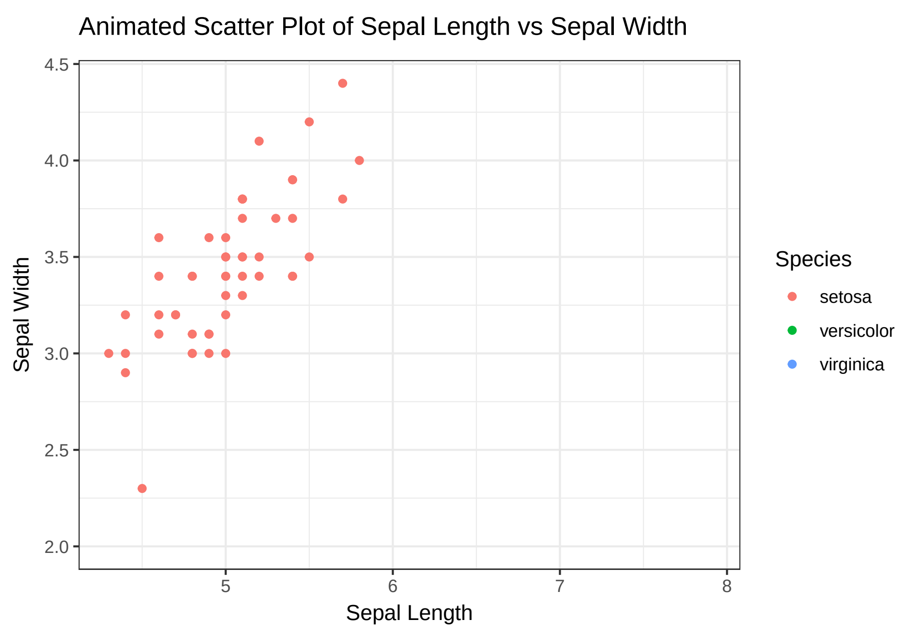

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

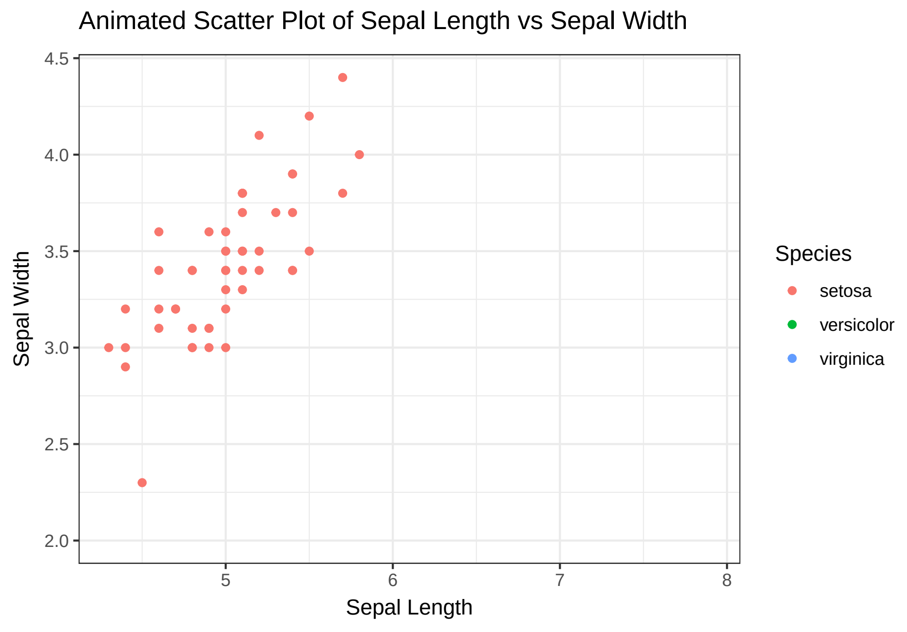

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

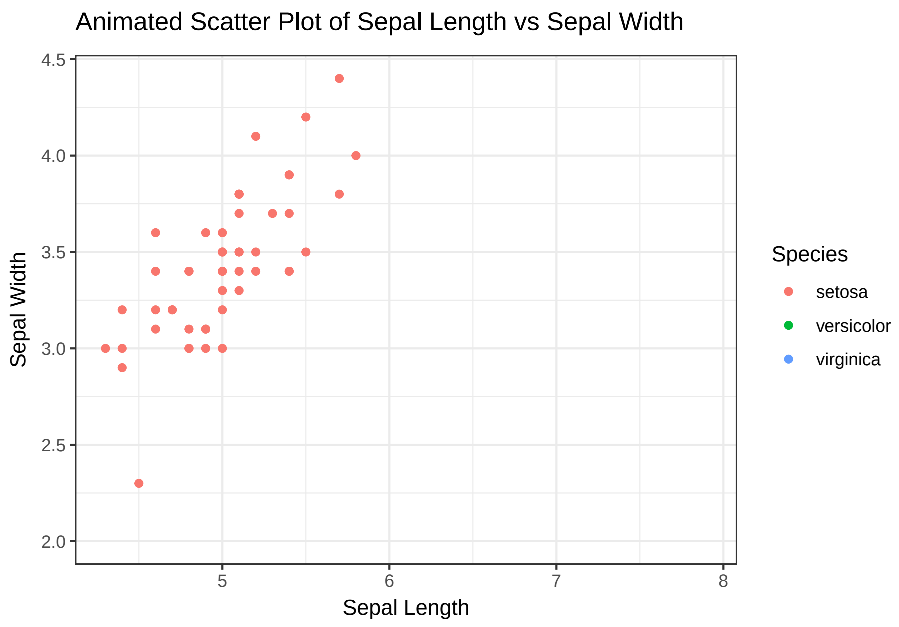

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

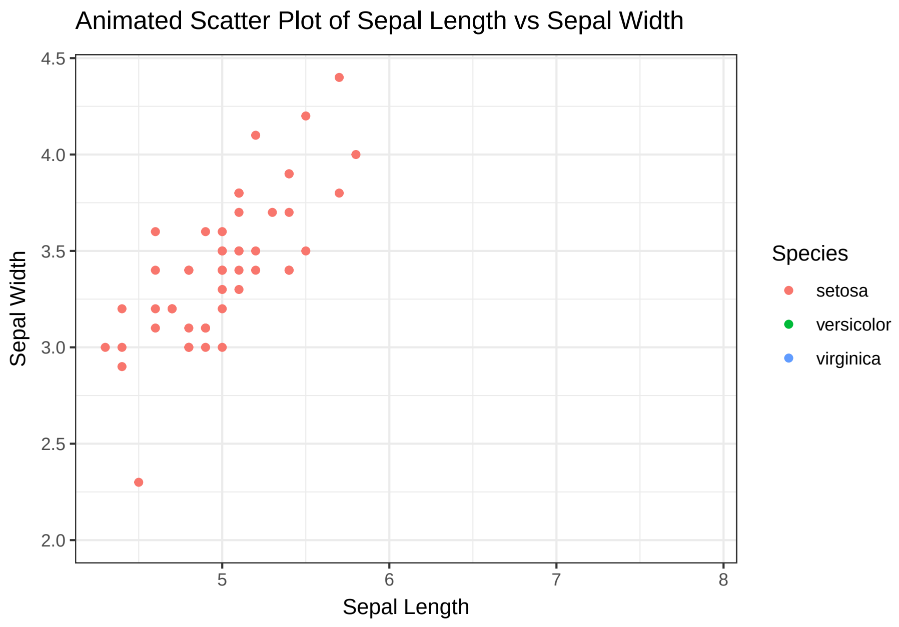

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

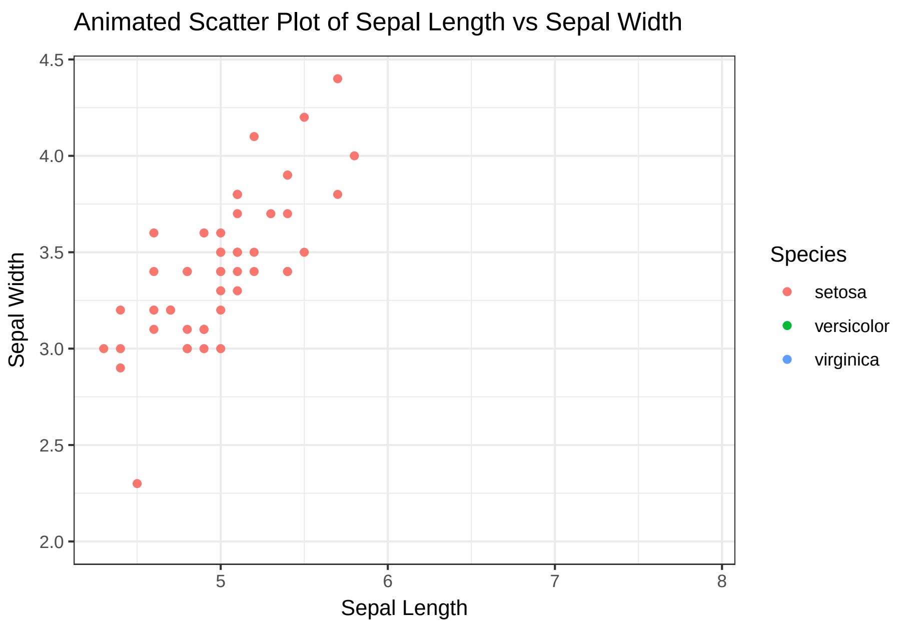

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

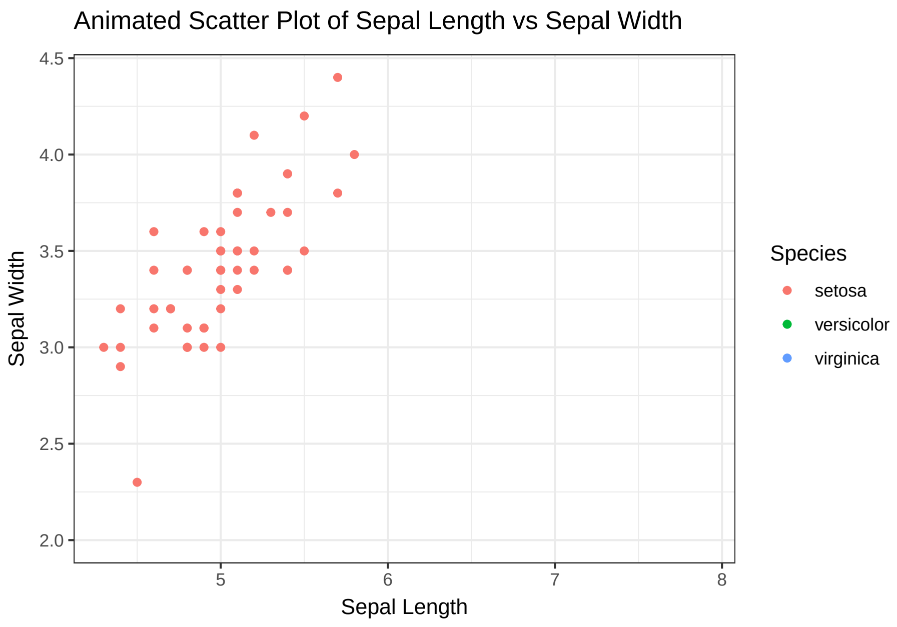

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

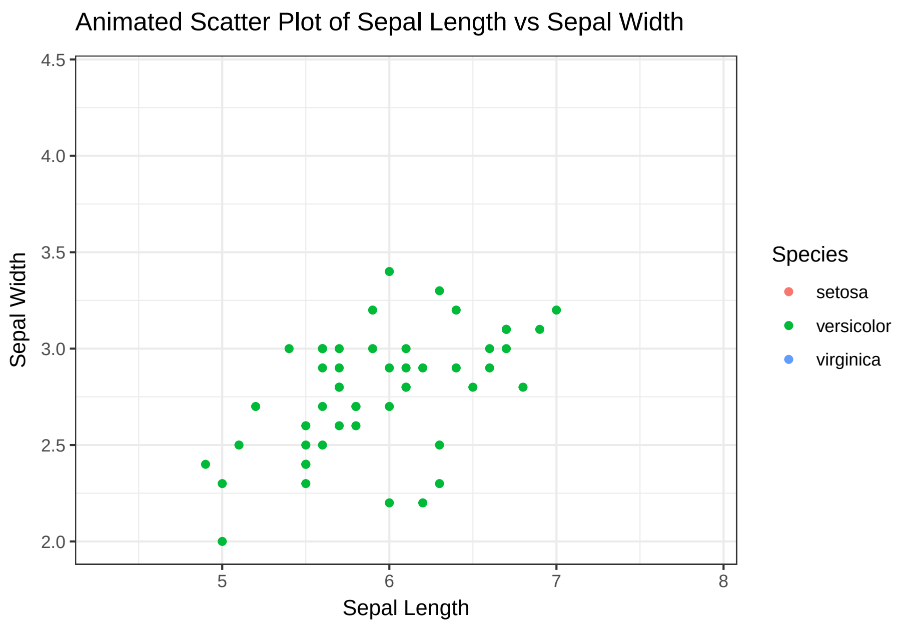

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

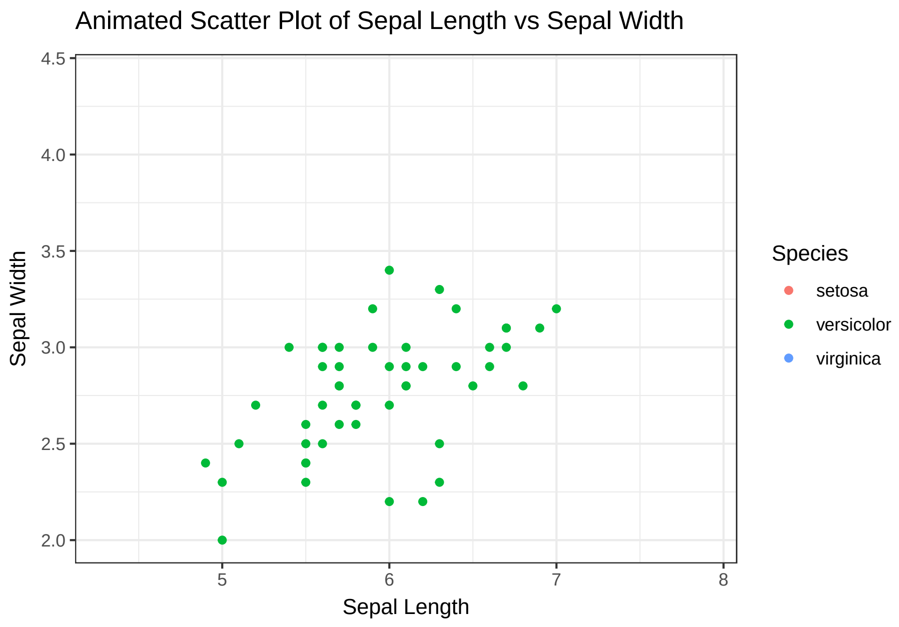

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

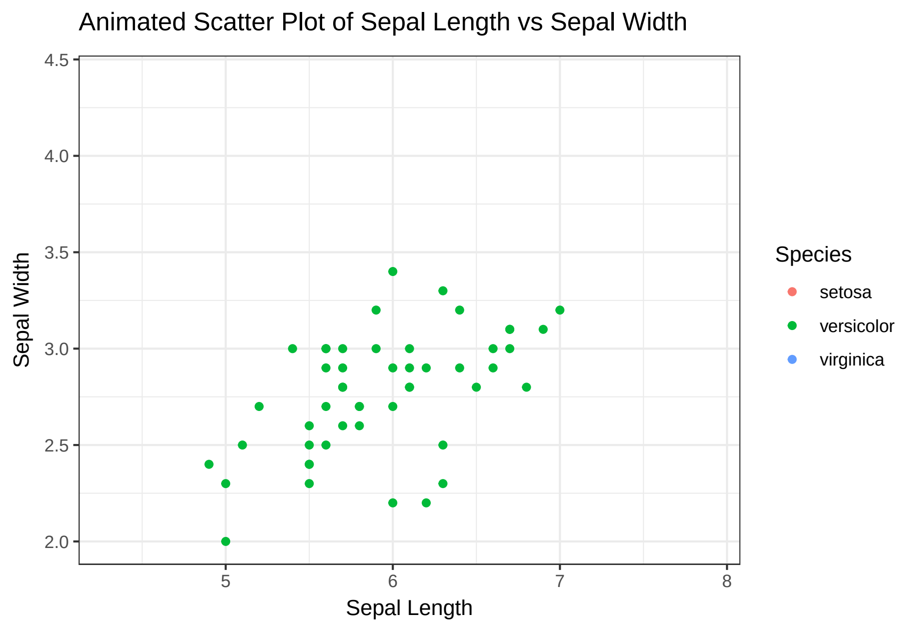

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

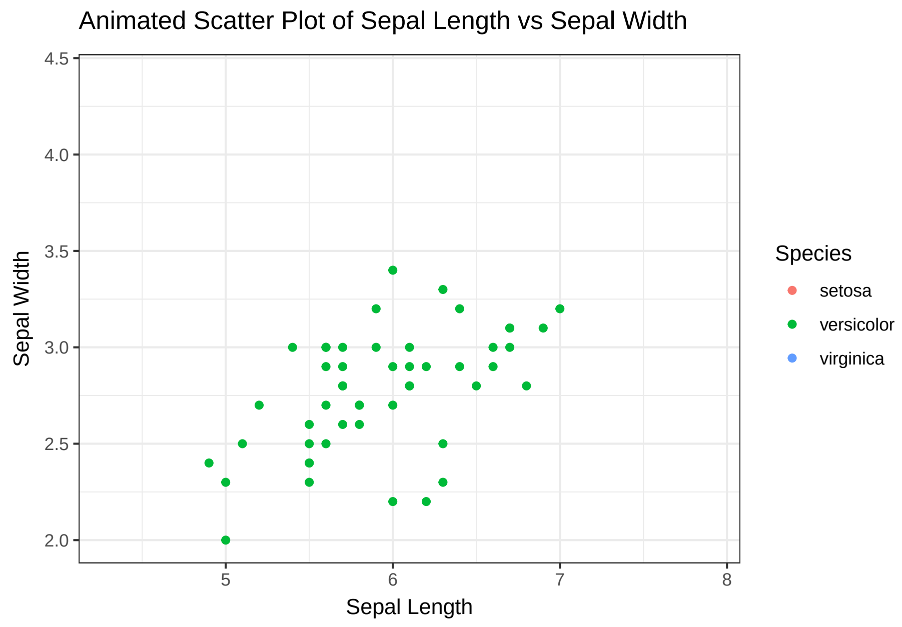

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
Species
setosa
versicolor
virginica
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8

