# Assignment Zero: Practicing the basics of R

Introduction To Chaos Applied To Systems, Processes And Products (ETSIDI, UPM)

Alfonso Allen-Perkins, Juan Carlos Bueno and Eduardo Faleiro

2026-02-12

## Contents



## 1: Getting started with R

### 1.1 Installing R and RStudio (if not already installed).

RStudio is an interface that makes working with R easier.

- **R** can be downloaded from:
  https://cran.r-project.org/
- **RStudio** can be downloaded from:
  https://posit.co/download/rstudio-desktop/

### 1.2 Loading the necessary libraries

In R, a **library** is a collection of functions that extends what R can do.
We use the `library()` function to make a library available in our current R session.

In this course, we will use the **ggplot2** library to create plots and visualizations. To load it, type:

```
library(ggplot2)
```

If you get an error saying that the package is not installed, you need to install it first. Installing a library only needs to be done once. Just type: `install.packages("ggplot2")`.

As another example, try installing `deSolve`, a library for working with differential equations and dynamical systems.

**Important:** You install a package only once, but you must load it every time you start a new R session. Also, note that package names are written in quotes when installing, but not when loading them with `library()`.

### 1.3 Creating variables and assigning values

In R, we store values using **variables**.
A variable has a **name** and stores a **value** that can be used later.

To assign a value to a variable, we use the assignment operator `<-`.

```
x <- 5
```

Here:

- `x` is the variable name
- `5` is the value stored in `x`
- `<-` assigns the value to the variable

To display `x` and check that it was stored correctly, just type:

```
x
```

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

R will print the value of `x` in the Console.

You can then use the variable in your calculations:

```
x + 2
```

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

### 1.4 Creating other simple data structures: vector, matrix, and data frame in R.

These are three of the most common data structures in R:

- A **vector** stores a sequence of values of the same type, such as:
  - **numeric values** (e.g. `1`, `2.5`, `10`)
  - **integers** (e.g. `1L`, `2L`, `3L`)
  - **character strings** (e.g. `"Alice"`, `"Bob"`, `"Charlie"`)
  - **logical values** (e.g. `TRUE`, `FALSE`)
- A **matrix** stores values in rows and columns
- A **data frame** stores data in a table, where each column can represent a different variable

After creating each object, display it and check its structure.

```r
# Numeric vector
my_numeric_vector <- c(1, 2, 3, 4, 5)
my_numeric_vector
```

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

```r
# Character vector
my_character_vector <- c("Alice", "Bob", "Charlie")
my_character_vector
```

```
## [1] "Alice"   "Bob"     "Charlie"
```

```r
# Create a matrix
my_matrix <- matrix(1:9, nrow = 3)
my_matrix
```

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

```r
# Create a data frame
my_data_frame <- data.frame(
  Name = c("Alice", "Bob"),
  Age = c(25, 30)
)
my_data_frame
```

```
##    Name Age
## 1 Alice  25
## 2   Bob  30
```

### Comments in R

Comments are lines of text that are **ignored by R**. They are used to explain what the code does and to make scripts easier to read. In R, any text that comes after the `#` symbol is treated as a comment.

```r
# This is a comment
# R will not run this line

my_vector <- c(1, 2, 3)  # This is also a comment
my_vector
```

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

Comments are useful for:

- Explaining your code
- Taking notes for yourself
- Making your code easier for others to understand

### Classes in R

Every object in R has a **class**, which tells R how to treat it. To check an object's class, use the function `class()`.

```
class(my_character_vector)
```

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

```
class(my_matrix)
```

```
## [1] "matrix" "array"
```

```
class(my_data_frame)
```

```
## [1] "data.frame"
```

### Getting help in R

If you want to learn more about what a function does, you can open its documentation by typing `?` followed by the function name. For example, to learn more about `class()`, type `?class` in the Console and press enter. This will open the help page in RStudio, where you can see:

- What the function does
- Its arguments
- Examples of how to use it

---

## 2: Accessing elements of data structures in R

In R, we can access (or extract) specific elements from objects such as **vectors**, **matrices**, and **data frames** using **square brackets** `[ ]`.

