Probability and Statistics (IT302) Class No. 26
14th October 2020 Wednesday 11:15 AM - 11:45 AM

Example: Find Roots of a Quadratic Equation

Here is a simple example of a program for calculating the real roots of a quadratic equation. **Note the use of # for commenting the code.**

```
# program: spuRs/resources/scripts/quad1.r        find the zeros of a2*x^2 + a1*x + a0 = 0
# clear the workspace
rm(list=ls())
# input
a2 <- 1
a1 <- 4
a0 <- 2
# calculation
root1 <- (-a1 + sqrt(a1^2 - 4*a2*a0))/(2*a2)
root2 <- (-a1 - sqrt(a1^2 - 4*a2*a0))/(2*a2)
# output
print(c(root1, root2))
```

Executing this code (running the program) produces the following output

```
> source("../scripts/quad1.r")
[1]  -0.5857864  -3.4142136
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: Summing a Vector

The following example uses a loop to sum the elements of a vector. Note that **function cat (for concatenate) is used to display the values of certain variables**. The advantage of cat over print or show is that it allows us to combine text and variables together. The combination of characters **\n (backslash-n) is used to 'print' a new line**.

Also note that to sum the elements of a vector, it is more accurate and much easier (but less instructive) to use the built-in function sum.

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: Summing a Vector Contd.

```
x_list <- seq(1, 9, by = 2)
sum_x <- 0
for (x in x_list)
{     sum_x <- sum_x + x
      cat("The current loop element is", x, "\n")
      cat("The cumulative total is", sum_x, "\n")
}
print(sum(x_list))
```

```
x_list <- seq(1, 9, by = 2)

sum_x <- 0

for (x in x_list)
{     sum_x <- sum_x + x +
    cat("The current loop element is", x, "\n")  +
    cat("The cumulative total is", sum_x, "\n")
}
print(sum(x_list))
```

Output

```
The current loop element is 1
The cumulative total is 0
The current loop element is 3
The cumulative total is
The current loop element is 5
The cumulative total is
The current loop element is 7
The cumulative total is
The current loop element is 9
The cumulative total is
[1] 25
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: n factorial

```
# Calculate n factorial
# clear the workspace
rm(list=ls())

# Input
n <- 6

# Calculation
n_factorial <- 1
for (i in 1:n) {  n_factorial <- n_factorial * i }

# Output
show(n_factorial)
```

Output

```
[1] 720
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: n factorial

Note that we can also compute the factorial easily using **prod(1:n)** or even **factorial(n)**.

```
# Calculate n factorial
# clear the workspace
rm(list=ls())
# Input
n <- 7
show(prod(1:n))
```

Output

```
[1]       5040
```

```
# Calculate n factorial
# clear the workspace
rm(list=ls())
# Input
n <- 7
show(factorial(n))
```

Output

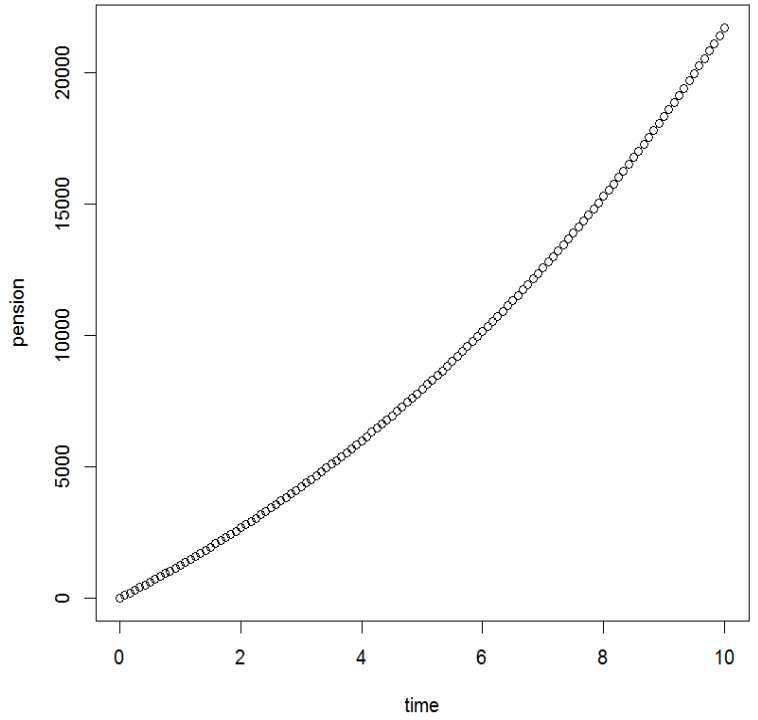
```
[1]       5040
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: Pension Value