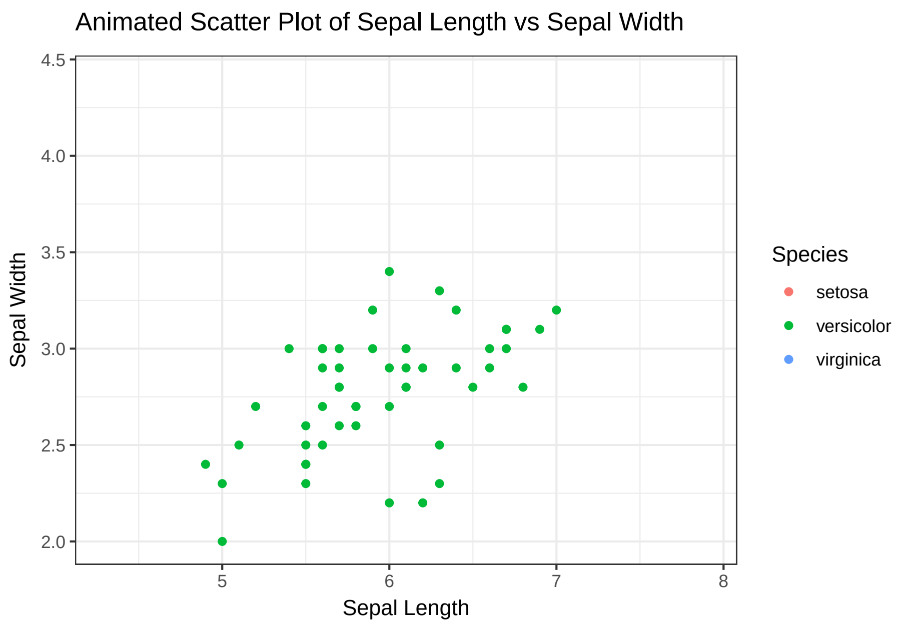

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
Species
setosa
versicolor
virginica
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8

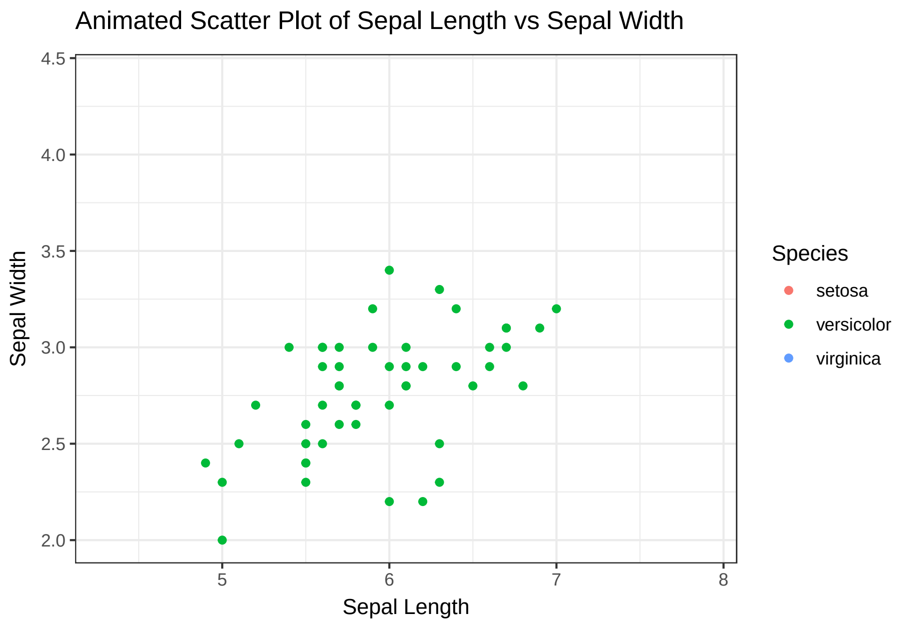

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

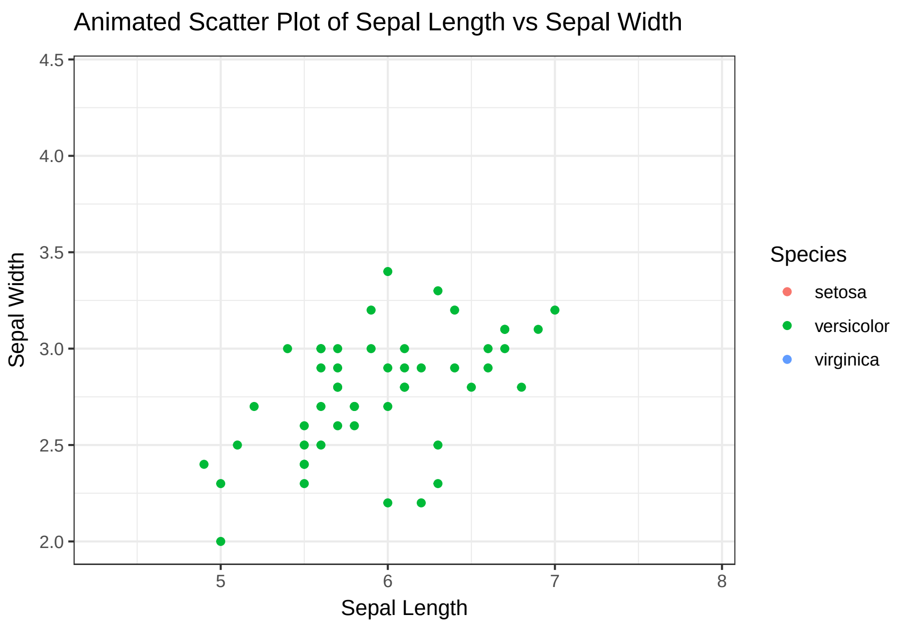

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

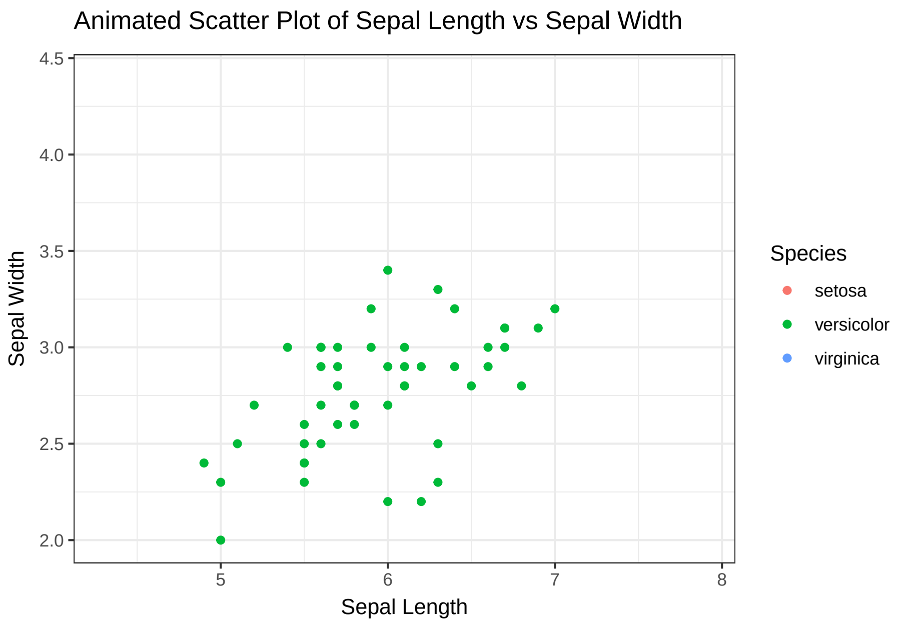

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

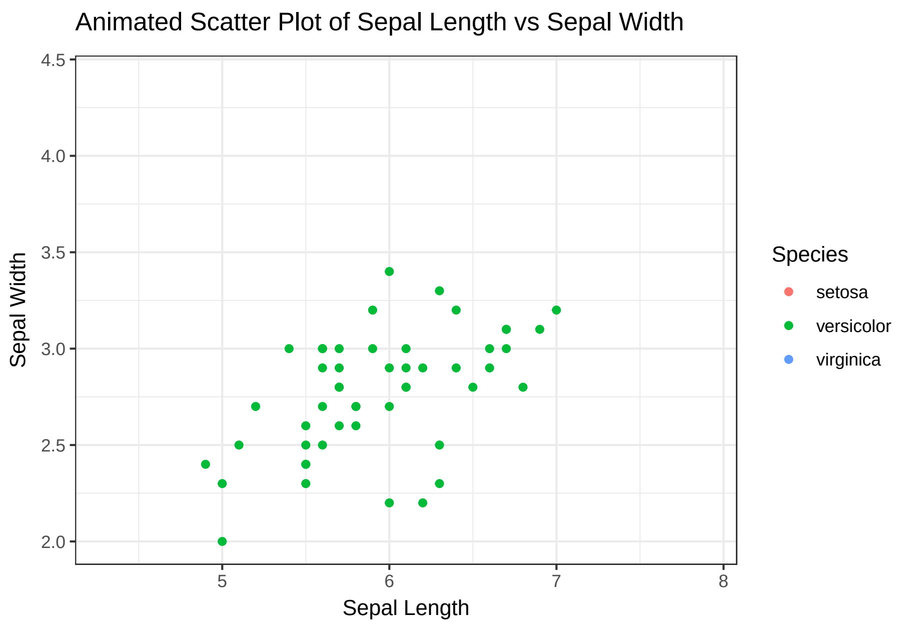

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

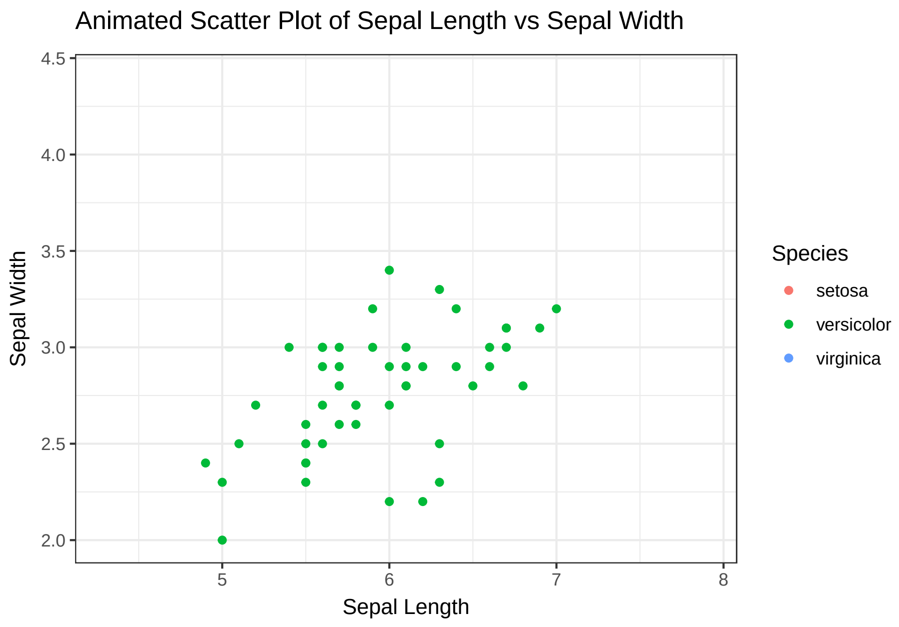

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

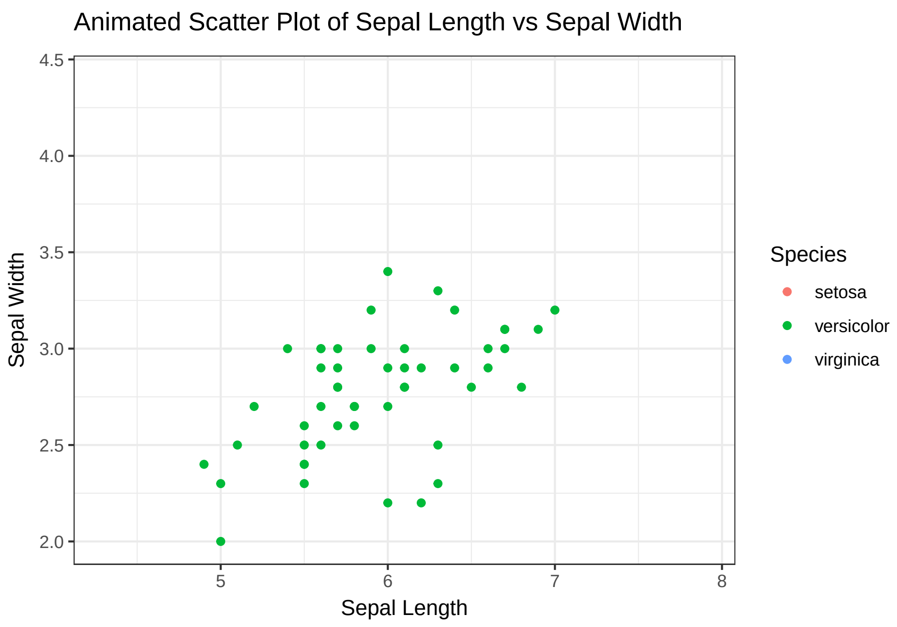

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

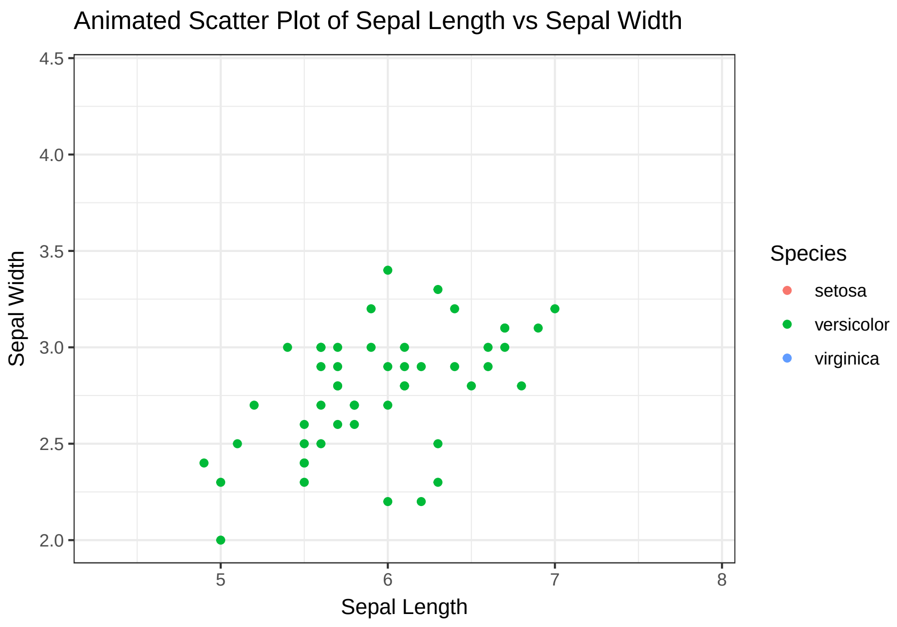

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
Species
setosa
versicolor
virginica
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8

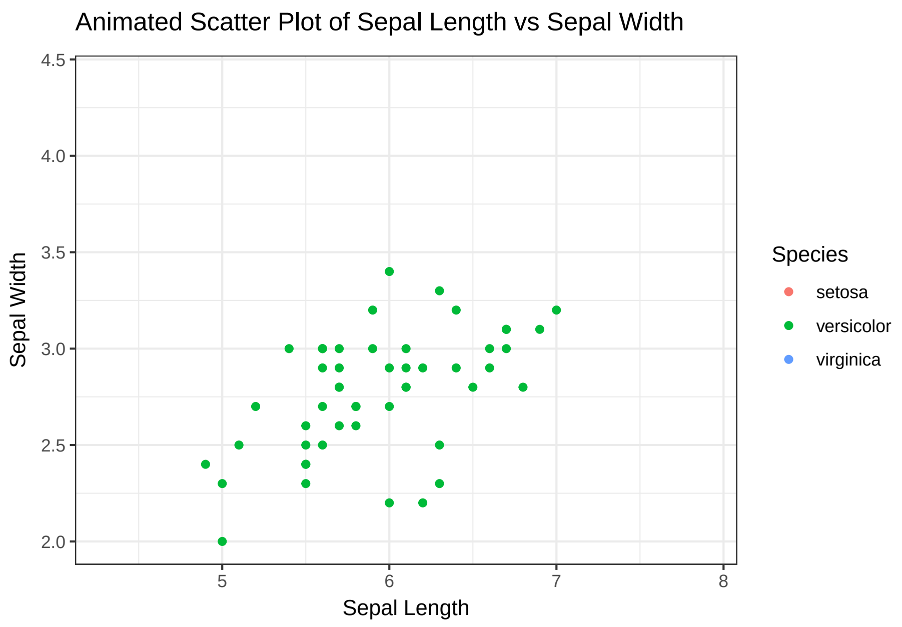

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
Species
setosa
versicolor
virginica
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8