---

### 2.1: Accessing elements of a vector

Vectors are one-dimensional, so we access elements using a single index.

```
my_vector <- c(10, 20, 30, 40, 50)

my_vector[1]   # First element
```

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

```
my_vector[3]   # Third element
```

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

Note: R uses 1-based indexing, meaning the first element has index 1 (not 0).

### 2.2: Accessing elements of a matrix

Matrices are two-dimensional, so we specify:

- the row index
- the column index

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

my_matrix[1, 2]  # Element in row 1, column 2
```

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

You can also extract:

```
my_matrix[ , 1]  # First column
```

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

```
my_matrix[2, ]   # Second row
```

```
## [1] 2 5 8
```

### 2.3: Accessing elements of a data frame

Data frames are similar to matrices, but columns can have different types.

```
my_data_frame <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 28)
)
```

Access elements using row and column indices:

```
my_data_frame[1, 2]   # First row, second column
```

```
## [1] 25
```

Access a column by name:

```
my_data_frame$Age
```

```
## [1] 25 30 28
```

```
my_data_frame[["Name"]]
```

```
## [1] "Alice"   "Bob"     "Charlie"
```

Summary: Use `[ ]` to access elements:

- Vectors use one index: `[i]`
- Matrices and data frames use two indices: `[row, column]`
- Data frame columns can be accessed by name using `$`
- Leaving an index empty means "select everything" in that dimension

---

## 3: Creating functions in R

A function in R is a reusable block of code that performs a specific task.
Functions help you:

- Avoid repeating the same code
- Make your code easier to read
- Organize your work into logical pieces

### 3.1 Define a function to compute a parabola:

```
parabola <- function(x) {
  return(x^2)
}
```

### 3.2 Define a function for a straight line:

```
straight_line <- function(x, m=1, b=0) {
  return(m * x + b)
}
```

### 3.3 Apply these functions to a sequence of input values and store the results in appropriate data structures.

For example, evaluate the functions over the interval $[-10, 10]$. To do so, first, we create a sequence of evenly spaced values:

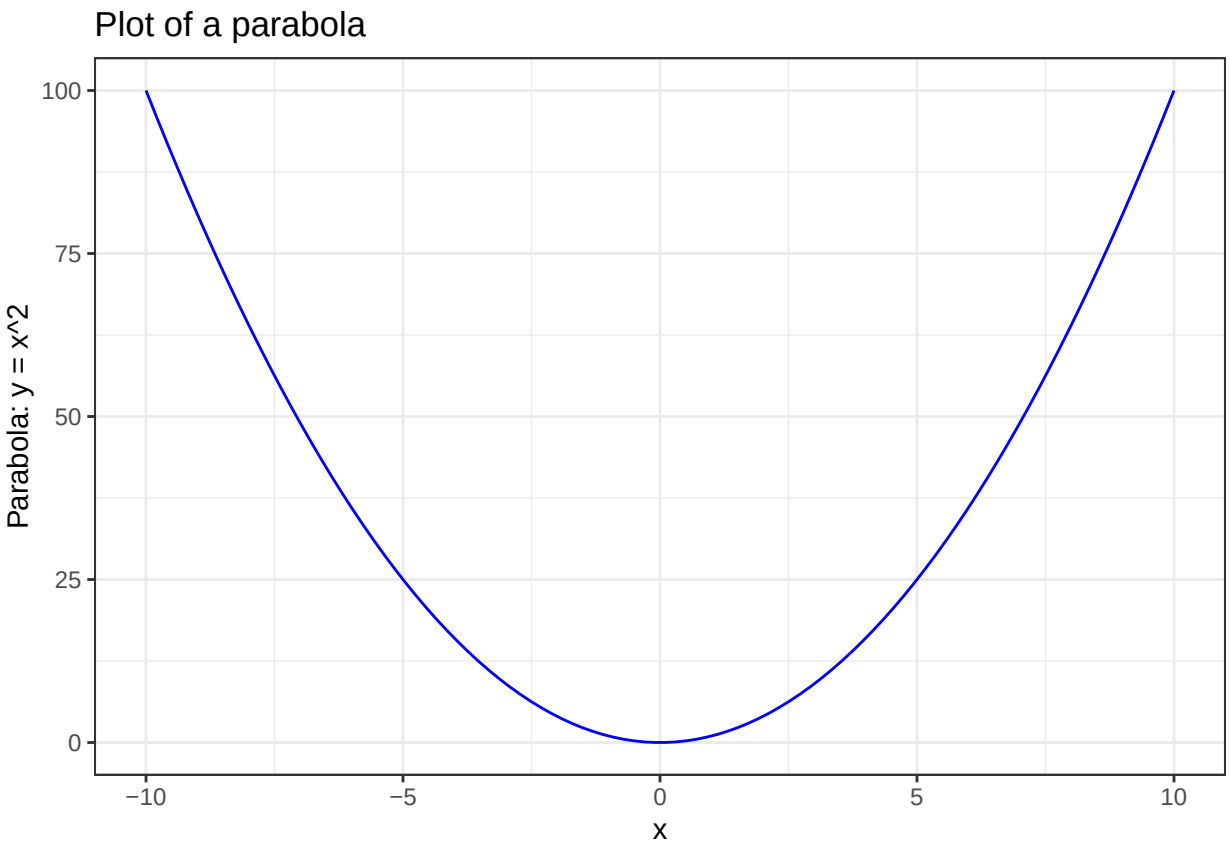
```
  x_values <- seq(-10, 10, length.out = 100)
```

Next, we apply each function to this sequence and store the results in data frames:

```
  datafr_parabola <- data.frame(x = x_values, y = parabola(x_values))
  datafr_line <- data.frame(x = x_values, y = straight_line(x_values, m=1, b=0))
```

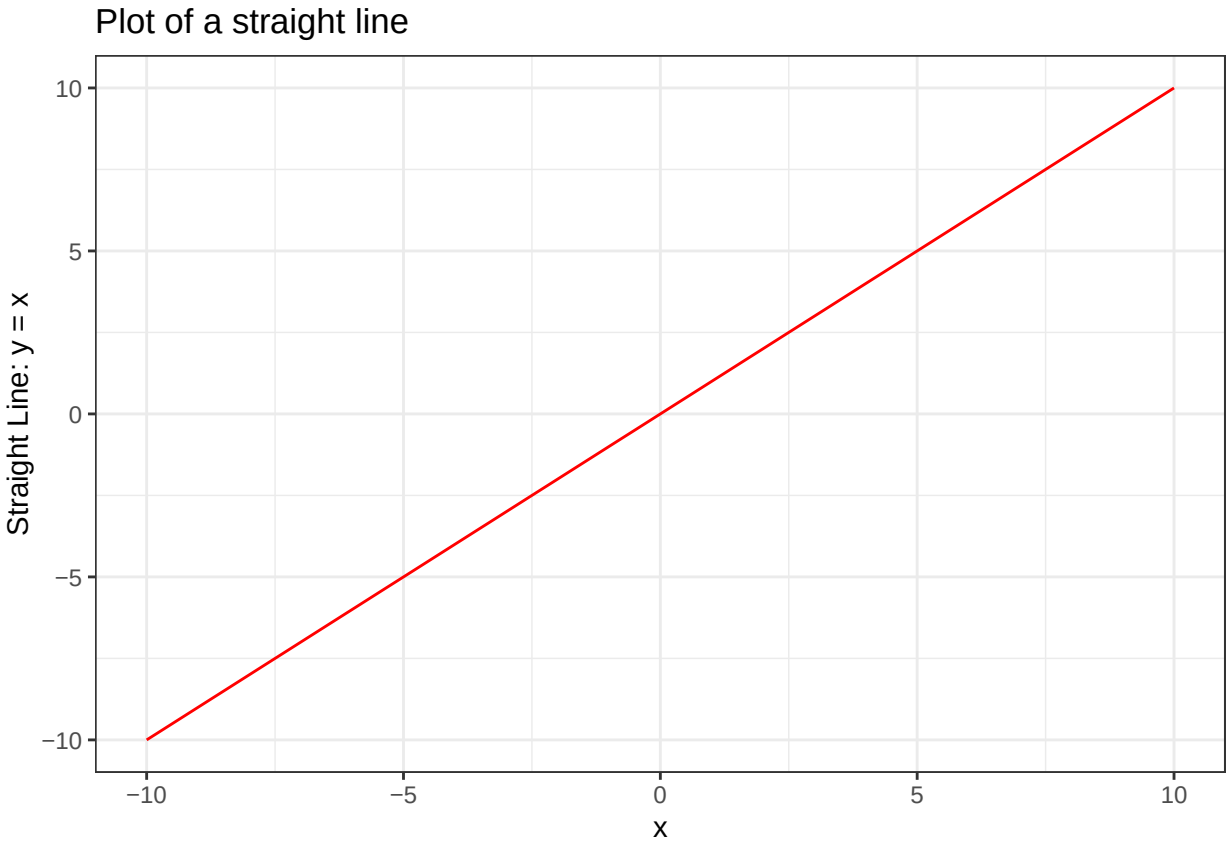
### 3.4 Use ggplot2 to plot both functions over the range [-10, 10].