Here is an example for calculating the value of a pension fund under compounding interest. It uses the function floor(x), whose value is the largest integer smaller than x.

```
# Forecast pension growth under compound interest
# clear the workspace
rm(list=ls())
# Inputs
r <- 0.11                    # Annual interest rate
term <- 10                   # Forecast duration (in years)
period <- 1/12               # Time between payments (in years)
payments <- 100              # Amount deposited each period
# Calculations
n <- floor(term/period)      # Number of payments
pension <- 0
for (i in 1:n) {pension[i+1] <- pension[i]*(1 + r*period) + payments}
time <- (0:n)*period
# Output
plot(time, pension)
```



Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: Fibonacci Numbers Contd.

Consider the Fibonacci numbers F1, F2, . . ., which are defined inductively using the rules $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. Suppose that you wished to know the first Fibonacci number larger than 100. We can find this using a while loop as follows:

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: Fibonacci Numbers Contd.

```
# program: spuRs/resources/scripts/fibonacci.r
# calculate the first Fibonacci number greater than 100
# clear the workspace
rm(list=ls())
# initialise variables
F <- c(1, 1)                    # list of Fibonacci numbers
n <- 2                          # length of F
# iteratively calculate new Fibonacci numbers
while (F[n] <= 100) {
# cat("n =", n, " F[n] =", F[n], "\n")
n <- n + 1
F[n] <- F[n-1] + F[n-2]
}
# output
cat("The first Fibonacci number > 100 is F(", n, ") =", F[n], "\n")
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: Compound Interest

```
# program: spuRs/resources/scripts/compound.r    Duration of a loan under compound interest
# clear the workspace
rm(list=ls())
# Inputs
r <- 0.11                    # Annual interest rate
period <- 1/12               # Time between repayments (in years)
debt_initial <- 1000         # Amount borrowed
repayments <- 12             # Amount repaid each period
# Calculations
time <- 0
debt <- debt_initial
while (debt > 0) {  time <- time + period
                      debt <- debt*(1 + r*period) - repayments
                    }
# Output
cat('Loan will be repaid in', time, 'years\n')
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Vector-based programming

It is often necessary to perform an operation upon each of the elements of a vector. For example, given a vector of measurements in inches, we may wish to convert them all to centimetres. R is set up so that such programming tasks can be accomplished using vector operations rather than looping. Using vector operations is more efficient computationally, as well as more concise literally.

For example, we could find the sum of the first n squares using a loop as follows:

```
> n <- 100
> S <- 0
> for (i in 1:n) {
+     S <- S + i^2
+ }
> S

[1] 338350
```

Alternatively, using vector operations we have:

```
> sum((1:n)^2)

[1] 338350
```

Here, R has interpreted 1:n to mean “the integers from 1 up to n, inclusive”, then squared each of those integers using the vectorised “^2”, and added them up in sum.

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

ifelse function

The powerful ifelse function performs elementwise conditional evaluation upon a vector. **ifelse(test, A, B) takes three vector arguments: a logical expression test, and two expressions A and B.** The function returns a vector that is a combination of the evaluated expressions A and B: **the elements of A that correspond to the elements of test that are TRUE,** and the elements of B that correspond to the elements of test that are FALSE.

As before, if the vectors have differing lengths then R will repeat the shorter vector(s) to match the longer, if possible. An example follows.

```
> x <- c(-2, -1, 1, 2)

> ifelse(x > 0, "Positive", "Negative")

[1] "Negative" "Negative" "Positive" "Positive"
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

pmin and pmax

Two other useful functions are pmin and pmax, which provide vectorised versions of the minimum and maximum.

Example-1