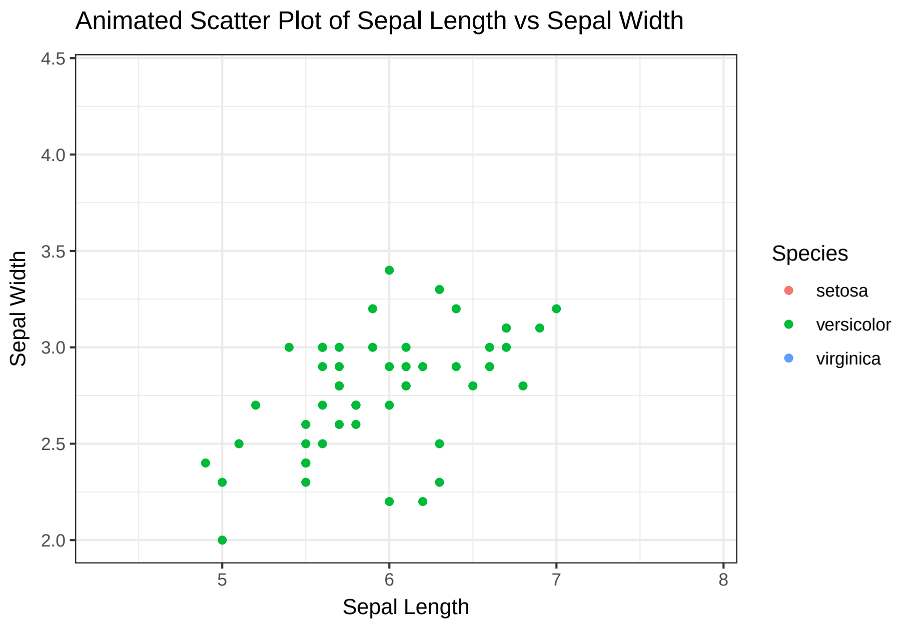

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

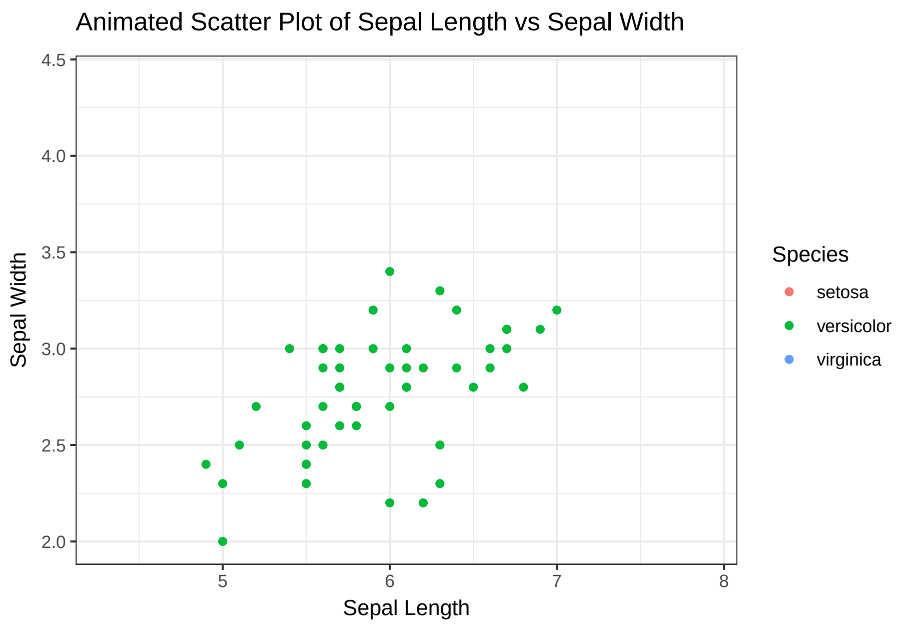

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

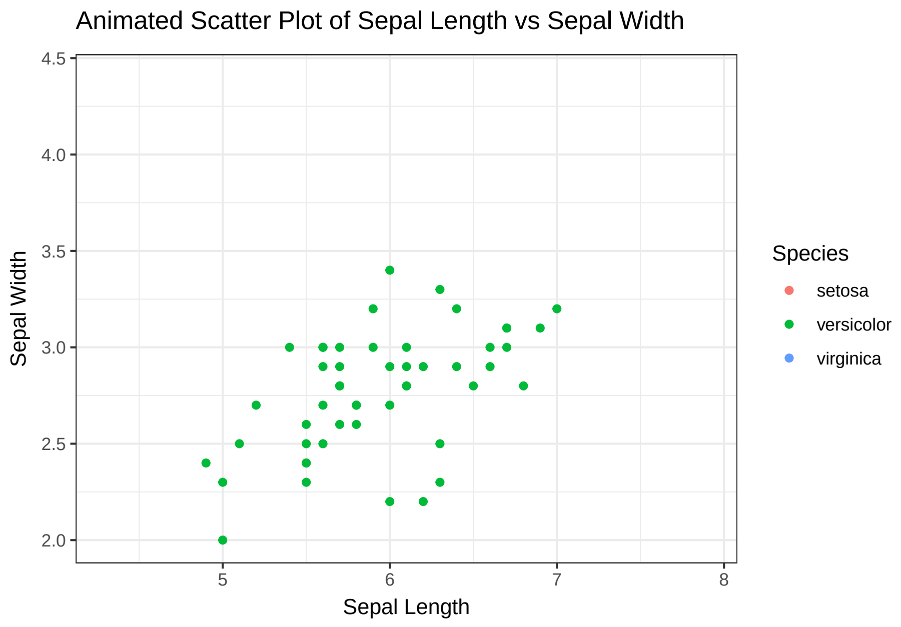

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

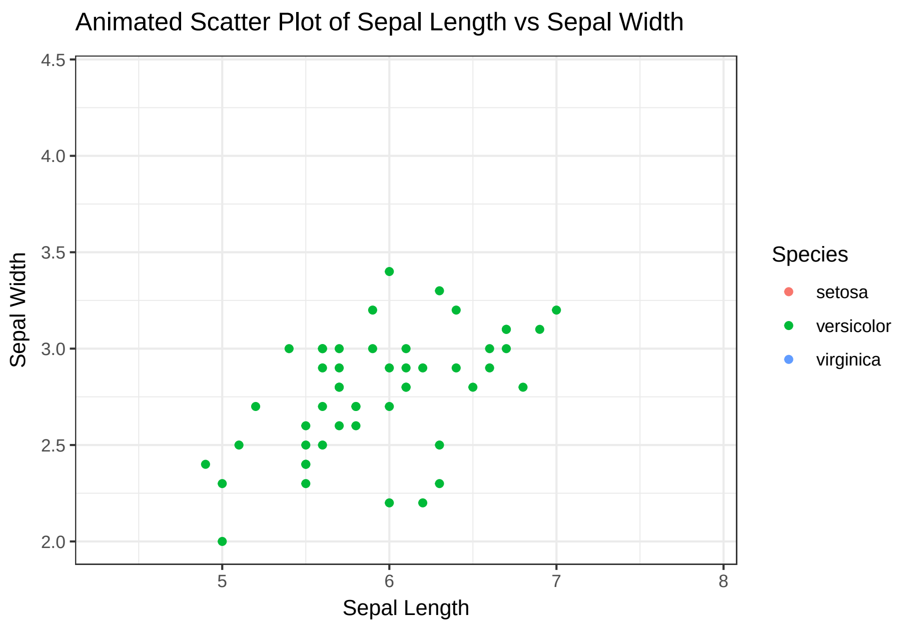

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

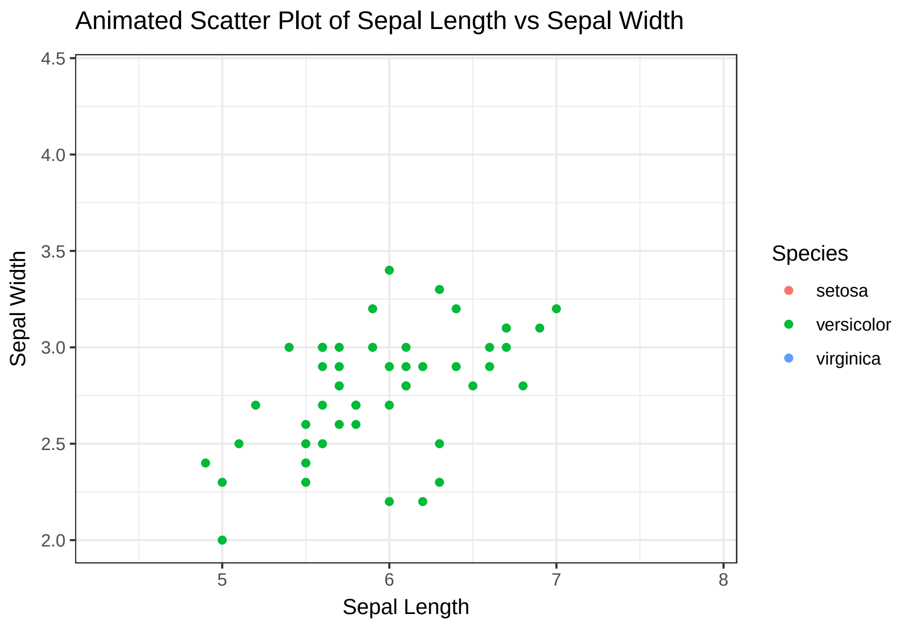

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

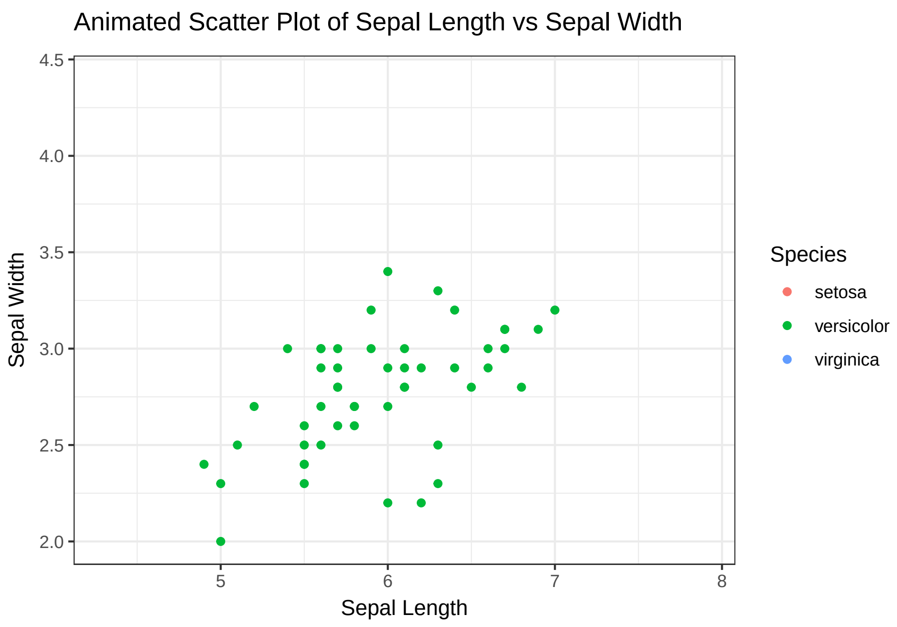

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

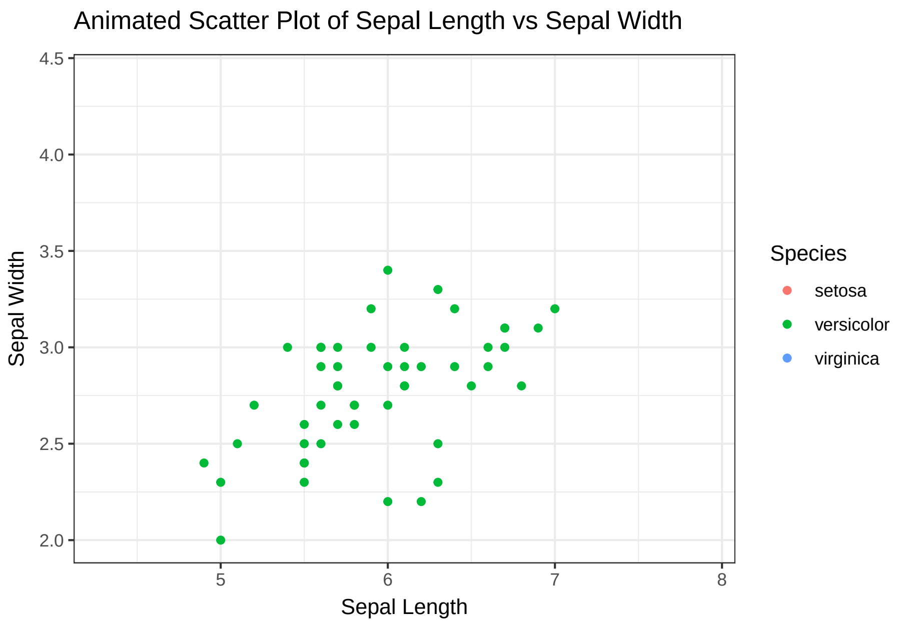

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

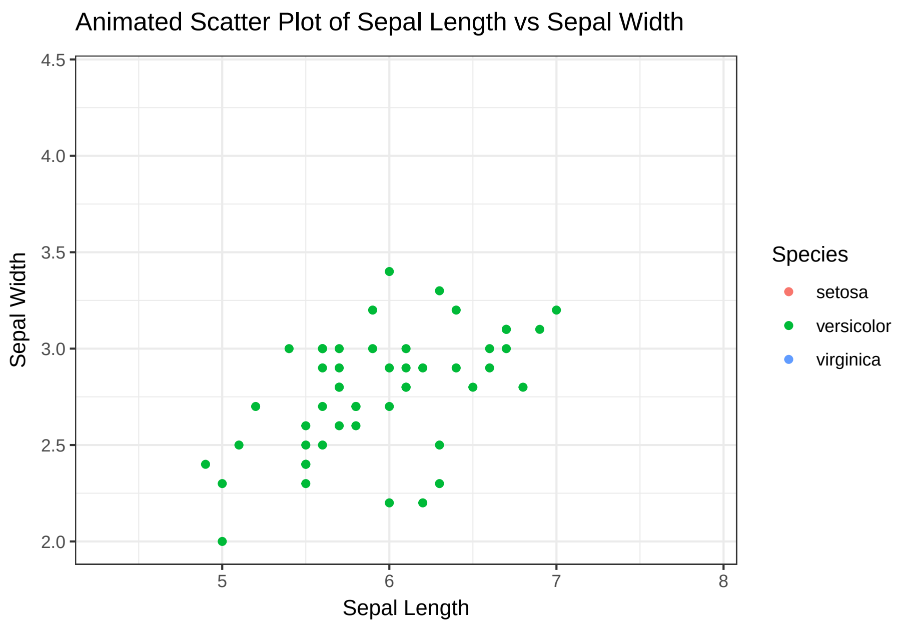

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

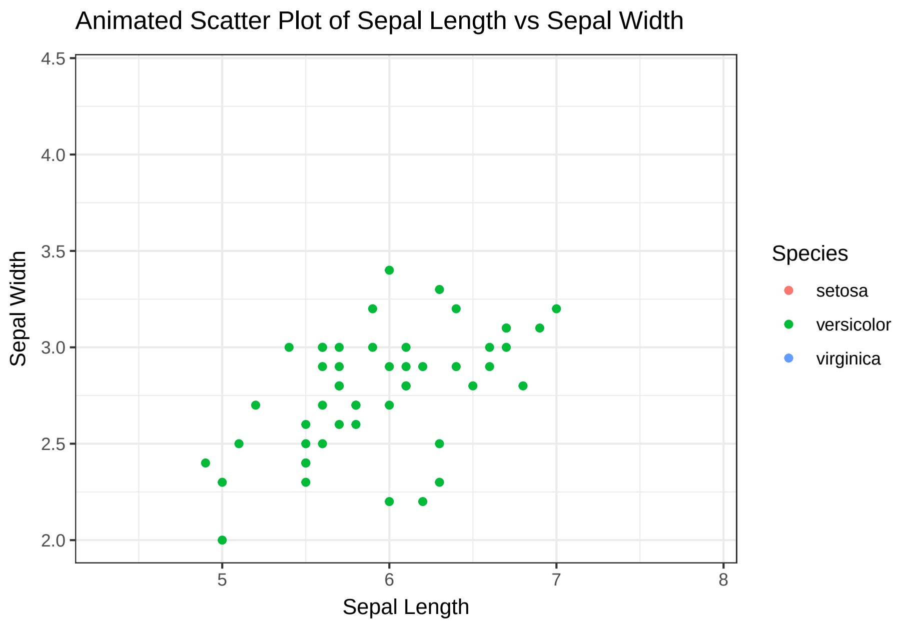

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

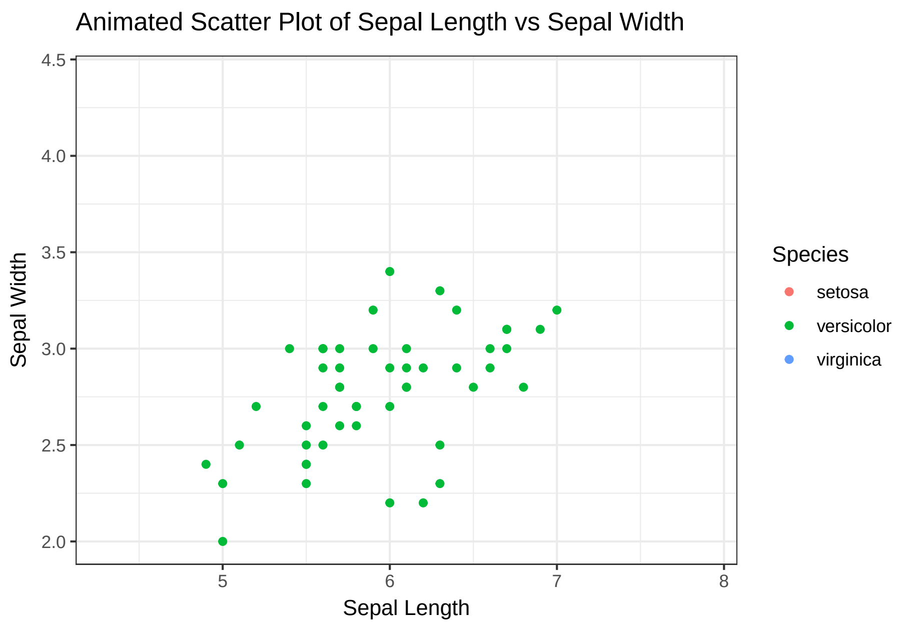