```
# -------------------------
# Parabola plot
# -------------------------

# Create a plot using the data frame for the parabola
ggplot(data = datafr_parabola, aes(x = x, y = y)) +

  # Add a line representing y = x^2
  geom_line(color = "blue") +

  # Label the x-axis
  xlab("x") +

  # Label the y-axis
  ylab("Parabola: y = x^2") +

  # Add a title to the plot
  ggtitle("Plot of a parabola") +

  # Use a clean black-and-white theme
  theme_bw()
```




```
# -------------------------
# Straight line plot
# -------------------------
```

```r
# Create a plot using the data frame for the straight line
ggplot(data = datafr_line, aes(x = x, y = y)) +

  # Add a line representing y = x
  geom_line(color = "red") +

  # Label the x-axis
  xlab("x") +

  # Label the y-axis
  ylab("Straight Line: y = x") +

  # Add a title to the plot
  ggtitle("Plot of a straight line") +

  # Use the same black-and-white theme
  theme_bw()
```




---

## 4: Using a For loop

A for loop is a way to repeat the same piece of code multiple times, changing a value each time it runs.

In R, a for loop is typically used when: -You want to perform the same calculation many times -Each step depends on the previous one -You need to keep track of an index (like time or iteration number)

For example, we can use a `for` loop to print the numbers from 1 to 5.

```
for (i in 1:5) {
  print(i)
}
```

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

This loop works as follows:

- Start with `i = 1`
- Run the code inside the `{ }`
- Increase `i` to the next value
- Repeat until `i = 5`

**Example: The Logistic Map**

As an example of a system that can be studied using a `for` loop, we consider the **logistic map**.

The logistic map is a simple mathematical model that describes how a quantity evolves over time according to a fixed rule. It is often used to model **population growth**, where:

- $x_n$ represents the population at time (or iteration) $n$, scaled between 0 and 1
- $r$ is a parameter that controls the growth rate
- Each new value depends on the previous one

The model is defined by the recursive equation

$$x_{n+1} = rx_n(1 - x_n)$$

Starting from an initial condition $x_0$, the equation is applied repeatedly to generate a sequence $x_1$, $x_2$, $x_3$, ...

Because each value depends on the one before it, the logistic map is a natural example where a **for loop** is needed.

So here, starting from an initial condition $x_0$ and a given value of $r$, we compute the first **50 iterations** of the logistic map and visualize the result by plotting $x_n$ **versus** $n$ using `ggplot2`.

### 4.1 Define a function to compute the logistic map.