```
# R program to illustrate
# pmin() function

# First example vector
x1 <- c(2, 9, 5, 7, 1, 8)

# Second example vector
x2 <- c(0, 7, 2, 3, 9, 6)

# Application of pmin() in R
pmin(x1, x2)
```

Output:

```
[1]       0  7 2 3 1 6
```

Example-2

```
# R program to illustrate
# pmin() function

# First example vector
x1 <- c(2, 9, 5, 7, 1, 8)

# Second example vector
x2 <- c(0, 7, 2, 3, 9, 6)

# Application of pmin() in R
pmax(x1, x2)
```

Output:

```
[1]       2 9 5 7 9 8
```

Example-3

```
> pmin( c(1,2,3),   c(3,2,1),   c(2,2,2))
```

Output:

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

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Source : https://www.geeksforgeeks.org/compute-the-parallel-minima-and-maxima-between-vectors-in-r-programming-pmin-and-pmax-functions/

Basic debugging

We recognise errors in a few ways. First, R may stop processing and report an error, along with a brief summary of the violation. Second, we may identify an error by examining the output of a program or function, and noting that it makes no sense. Finally, R may throw a warning upon the execution of some code, and this warning might point to an earlier error. To ask R to convert warnings into errors, and hence stop processing at the point of warning, type

> options(warn = 2)

You will spend a lot of time correcting errors in your programs. To find an error or bug, you need to be able to see how your variables change as you move through the branches and loops of your code.

An effective and simple way of doing this is to include statements like cat("var =", var, "\n") throughout the program, to display the values of variables such as var as the program executes.

Once you have the program working you can delete these or just comment them so they are not executed.

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Debugging Example

For example, if we wanted to see how the variable i changed in the program above, we could add a line as follows:

```
# program: spuRs/resources/scripts/threexplus1.r
x <- 3
for (i in 1:3)
{ show(x)
  cat("i = ", i, "\n")
  if (x %% 2 == 0) {x <- x/2}
  else
    {x <- 3*x + 1}
 }
show(x)
```

Running the program gives the following output

```
> source("../scripts/threexplus1.r")
[1] 3
i = 1
[1] 10
i = 2
[1] 5
i = 3
[1] 16
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Strings

- Character strings are denoted using either double quotes " " or single quotes ' '. Strings can be arranged into vectors and matrices just like numbers.

- We can also paste strings together using paste(..., sep). Here sep is an optional input, with default value " ", that determines which padding character is to be placed between the strings (which are input where ... appears).

```
> x <- "Citroen SM"
> y <- "Jaguar XK150"
> z <- "Ford Falcon GT-HO"
> (wish.list <- paste(x, y, z, sep = ", "))
```

[1] "Citroen SM, Jaguar XK150, Ford Falcon GT-HO“

Special characters can be included in strings using the escape character \. **Use \" for ";**
\n for a newline; **\t for a tab;** **\b for a backspace;** and **\\ for \.**

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Strings Contd.

If a character string x can be understood as a number, then as.numeric(x) coerces it to be that number. **Use as.character(x) to coerce a number x into a character string**, though note that R will often do this for you as your code requires.

A generally more useful method of converting a number to a character string is to use the function **format(x, digits, nsmall, width)**.

digits, nsmall, and width are all optional:

- **nsmall** suggests how many decimal places to use;
- **digits** suggests how many significant digits to include;
- and **width** suggests how long the total character string should be.

Note that R will quite happily override your suggested values for digits, nsmall, and width. **This can be avoided by using the function round(x, k) to round x to k digits before you use format.**

command cat displays concatenated character strings.

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example Shows how to Use Formatted Output to Print a Table of Numbers

The program writes out the first n powers of the number x.

```
# program spuRs/resources/scripts/powers.r
# display powers 1 to n of x
# input
x <- 7
n <- 5
# display powers
cat("Powers of", x, "\n")
cat("exponent result\n\n")
result <- 1
for (i in 1:n) {    result <- result * x
                    cat(format(i, width = 8),
                    format(result, width = 10),
                    "\n", sep = "")
                }
```

It produces the following output:

```
> source("../scripts/powers.r")

Powers of  7
exponent          result
       1               7
       2              49
       3             343
       4            2401
       5           16807
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example Shows how to Use Formatted Output to Print a Table of Numbers Contd.

Functions format and paste also take vector input. Thus the program above could be vectorised as follows:

```
> cat(paste(format(1:n, width=8),  format(x^(1:n), width=10), "\n"), sep="")
```

```
1        7
2       49
3      343
4     2401
5    16807
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Input from a file

R provides a number of ways to read data from a file, the most flexible of which is the scan function. We use scan to read a vector of values from a file. scan has a large number of options, of which we only need a few at this point. It has the form

scan(file = "", what = 0, n = -1, sep = "", skip = 0, quiet = FALSE)

scan returns a vector. All the parameters are optional; the defaults are indicated above.

file gives the file to read from. The default "" indicates read from the keyboard.

what gives an example of the mode of data to be read, **with a default of 0 for numeric data**. **Use " " for character data**.

n gives the number of elements to read. **If n = -1 then scan keeps reading until the end of the file.**

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Input from a file Contd.

- **sep** allows you to specify the character that is used to separate values, such as ",". The default “ " has the special meaning of allowing any amount of white space (including tabs) to separate values. Note that a newline/return always separates values.

- **skip** is the number of lines to skip before you start reading, default of 0. This is useful if your file includes some lines of description before the data start.