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

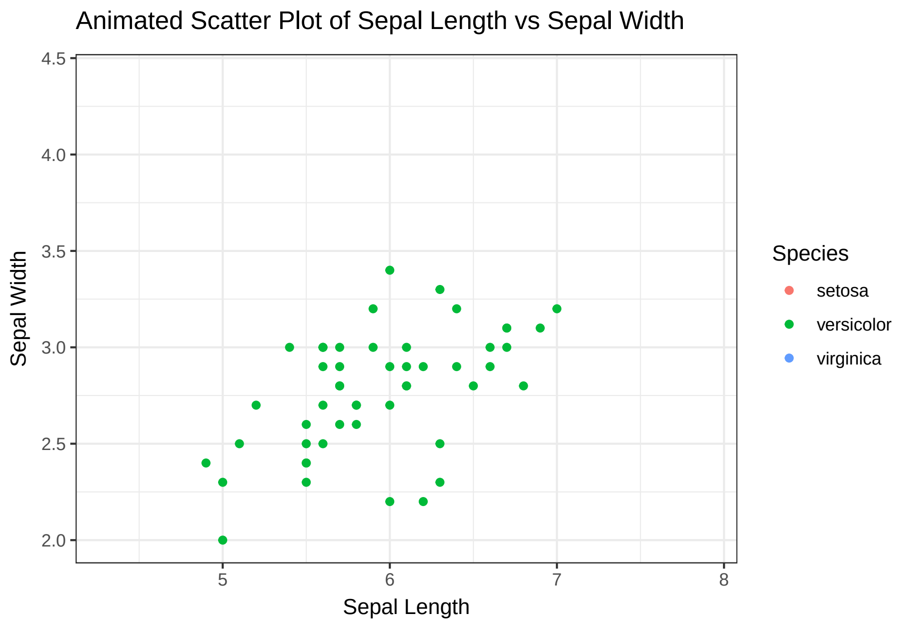

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

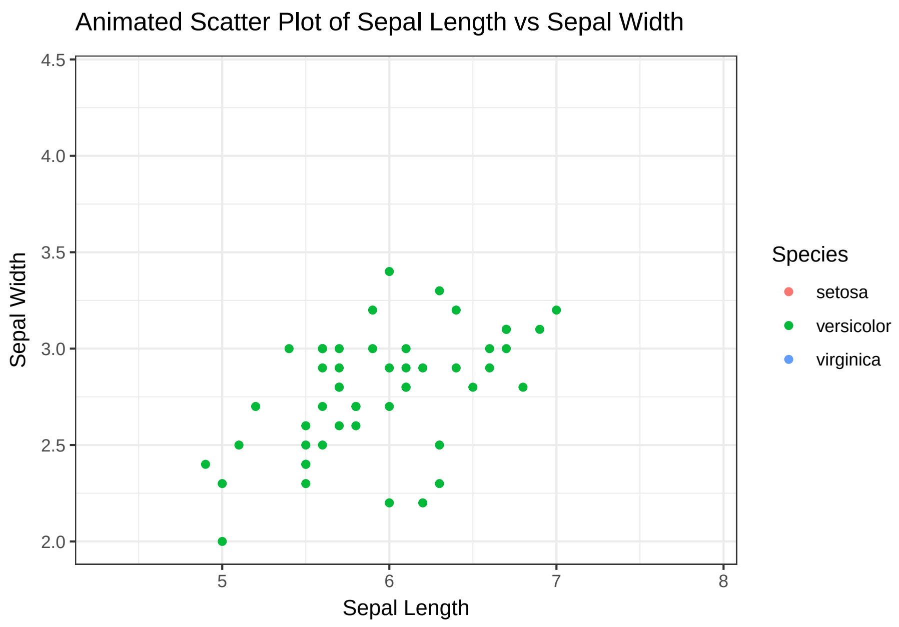

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

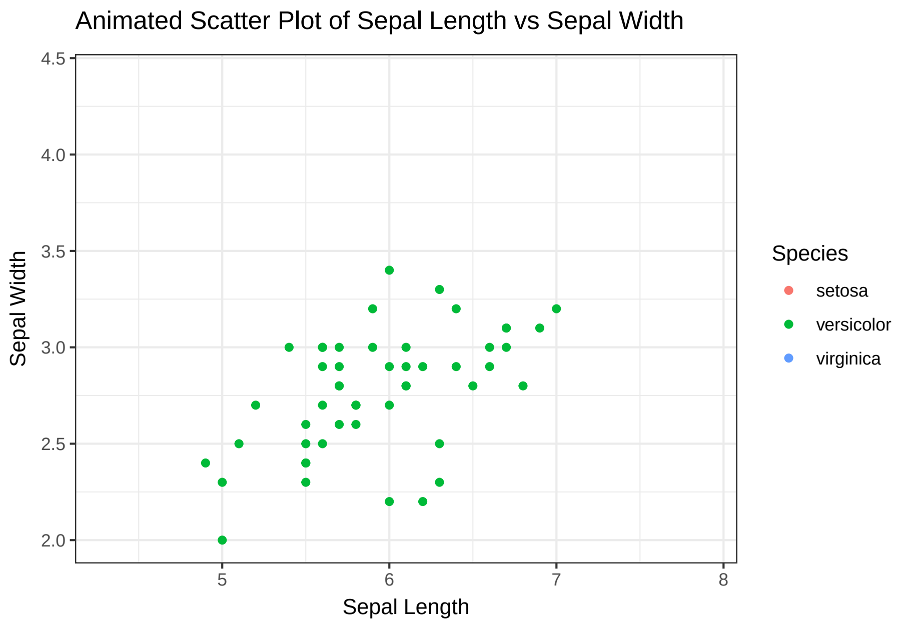

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

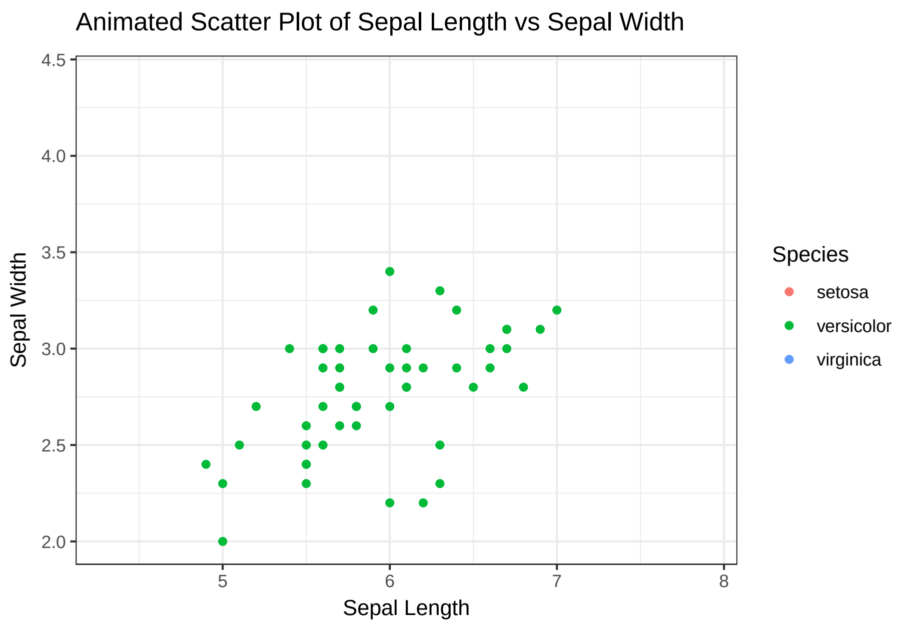

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

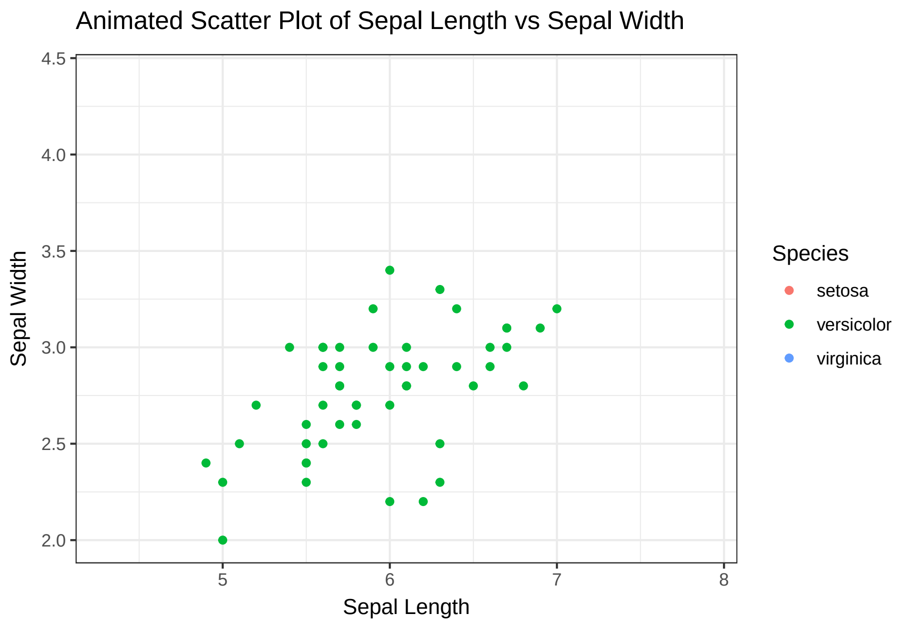

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

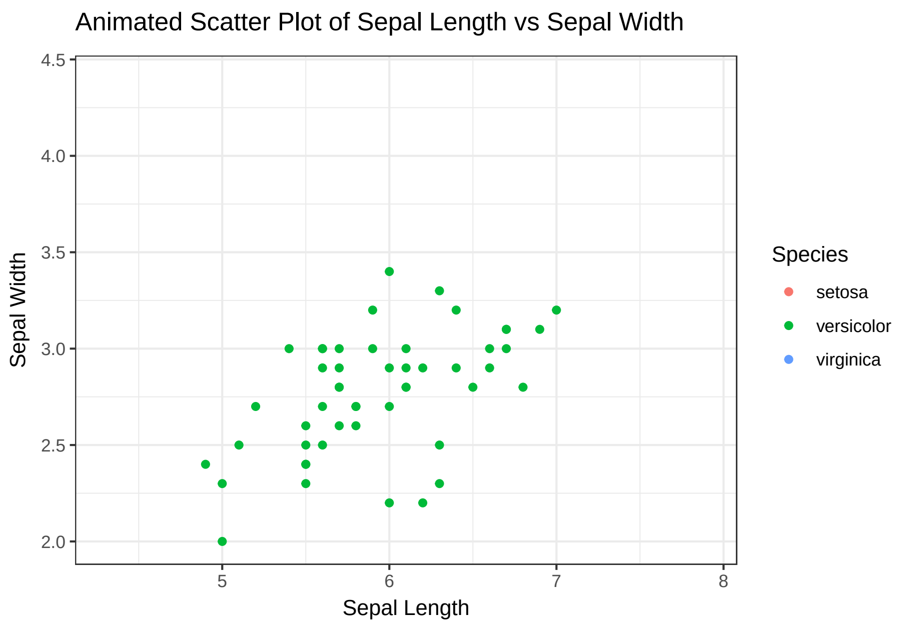

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

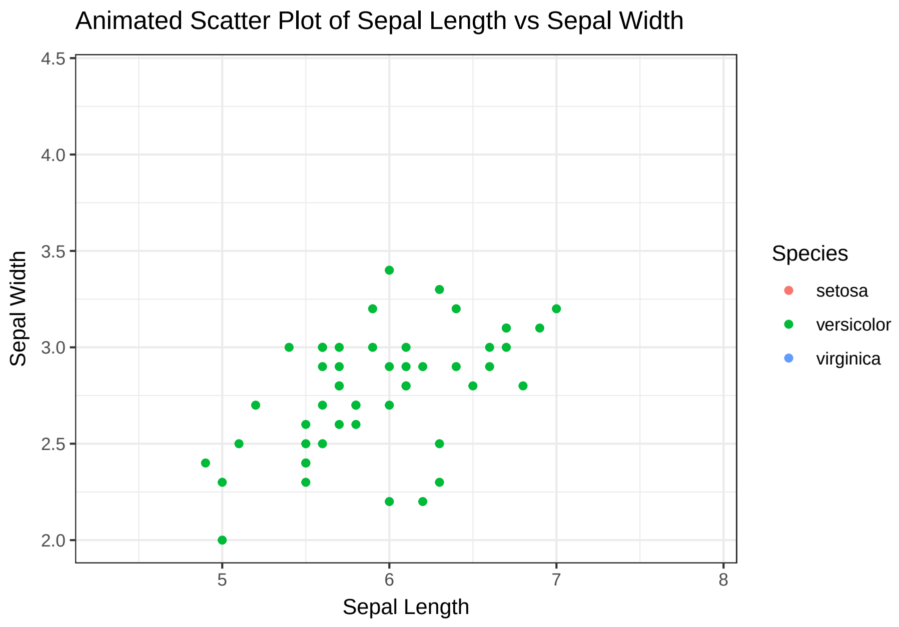

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

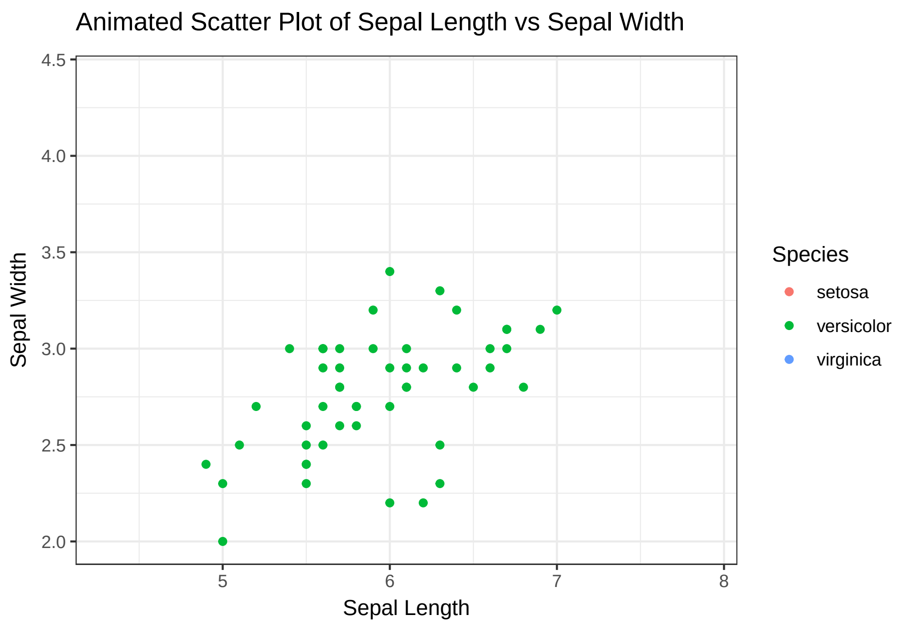

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

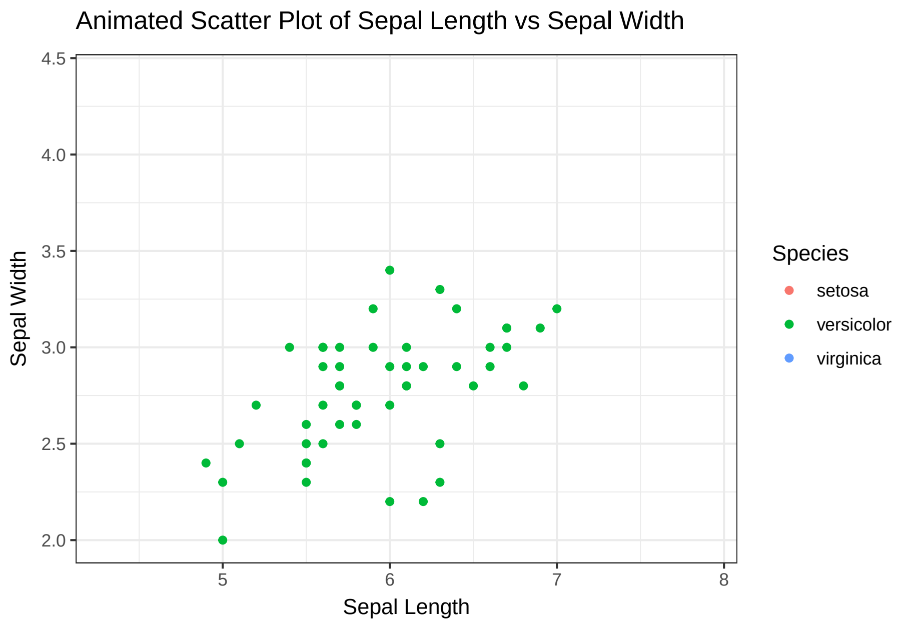