```r
# Define a function to compute the logistic map
logistic_map <- function(r, x0, n) {

  # Create a numeric vector to store the values of x_n
  x_values <- numeric(n)

  # Set the initial condition x_0
  x_values[1] <- x0

  # Use a for loop to compute the logistic map iteratively
  for (i in 2:n) {
    # Apply the logistic map equation:
    # x_{n+1} = r * x_n * (1 - x_n)
    x_values[i] <- r * x_values[i - 1] * (1 - x_values[i - 1])
  }

  # Return the results as a data frame with iteration number and values
  return(data.frame(
    n = 1:n,
    x = x_values
  ))
}
```

**4.2 Define the initial condition $x_0$, the value of $r$ and the number of interactions.**

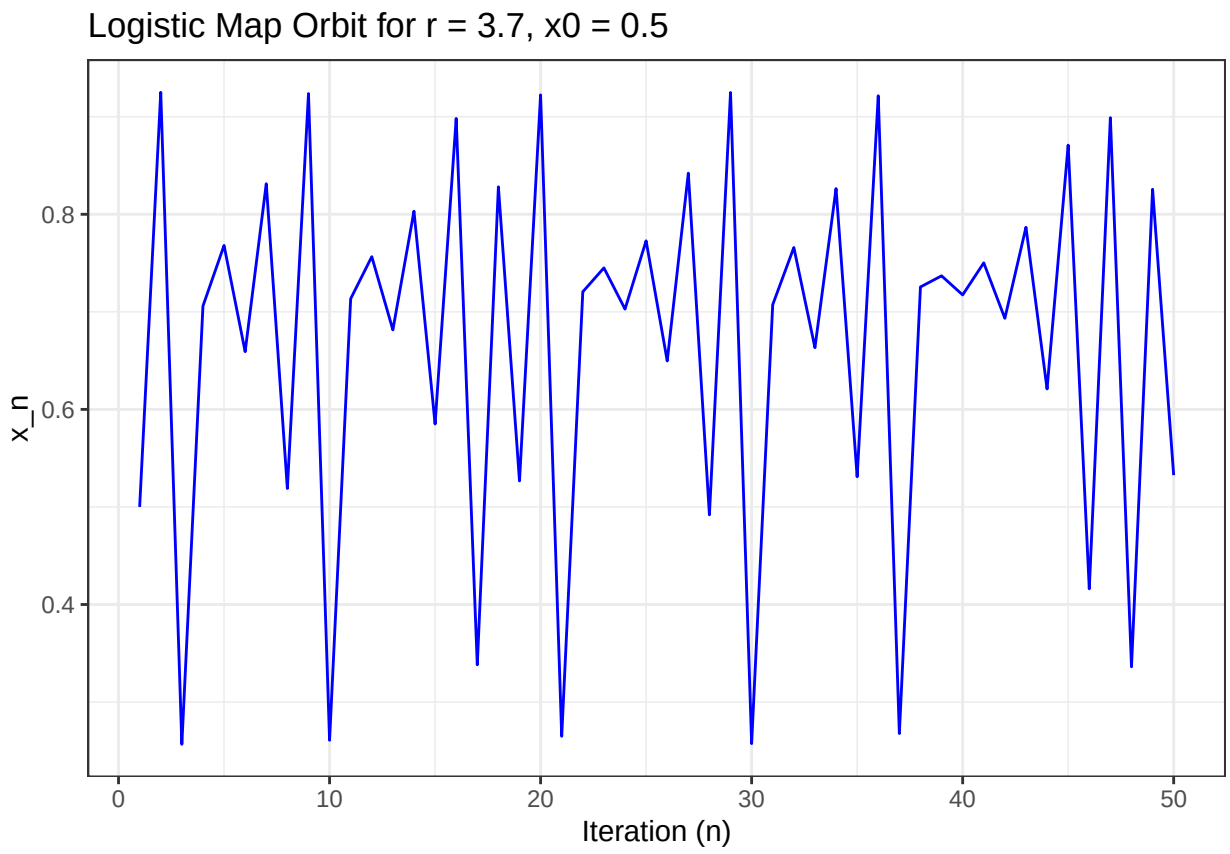
```r
# Example usage
r_value <- 3.7
x0_value <- 0.5
n_iter <- 50
```

**4.3 Compute the results.**