- **quiet** controls whether or not scan reports how many values it has read, default FALSE.

- If you try to read more items than are left in the file, by specifying n, then scan returns a vector of reduced length, possibly of length 0.

- To find out what files are in directory dir.name, use dir(path = "dir.name"), or equivalently list.files(path = "dir.name").

- The directory address can be relative to the current working directory or an absolute address. path has the default value ".", denoting the current working directory.

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: File Input

The following program reads a vector of numbers from a file then calculates their median, 1st quartile and 3rd quartile. The 100p-th percentage point of a sample is defined to be the smallest sample point x such that at least a fraction p of the sample is less than or equal to x. The first quartile is the 25% point of a sample, the third quartile the 75% point, and the median is the 50% point. (Note that some definitions of the quartiles and median vary slightly from these.)

For this example the file data1.txt was created beforehand using a text editor, and is stored in the directory ../data, which is a sibling to the working directory (that is, it has the same parent directory as the working directory).

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: File Input Contd.

```
# program: spuRs/resources/scripts/quartiles1.r
# Calculate median and quartiles.
# Clear the workspace
rm(list=ls())
# Input # We assume that the file file_name consists of numeric values  separated by spaces and/or newlines
file_name = "../data/data1.txt“
# Read from file
data <- scan(file = file_name)
# Calculations
n <- length(data)
data.sort <- sort(data)
data.1qrt <- data.sort[ceiling(n/4)]
data.med <- data.sort[ceiling(n/2)]
data.3qrt <- data.sort[ceiling(3*n/4)]
# Output
cat("1st Quartile:", data.1qrt, "\n")
cat("Median: ", data.med, "\n")
cat("3rd Quartile:", data.3qrt, "\n")
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Example: File Input Contd.

Suppose that the file data1.txt has the following single line
8 9 3 1 2 0 7 4 5 6

Running the program then produces the following output:

```
> source("../scripts/quartiles1.r")

1st Quartile: 2
Median: 4
3rd Quartile: 7
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Output to a File

R provides a number of commands for writing output to a file. We will generally use write or write.table for writing numeric values and cat for writing text, or a combination of numeric and character values.

The command write has the form

write(x, file = "data", ncolumns = if(is.character(x)) 1 else 5, append = FALSE)

Here x is the vector to be written. If x is a matrix or array then it is converted to a vector (column by column) before being written. The other parameters are optional.

file gives the file to write or append to, as a character string. The default "data" writes to a file called data in the current working directory. To write to the screen use file = "".

ncolumns gives the number of columns in which to write the vector x. The default is 5 for numbers and 1 for characters. Note that the vector is written row by row.

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Output to a File Contd.

```
> (x <- matrix(1:24, nrow = 4, ncol = 6))
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24

> write(t(x), file = "../results/out.txt", ncolumns = 6)
```