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

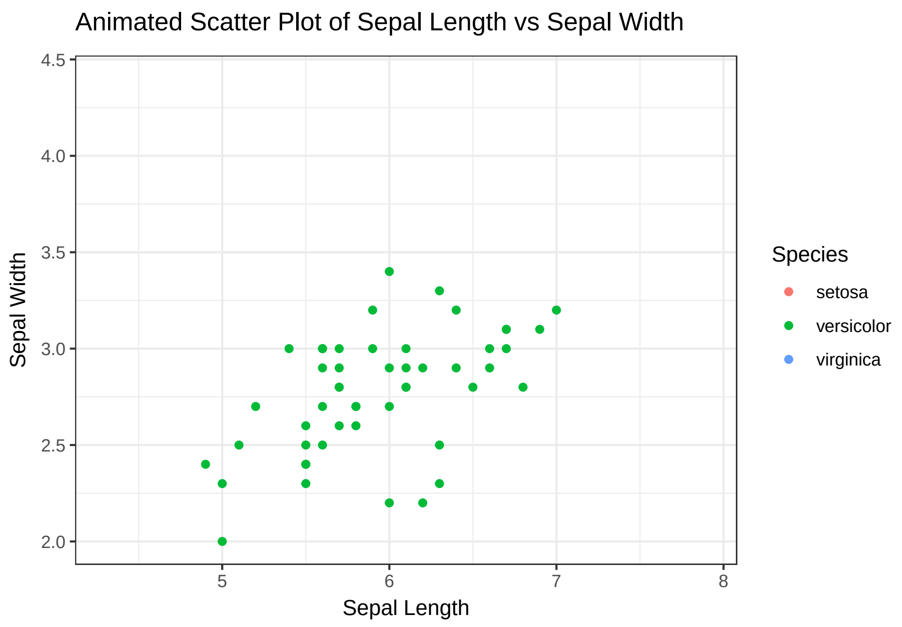

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

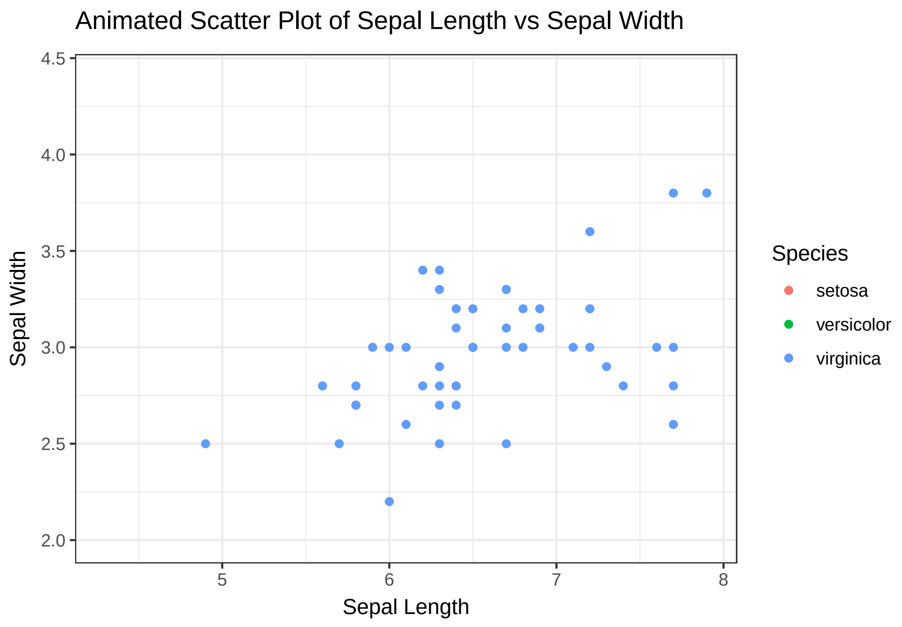

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

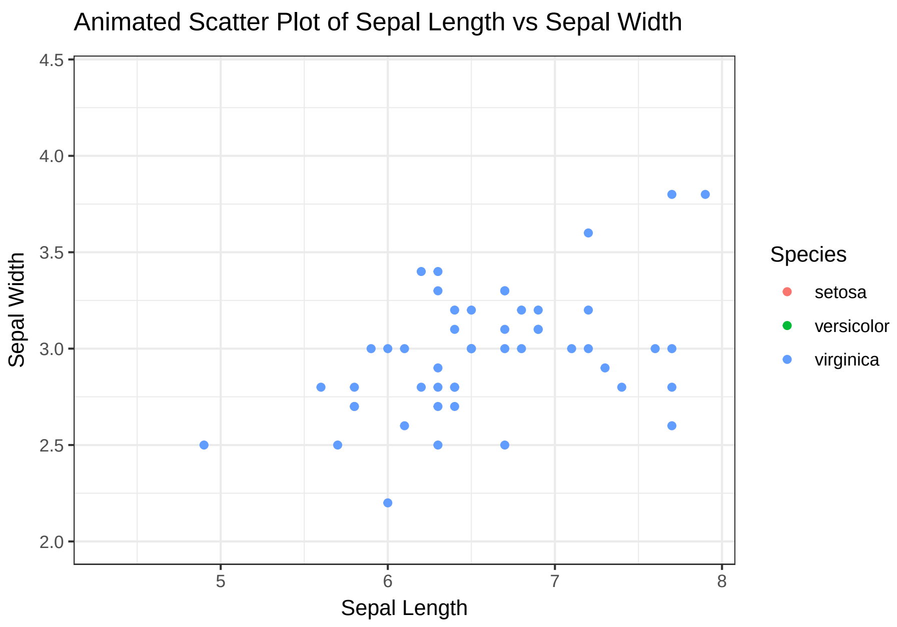

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

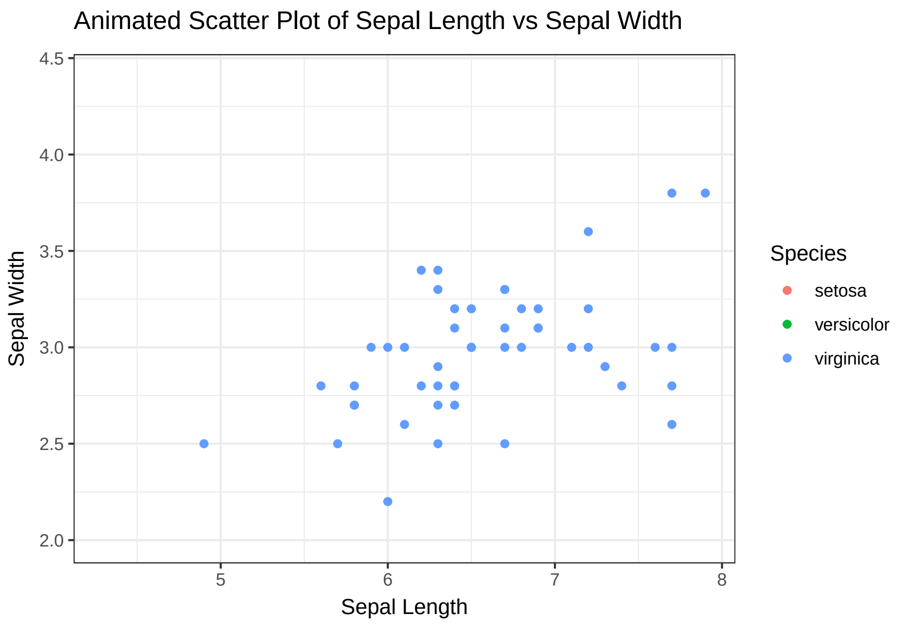

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

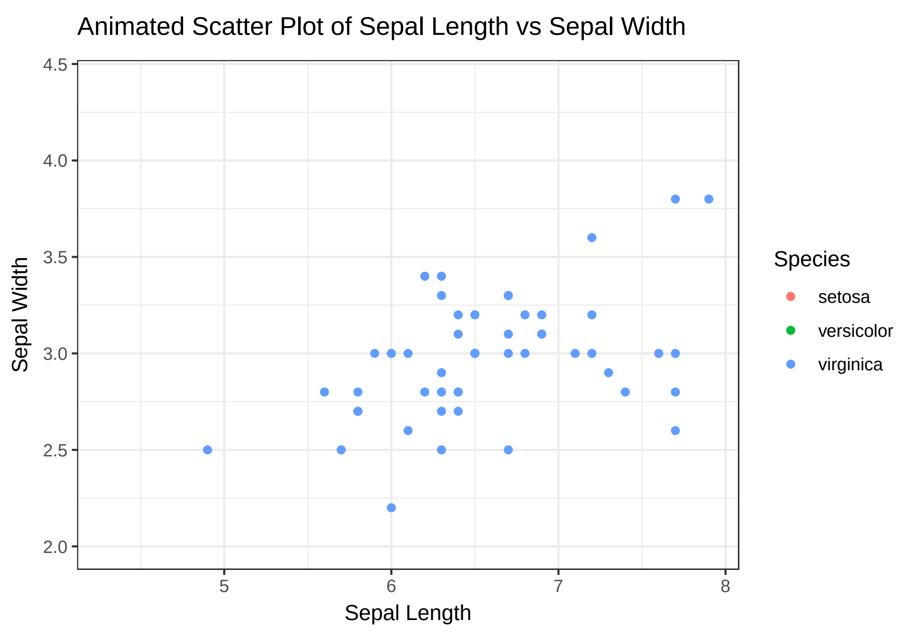

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

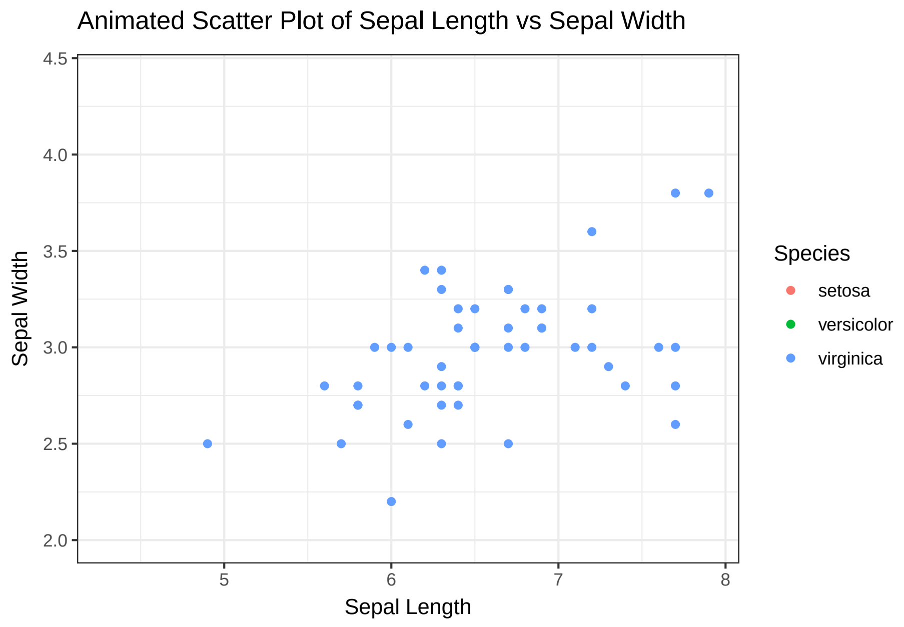

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

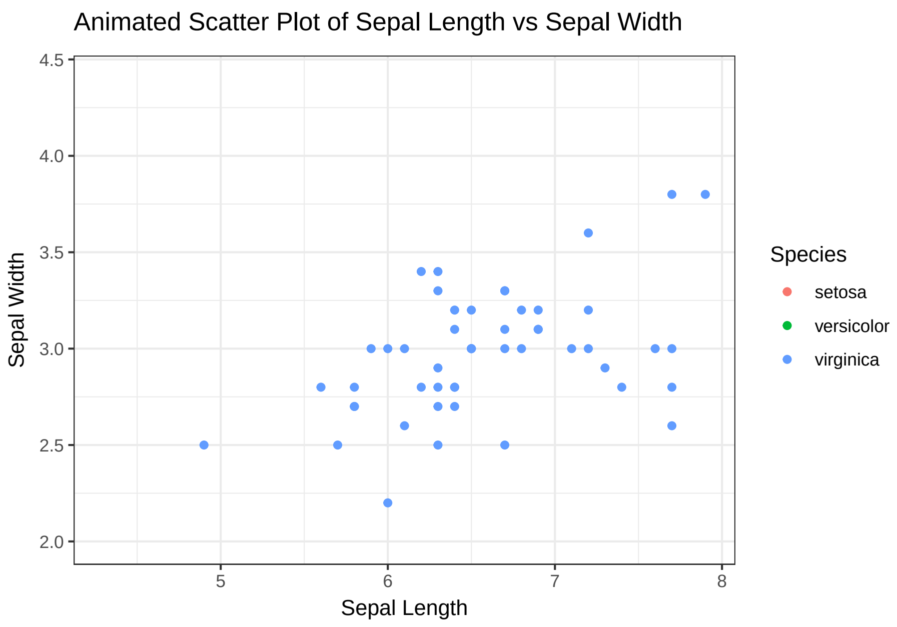

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

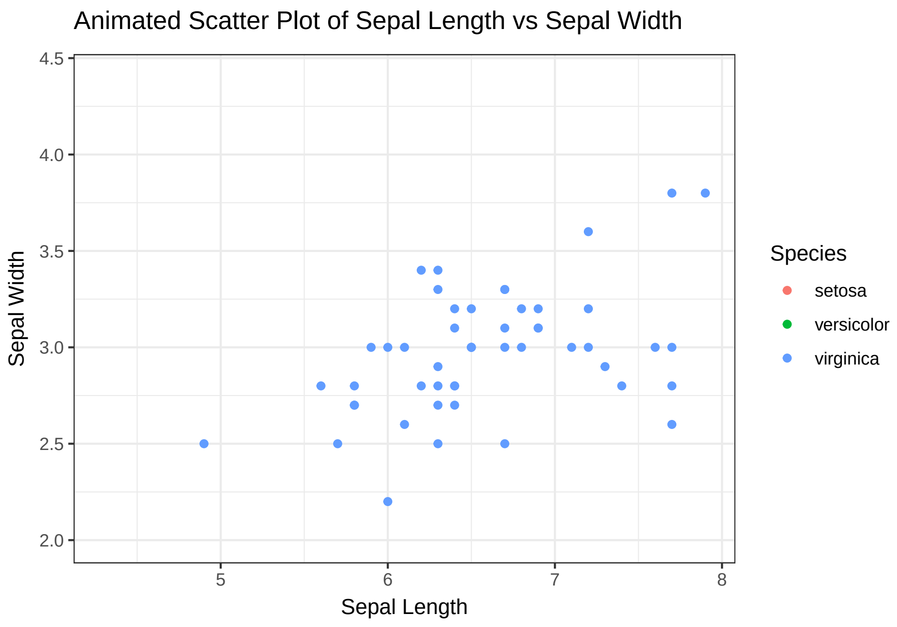

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

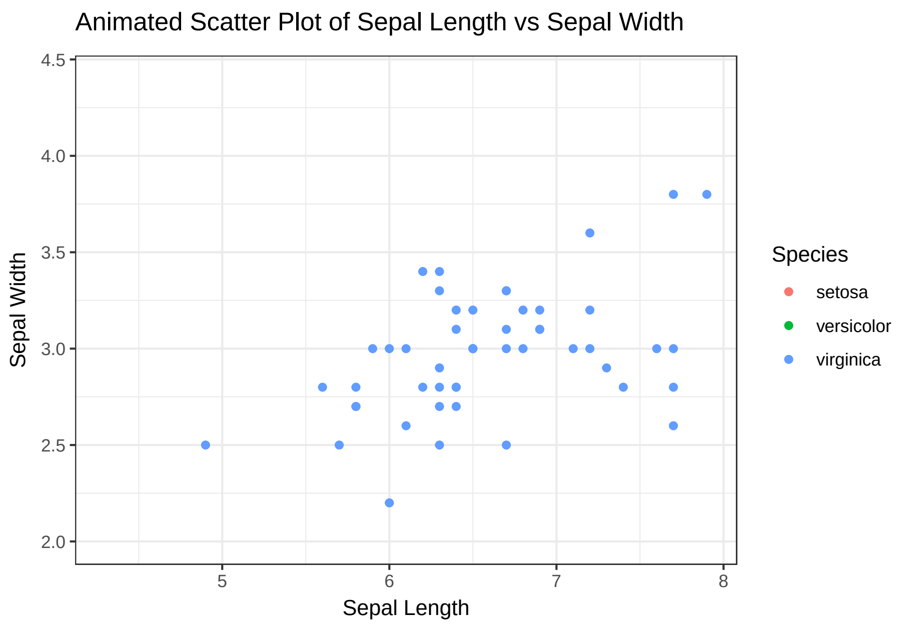

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