```r
orbit_data <- logistic_map(r_value, x0_value, n_iter)
```

**4.4 Visualize the result by plotting $x_n$ versus $n$ using `ggplot2`.**

```r
ggplot(orbit_data, aes(x = n, y = x)) +
  geom_line(color = "blue") +
  xlab("Iteration (n)") +
  ylab("x_n") +
  ggtitle("Logistic Map Orbit for r = 3.7, x0 = 0.5") +
  theme_bw()
```




**4.5 Try different values of $r$ (e.g., 2.5, 3.0, 3.5, 4.0) and observe the behavior.**

---

## 5: Finding and visualizing roots of functions

### 5.1. Finding roots of a polynomial

In R, the function `polyroot()` is used to find the **roots (zeros)** of a polynomial.
A root of a function is a value of $x$ for which the function is equal to zero.

5.1.1. Find all the roots of the polynomial $x^2 - 1$:

To use `polyroot()`, the polynomial must be written in terms of its **coefficients**, starting from the constant term and ordered by increasing powers of $x$.

For the polynomial $x^2 - 1$, we have:

$$-1 + 0x + 1x^2,$$

so the coefficients are `c(-1, 0, 1)`.

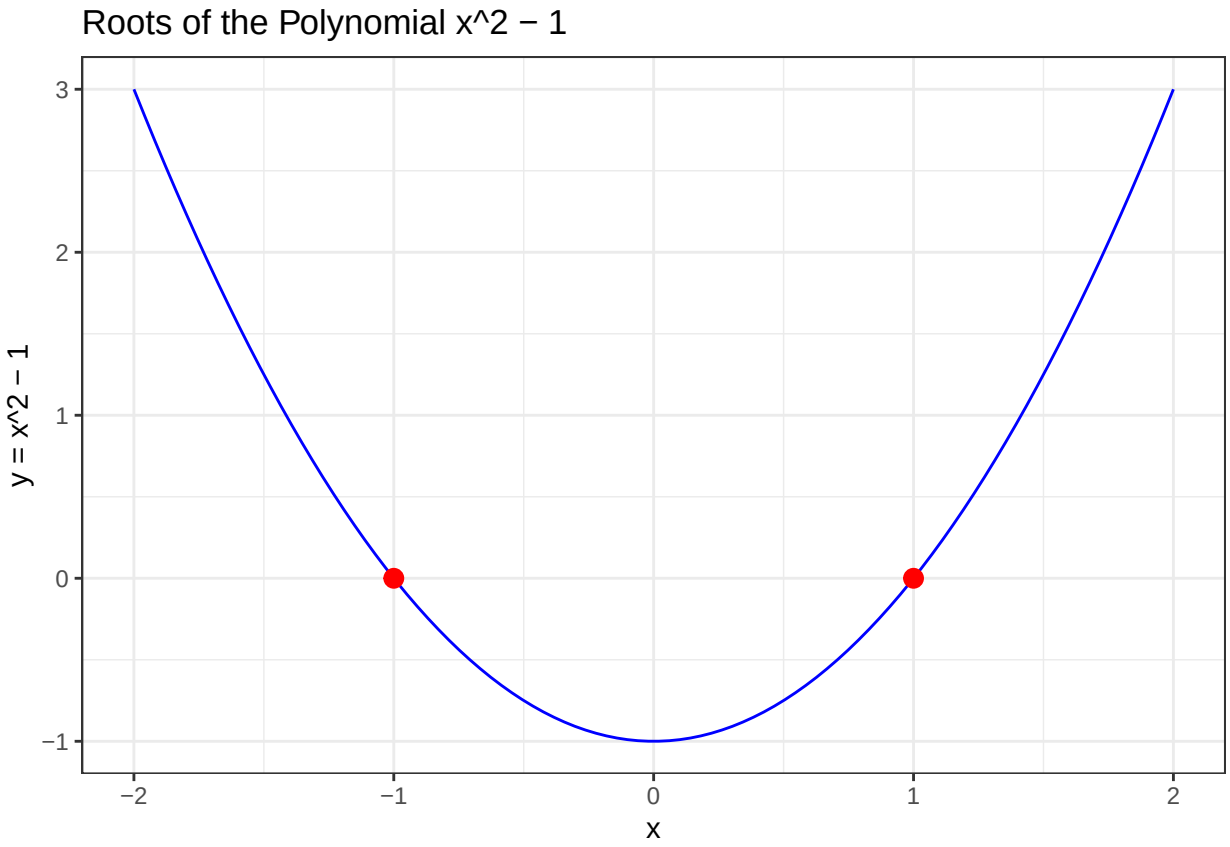
```
fixed_points <- polyroot(c(-1, 0, 1))
print(fixed_points)
```