Here is what the file out.txt looks like:

```
1  5  9  13 17 21
2  6  10 14 18 22
3  7  11 15 19 23
4  8  12 16 20 24
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Output to a File Contd.

A more flexible command for writing to a file is cat, which has the form

cat(..., file = "", sep = " ", append = FALSE)

... is a list of expressions (separated by commas) that are coerced into character strings, concatenated, and then written.

file gives the file to write or append to, as a character string. The default "" writes to the screen.
sep is a character string that is inserted between the written objects, with default value " ".

ap**pend** indicates whether to append to or overwrite the file, with default FALSE.

Note that cat does not automatically write a newline after the expressions .... If you want a newline you must explicitly include the string "\n".

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

Plotting

- **plot(x, y)** used to plot one vector against another, with the x values on the x-axis and the y values on the y-axis.

- In fact the input y is optional, and if omitted then x is plotted against 1:length(x) (so you get the x values on the y-axis and 1:length(x) on the x-axis).

- In addition to type, some other useful optional parameters are xlab, ylab, and main, which all take character strings and are used to label the x-axis, y-axis and the whole plot, respectively.

- To add points (x[1], y[1]), (x[2], y[2]), ... to the current plot, use points(x, y).

- To add lines instead use lines(x, y). Vertical or horizontal lines can be drawn using abline(v = xpos) and abline(h = ypos). Both points and lines take the optional input col, which determines the colour ("red", "blue", etc.).

- If the **current plot does not have a title, then title(main) will provide one (here main is a character string).**

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

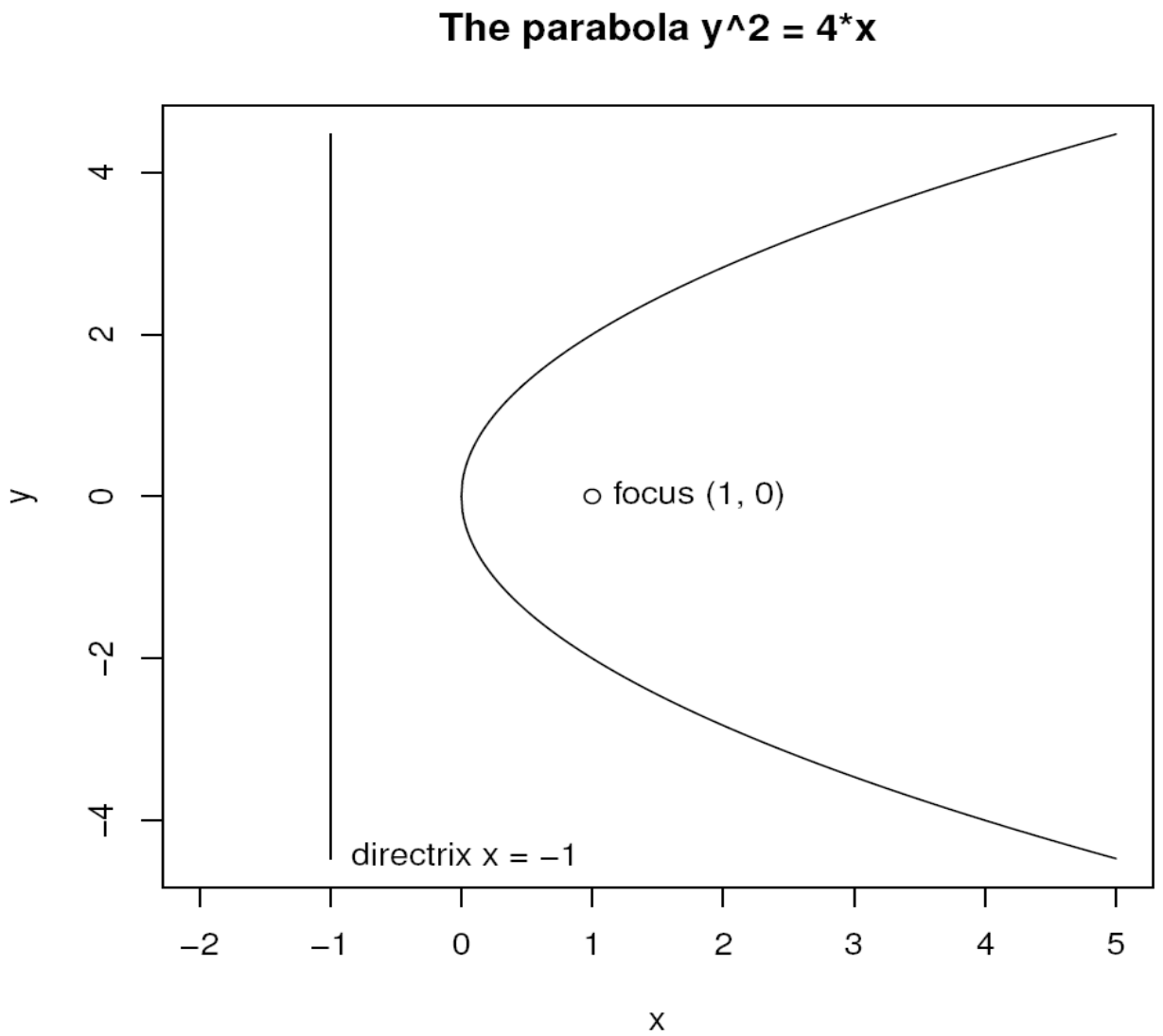
Plotting Contd.

As an example we plot part of the parabola $y^2 = 4x$, as well as its focus and directrix. We make use of the surprisingly useful input type = "n", which results in the graph dimensions being established, and the axes being drawn, but nothing else.

```
> x <- seq(0, 5, by = 0.01)
> y.upper <- 2*sqrt(x)
> y.lower <- -2*sqrt(x)
> y.max <- max(y.upper)
> y.min <- min(y.lower)
> plot(c(-2, 5), c(y.min, y.max), type = "n", xlab = "x", ylab = "y")
> lines(x, y.upper)
> lines(x, y.lower)
> abline(v=-1)
> points(1, 0)
> text(1, 0, "focus (1, 0)", pos=4)
> text(-1, y.min, "directrix x = -1", pos = 4)
> title("The parabola y^2 = 4*x")
```

Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers

The parabola $y^2 = 4x$



Source : Introduction to Scientific Programming and Simulation Using R Second Edition by John M. Chambers