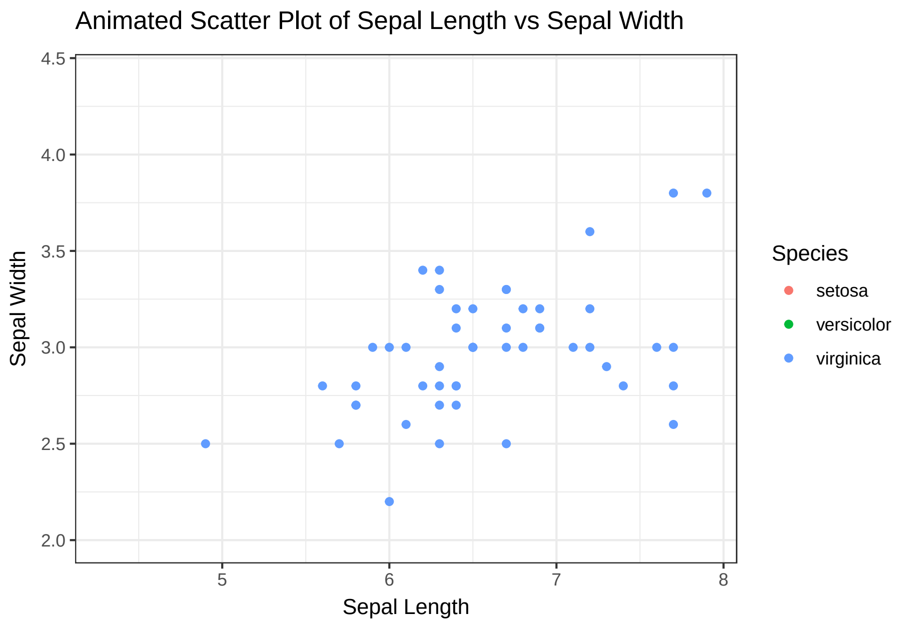

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

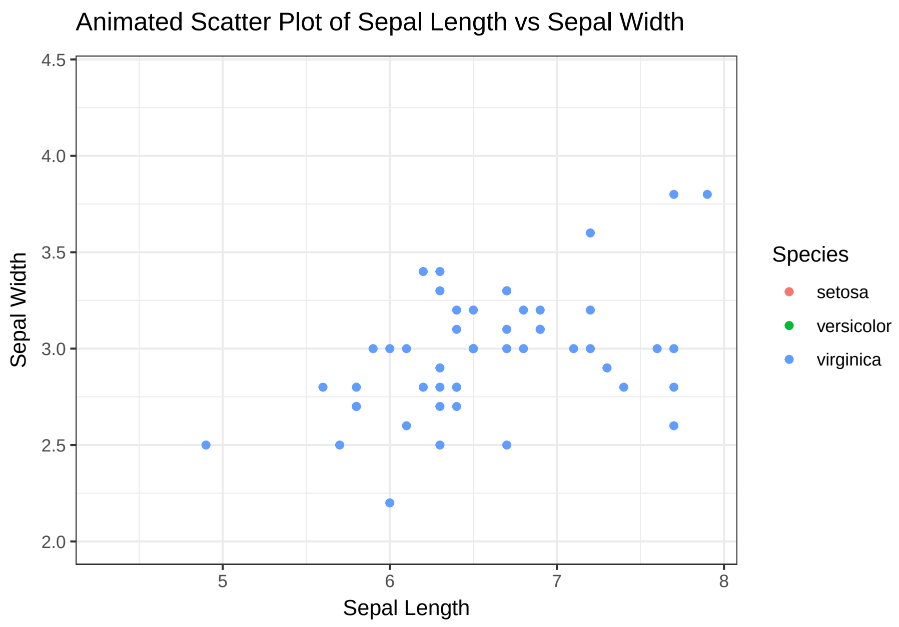

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

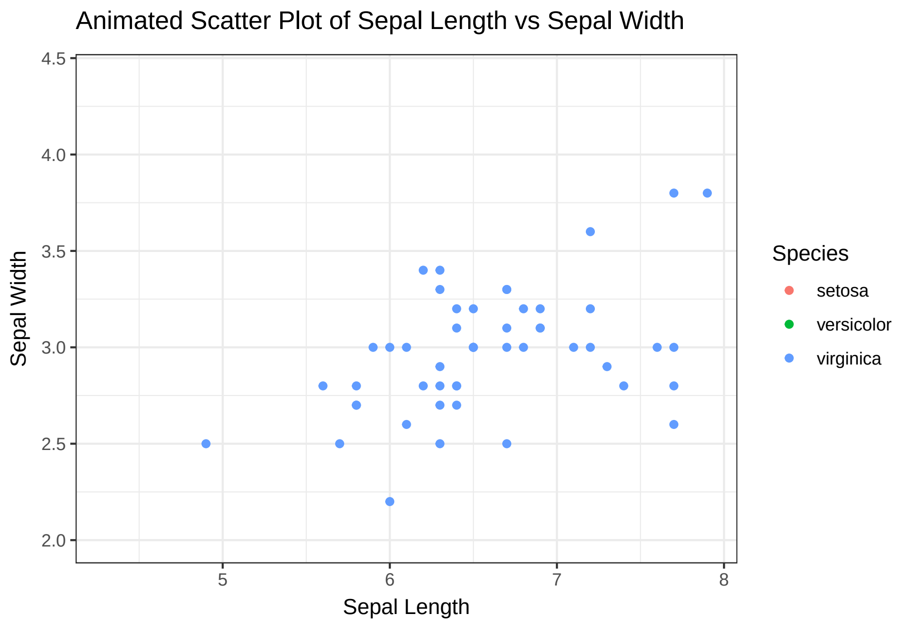

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

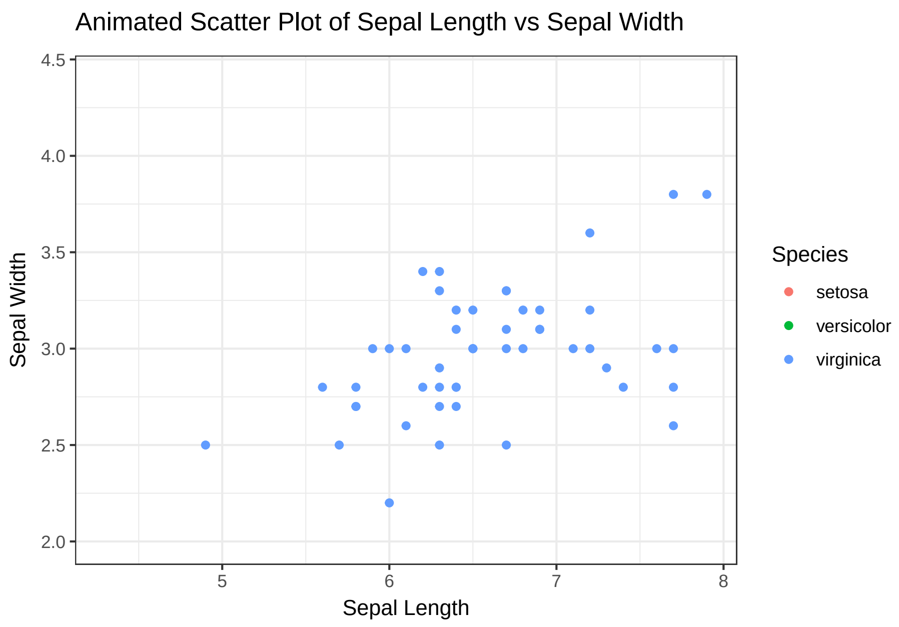

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

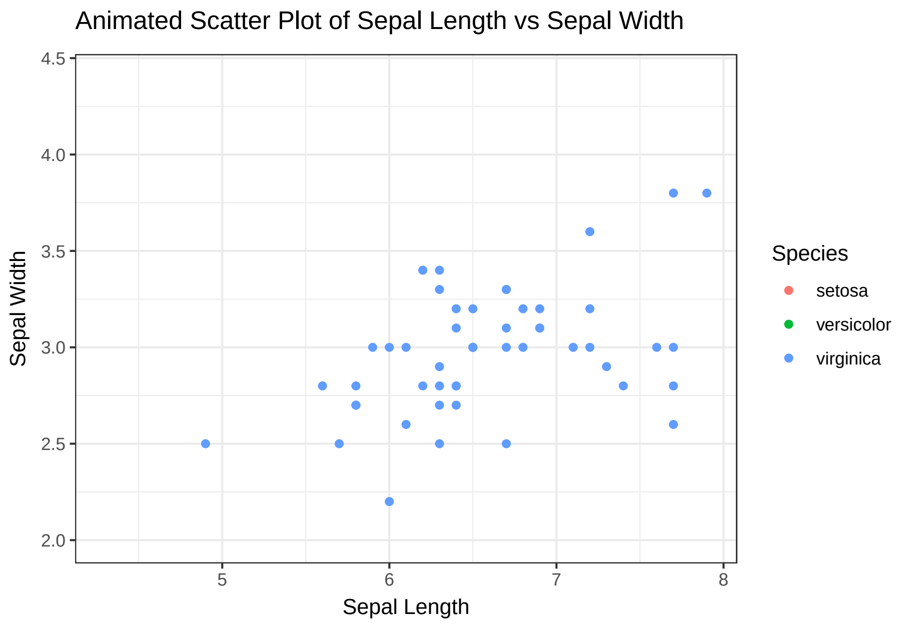

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

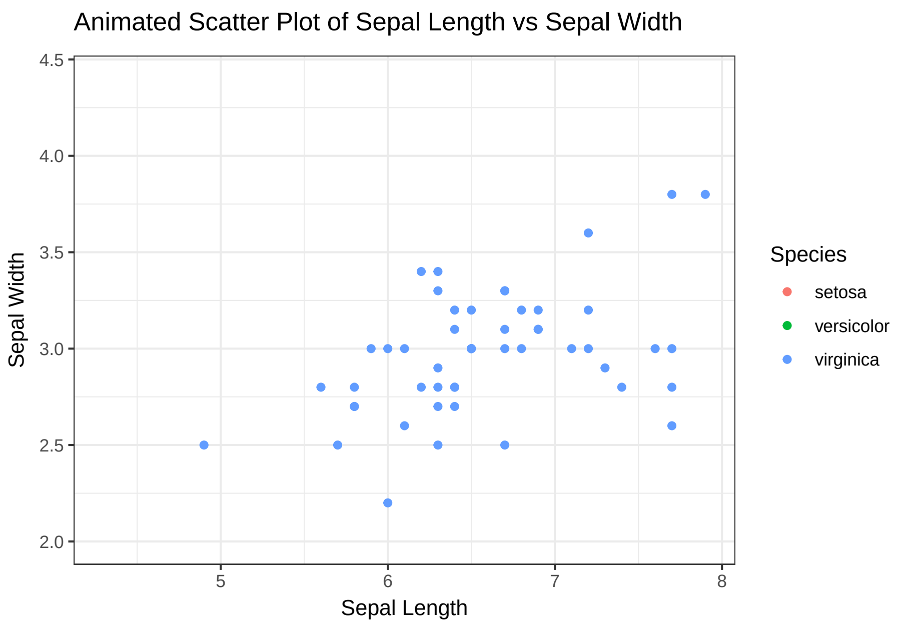

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

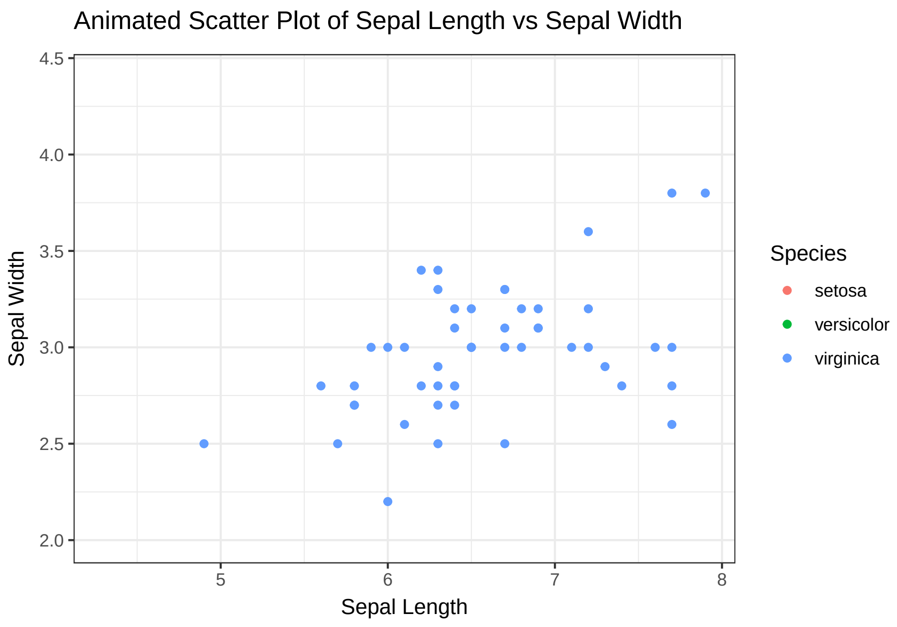

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

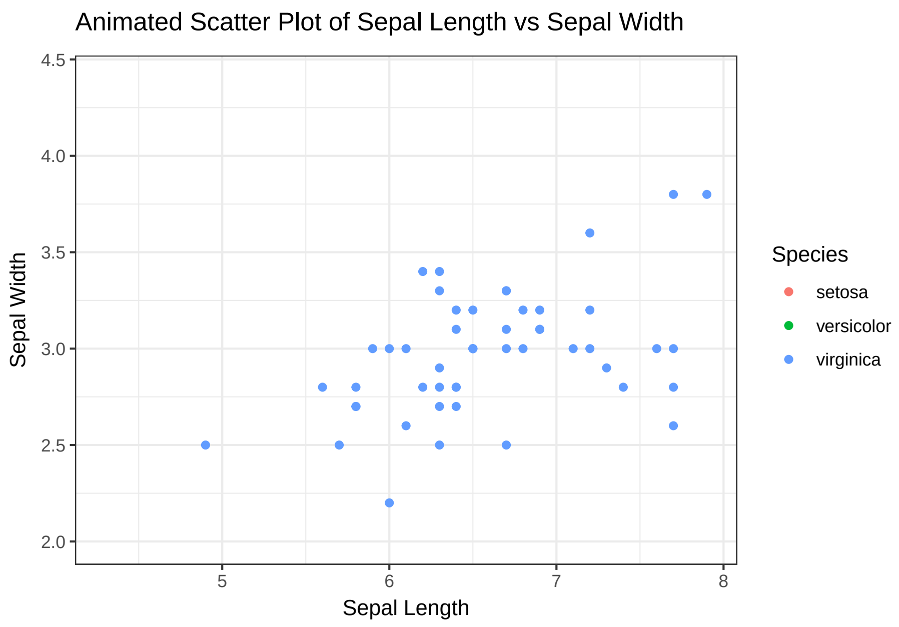

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

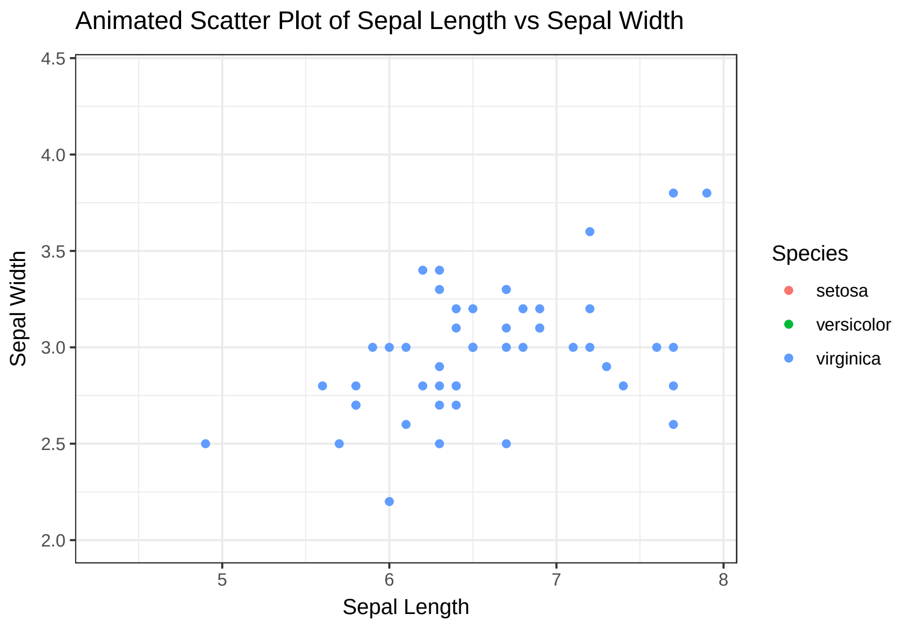

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