```
## [1]  1+0i -1+0i
```

**Note:** To learn how to use the polyroot function, type `?polyroot` in the console.

5.1.2. Plot the polynomial function and highlight the roots:

```
x_values <- seq(-2, 2, length.out = 100)
y_values <- x_values^2 - 1

polynomial_curve_df <- data.frame(x = x_values, y = y_values)

ggplot(data = polynomial_curve_df, aes(x = x, y = y)) +
  geom_line(color = "blue") +
  geom_point(data = data.frame(x = Re(fixed_points), y = 0), aes(x = x, y = y), color = "red", size = 3
  xlab("x") +
  ylab("y = x^2 - 1") +
  ggtitle("Roots of the Polynomial x^2 - 1") +
  theme_bw()
```




### 5.2: Finding roots of a non-polynomial function

Not all functions are polynomials. For more general functions, we cannot use `polyroot()`.
Instead, we use numerical methods to approximate roots.

To find a root of this function in a given interval, we use the function `uniroot()`. This function searches for a value of $x$ where $f(x) = 0$, within a specified interval.

Consider the function

$$f(x) = e^x - 10\cos(x).$$

5.2.1. Define the function $f(x) = e^x - 10\cos(x)$

```
non_polynomial_curve_f <- function(x) {
  exp(x) - 10 * cos(x)
}
```

5.2.2. Find a root in a given interval (for example, $[0, 2]$):

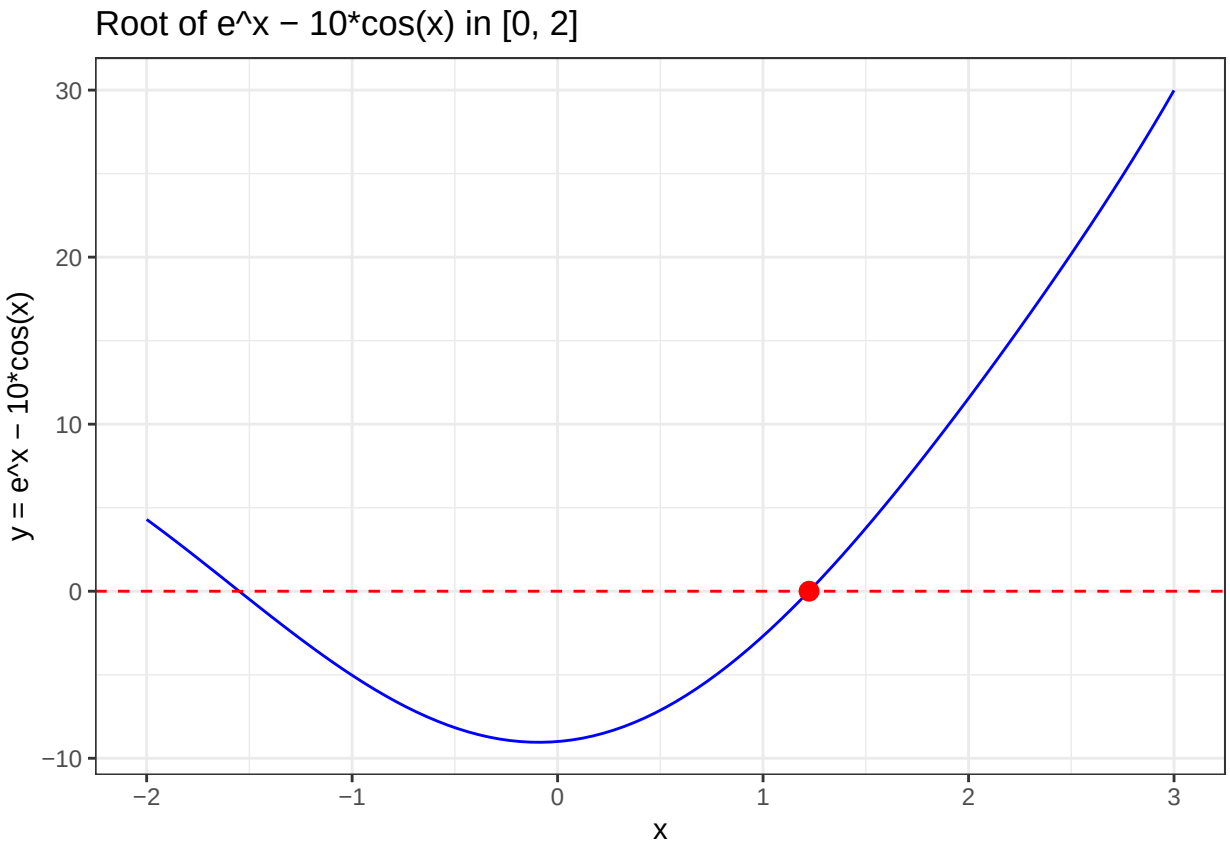
```
root_f <- uniroot(non_polynomial_curve_f, interval = c(0, 2))$root
print(root_f)
```

```
## [1] 1.223853
```

**Note:** To learn how to use the polyroot function, type `?uniroot` in the console.

5.2.2. Plot the function and highlight the root:

```
x_values <- seq(-2, 3, length.out = 100)

non_polynomial_curve_df <- data.frame(x = x_values, y = non_polynomial_curve_f(x_values))