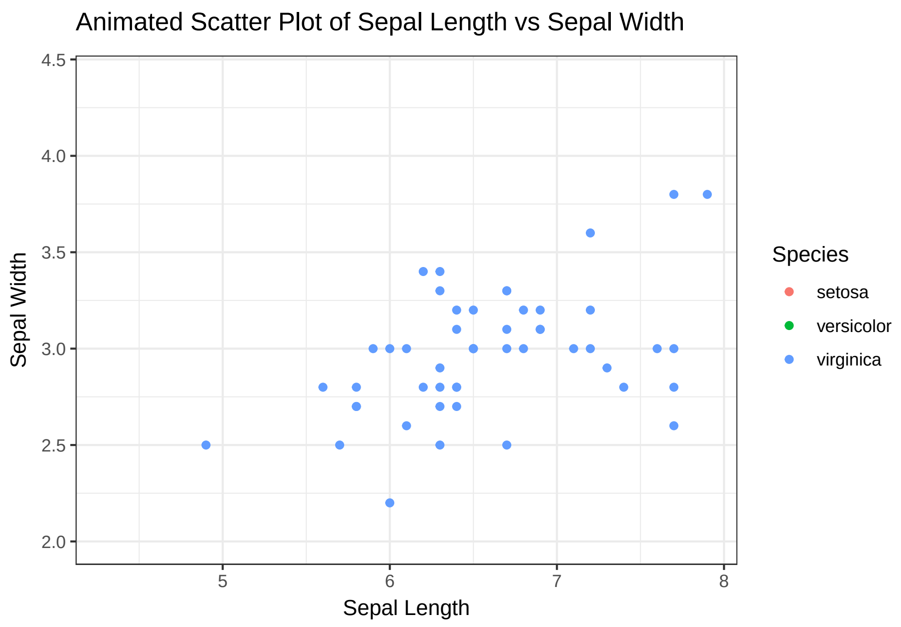

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

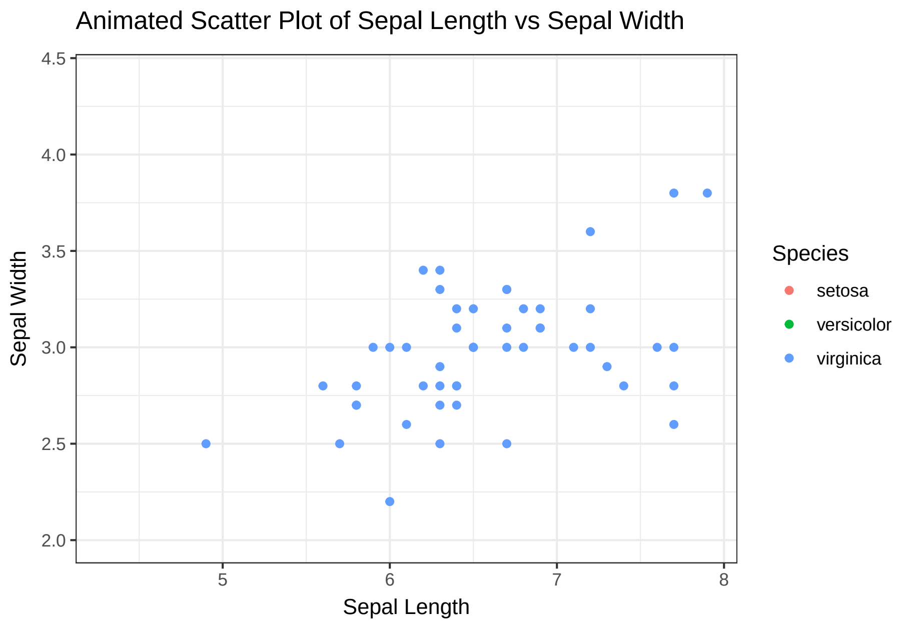

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

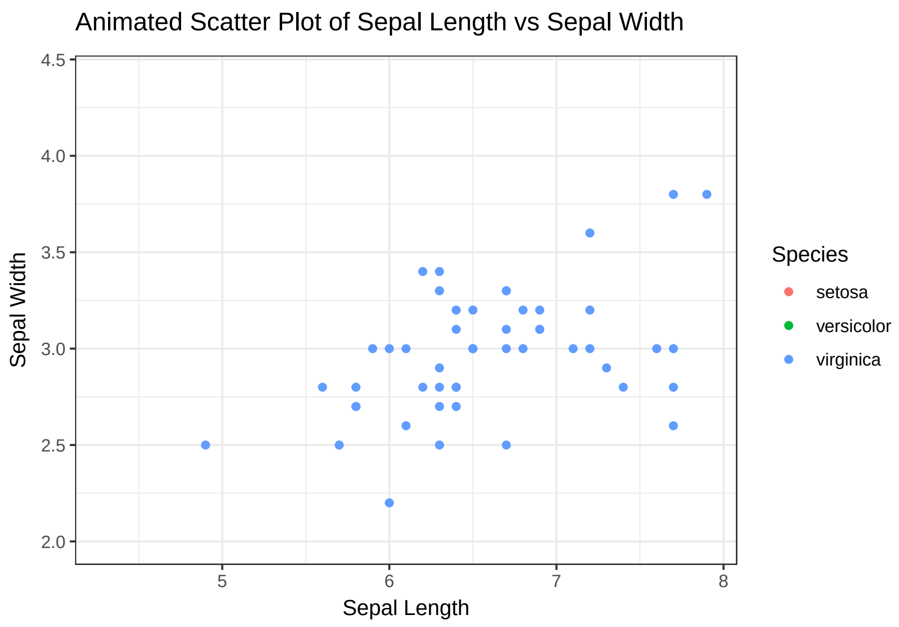

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

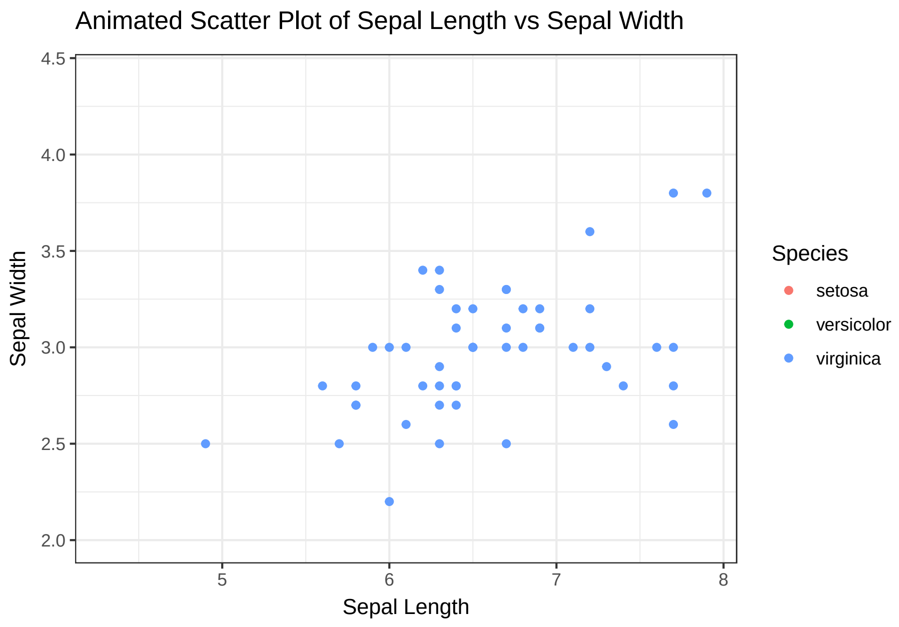

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

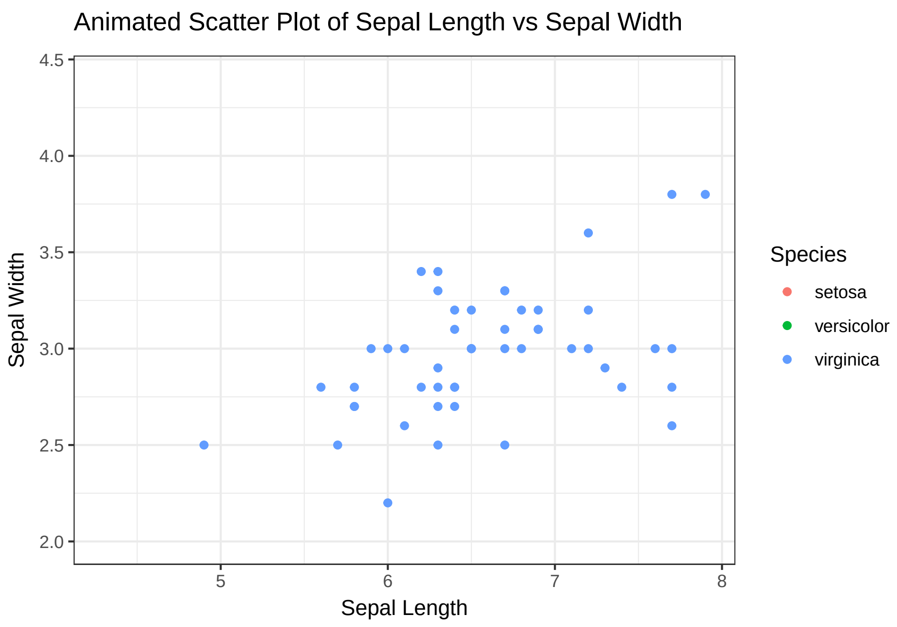

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

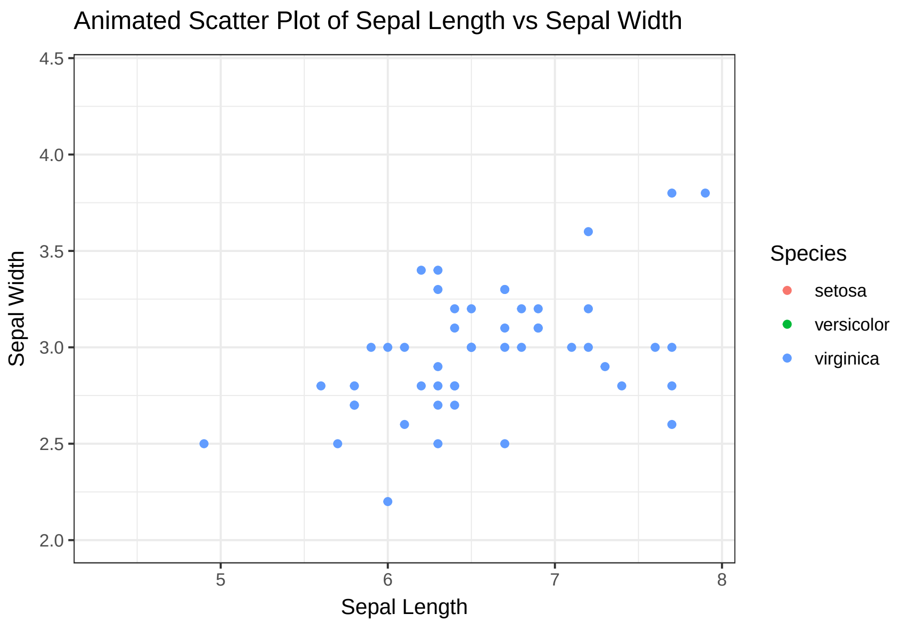

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

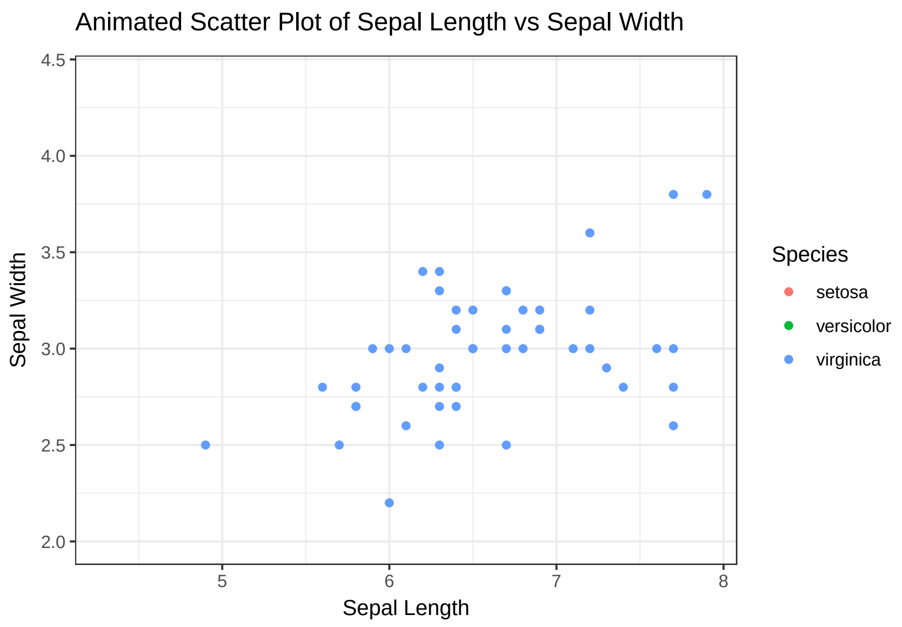

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

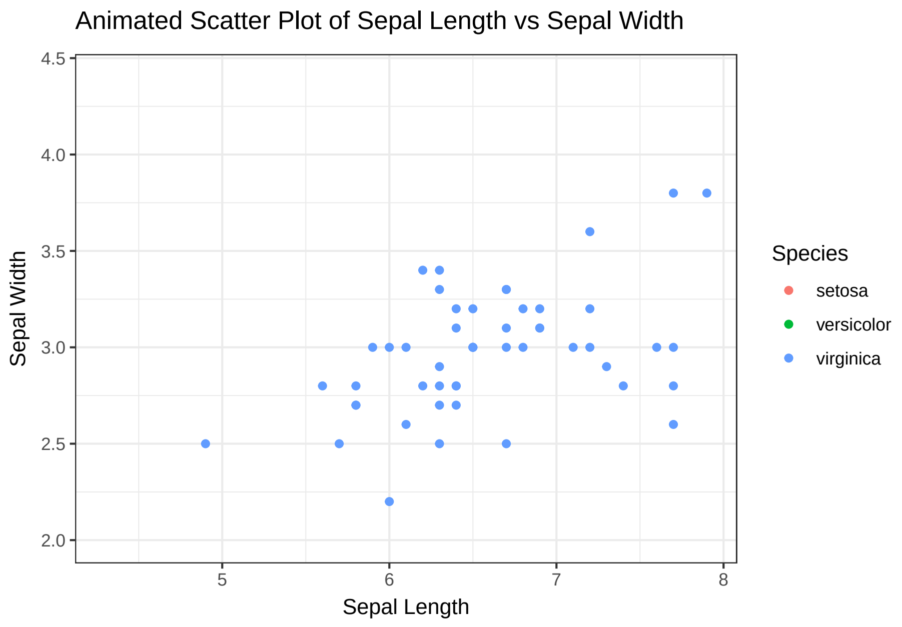

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

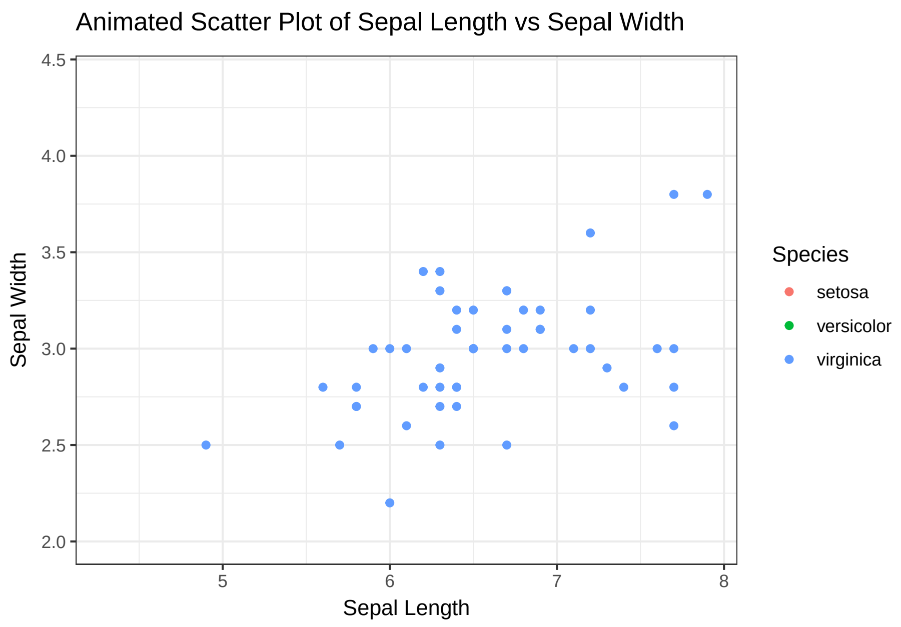

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Sepal Length
Species
setosa
versicolor
virginica