ggplot(data = non_polynomial_curve_df, aes(x = x, y = y)) +
   geom_line(color = "blue") +
   geom_hline(yintercept = 0, linetype = "dashed", color = "red") +
   annotate("point", x = root_f, y = 0, color = "red", size = 3) +
   xlab("x") +
   ylab("y = e^x - 10*cos(x)") +
   ggtitle("Root of e^x - 10*cos(x) in [0, 2]") +
   theme_bw()
```




---

## Assigment: Reproducing figures from Strogatz's book (avalaible in Moodle)

For this assignment, you will reproduce figures from *Nonlinear Dynamics and Chaos* by Steven H. Strogatz (available in Moodle):

https://www.biodyn.ro/course/literatura/Nonlinear_Dynamics_and_Chaos_2018_Steven_H._Strogatz.pdf

Your goal is to generate the same type of plot using R (and `ggplot2`) by:

- implementing the mathematical model or equation used in the figure,
- choosing appropriate parameters and initial conditions,
- generating the data numerically (e.g., with a loop or a function),
- and creating a plot that matches the original figure as closely as possible (axes, labels, shape, behavior, etc.).

### Task 1: Reproduce Figure 2.7.3

- Create an **R script** that uses `ggplot2` to reproduce **Figure 2.7.3** from Strogatz's book.
- Save the script as `figure_2_7_3.R` and generate the corresponding plot.

### Task 2: Reproduce Figure 10.2.3

- Create another script, `figure_10_2_3.R`, to replicate **Figure 10.2.3** from Strogatz's book.
- Save the resulting figures as PNG or PDF files.
- Submit both **scripts and figures** via email.

---

**Note:** Ensure that your plots use sufficient data points and appropriate labels for clarity.