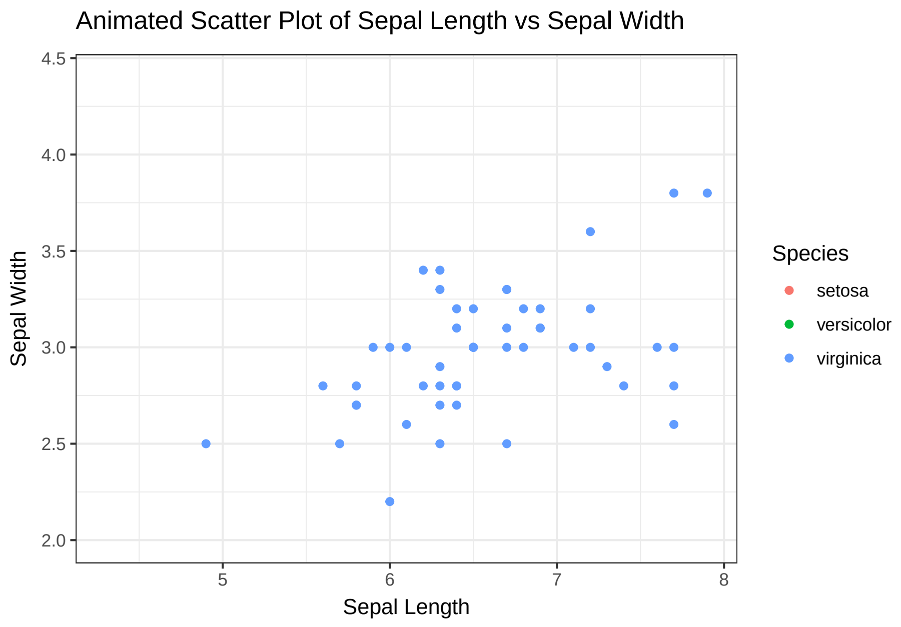

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

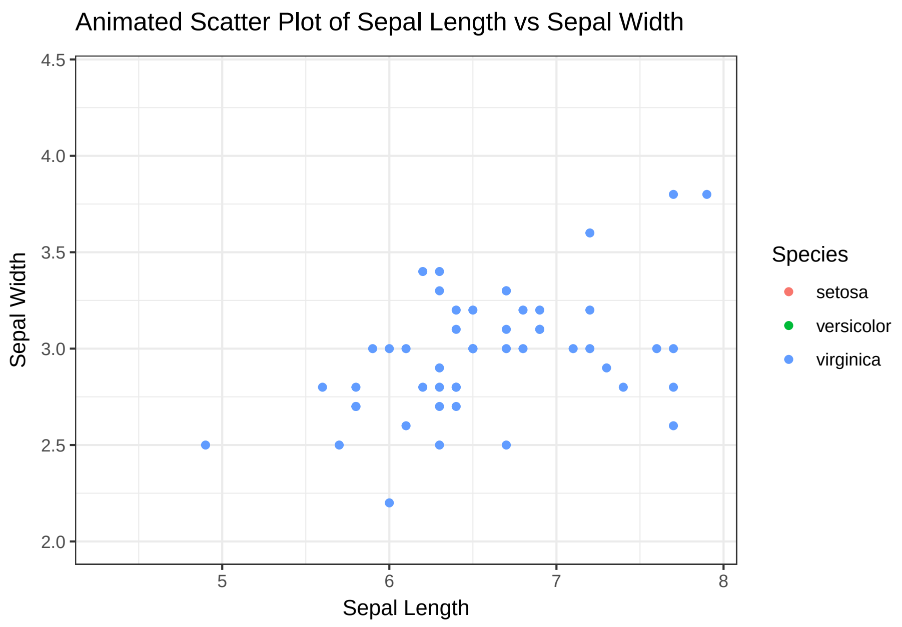

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

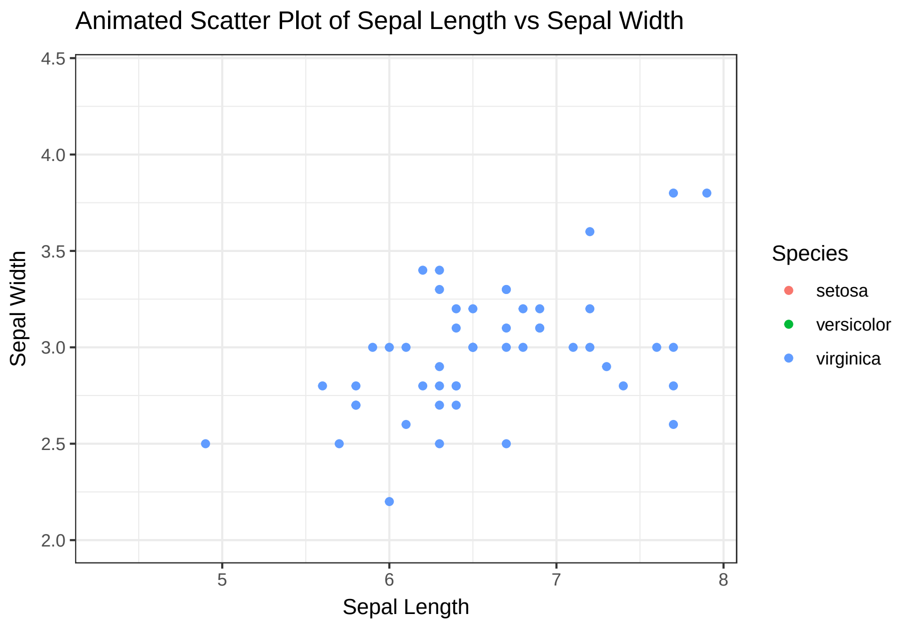

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

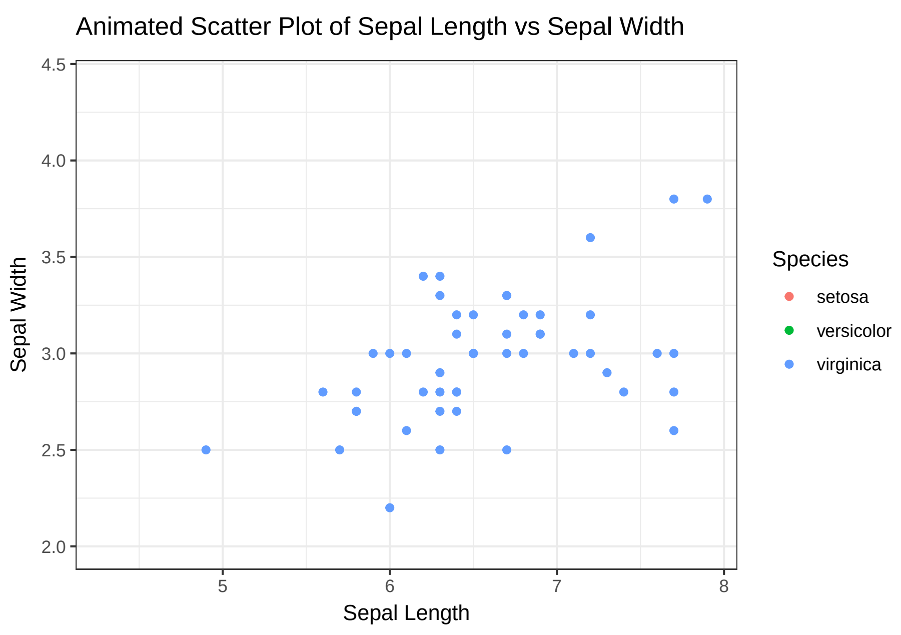

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

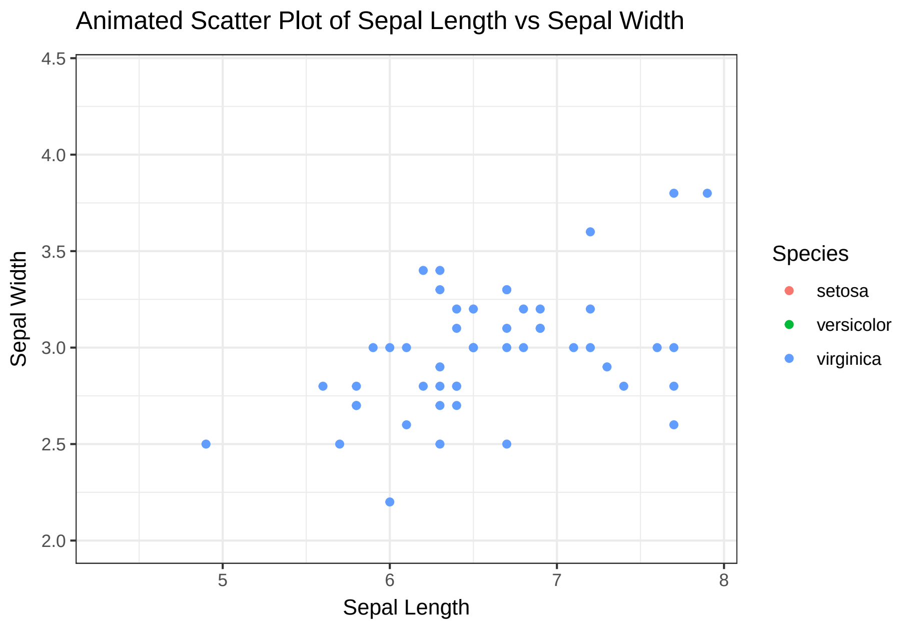

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

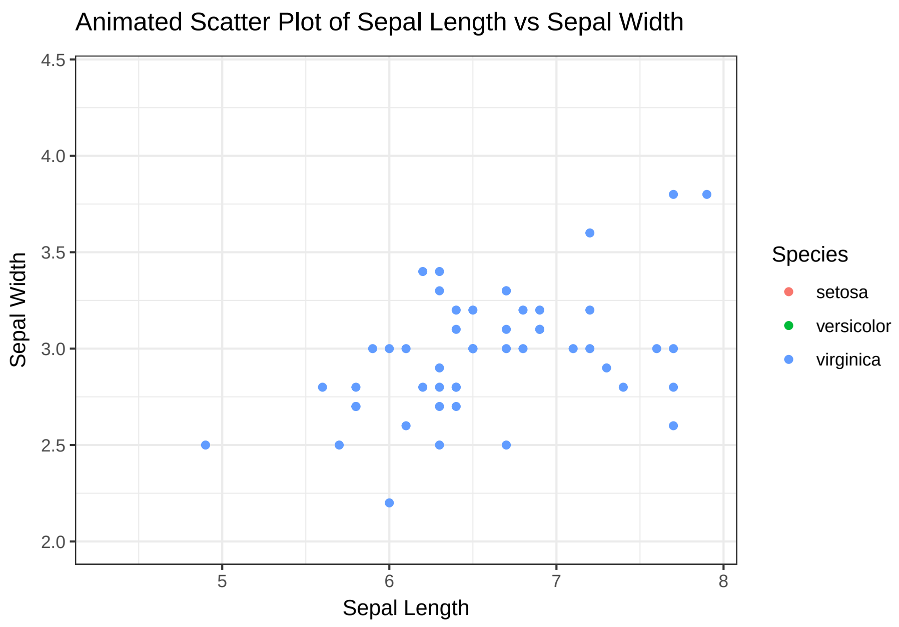

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

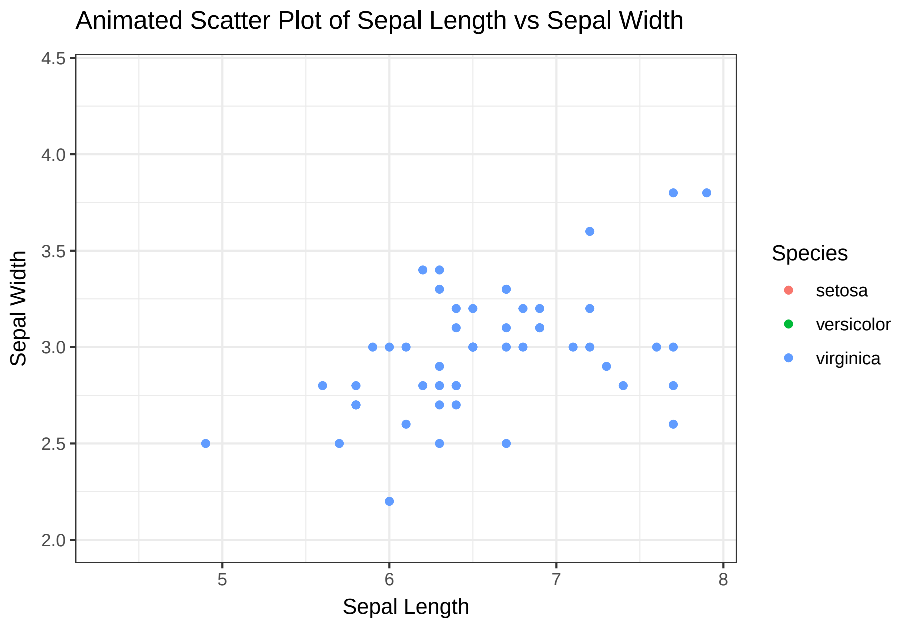

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

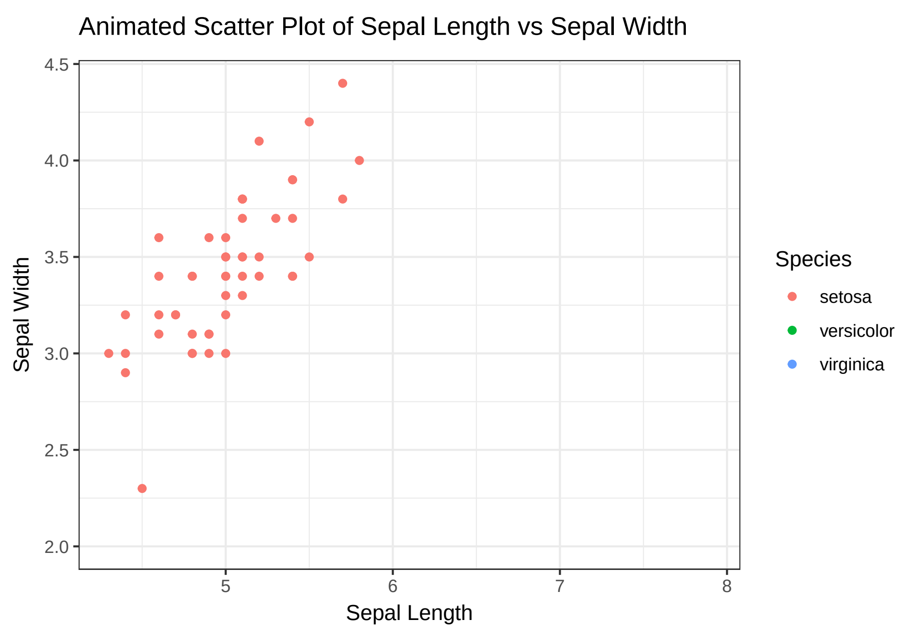

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica


Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

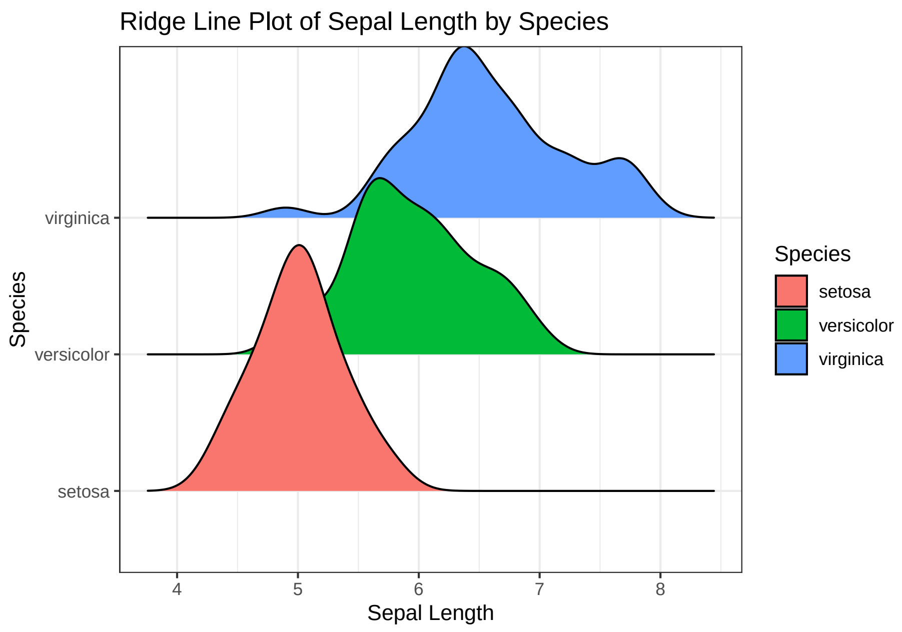

Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica


Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica


Animated Scatter Plot of Sepal Length vs Sepal Width
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica



```r
library(ggridges)

# Ridge line plot
ggplot(iris, aes(x = Sepal.Length, y = Species, fill = Species)) +
  geom_density_ridges() +
  labs(title = "Ridge Line Plot of Sepal Length by Species",
       x = "Sepal Length", y = "Species") +
  theme_bw()
```



9.4 Practical Examples: Exercises

Chapter 10

REAL-WORLD APPLICATIONS AND CASE STUDIES (CODE CHALLENGES)