Solutions to Exercises from Computational Physics with R

Contents

1 Chapter 01

1.1 Exercises on the computer representation of numbers

1.1.1 Exercise 01

Transform the following numbers from base 2 to base 10:

1. 1000.101
2. 111.11

SOLUTION

The bits of a binary number (base 2) multiply powers of 2. Thus:

1. $1000.101 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 8 + 0.5 + 0.125 = 8.625$

2. $111.11 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 2 + 1 + 0.5 + 0.25 = 7.75$

1.1.2 Exercise 02

Transform the following numbers from base 10 to base 2:

1. 32.3125
2. 5.375

SOLUTION

We can divide each number into its integer and decimal part. For the integer part we keep dividing by two and consider the remainder; for the decimal part, we multiply by 2 and use 0 if the result is smaller than 1 and 1 if it is greater or equal than 1.

1. Let us focus on the integer part, first. Using division by 2 consecutively we obtain:

$$
\begin{array}{rcrcrcl}
32 & \div & 2 & = & 16 & \text{remainder} & 0 \\
16 & \div & 2 & = & 8 & \text{remainder} & 0 \\
8 & \div & 2 & = & 4 & \text{remainder} & 0 \\
4 & \div & 2 & = & 2 & \text{remainder} & 0 \\
2 & \div & 2 & = & 1 & \text{remainder} & 0 \\
1 & \div & 2 & = & 0 & \text{remainder} & 1 \\
0 & & & & & \text{STOP} &
\end{array}
$$

The integer part of the number, 32, is represented by 100000 in base 2. Next, let us consider 0.3125 (the decimal part of the number), and follow a series of multiplications by 2:

$$
\begin{array}{rcrcllll}
0.3125 & \times & 2 & = & 0.625 & \text{take integer :} \; 0 & \text{value for next line :} & 0.625 \\
0.625 & \times & 2 & = & 1.25 & \text{take integer :} \; 1 & \text{value for next line :} & 0.25 \\
0.25 & \times & 2 & = & 0.5 & \text{take integer :} \; 0 & \text{value for next line :} & 0.5 \\
0.5 & \times & 2 & = & 1 & \text{take integer :} \; 1 & \text{value for next line : STOP} &
\end{array}
$$

The decimal part is therefore represented by 0101. In conclusion:

$$32.3125 \; \rightarrow \; 100000.0101.$$

2. Let us focus on the integer part, first. Using division by 2 consecutively we obtain:

$$
\begin{array}{rcrcrcl}
5 & \div & 2 & = & 2 & \text{remainder} & 1 \\
2 & \div & 2 & = & 1 & \text{remainder} & 0 \\
1 & \div & 2 & = & 0 & \text{remainder} & 1 \\
0 & & & & & \text{STOP} &
\end{array}
$$

The integer part of the number, 5, is represented by 101 in base 2. Next, let us consider 0.375 (the decimal part of the number), and follow a series of multiplications by 2:

$$\begin{array}{rcccll} 0.375 & \times & 2 & = & 0.75 & \text{take integer : } 0 \quad \text{value for next line : } 0.75 \\ 0.75 & \times & 2 & = & 1.5 & \text{take integer : } 1 \quad \text{value for next line : } 0.5 \\ 0.5 & \times & 2 & = & 1 & \text{take integer : } 1 \quad \text{value for next line : STOP} \end{array}$$

The decimal part is therefore represented by 011. In conclusion:

$$5.375 \ \rightarrow \ 101.011.$$

1.1.3 Exercise 03

Find the base 10 expression of the three consecutive numbers, $01101001, 01101010, 01101011$, written in the toy 8-bit IEEE system.

SOLUTION

Let us start from the first number. The initial 0 means that the number is positive. The next three bits define the exponent. 110 is the integer 6. Considering the offset 3, this corresponds to $2^{6-3} = 2^3$. The following four bits have to be interpreted as values following 1 and the floating point:

$$1001 \ \rightarrow \ 1.1001.$$

This number is multiplied by 2^3. Therefore:

$$01101001 \ \rightarrow \ 1.1001 \times 2^3 \ \rightarrow \ 1100.1 \ \rightarrow \ 12.5.$$

Thus, 01101001 corresponds to 12.5. The next, adjacent number is 01101010, and we have:

$$01101010 \ \rightarrow \ 1.1010 \times 2^3 \ \rightarrow \ 1101.0 \ \rightarrow \ 13.$$

Finally, for the next (and last) adjacent number we have:

$$01101011 \ \rightarrow \ 1.1011 \times 2^3 \ \rightarrow \ 1101.1 \ \rightarrow \ 13.5.$$

1.1.4 Exercise 04

The smallest normal positive number in the IEEE 8-bit toy system is 0.25. It is possible to represent exactly a handful of numbers smaller than 0.25, but greater than 0 within the IEEE system, the so-called subnormal numbers (see main text). List all 15 subnormal numbers representable with the IEEE 8-bit system.

SOLUTION

Subnormal numbers are of the form

$$0 \ 000 \ xxxx,$$

where the four bits in the significand cannot all be zero at the same time (this situation is identified as the

number 0). These values are then multiplied by the smallest power of 2 usabe, which is 2^{-2}. We have, thus

$$\begin{array}{llll} 0 & 000 & 0001 & \rightarrow \quad 0.015625 \\ 0 & 000 & 0010 & \rightarrow \quad 0.03125 \\ 0 & 000 & 0011 & \rightarrow \quad 0.046875 \\ 0 & 000 & 0100 & \rightarrow \quad 0.0625 \\ 0 & 000 & 0101 & \rightarrow \quad 0.078125 \\ 0 & 000 & 0110 & \rightarrow \quad 0.09375 \\ 0 & 000 & 0111 & \rightarrow \quad 0.109375 \\ 0 & 000 & 1000 & \rightarrow \quad 0.125 \\ 0 & 000 & 1001 & \rightarrow \quad 0.140625 \\ 0 & 000 & 1010 & \rightarrow \quad 0.15625 \\ 0 & 000 & 1011 & \rightarrow \quad 0.171875 \\ 0 & 000 & 1100 & \rightarrow \quad 0.1875 \\ 0 & 000 & 1101 & \rightarrow \quad 0.203125 \\ 0 & 000 & 1110 & \rightarrow \quad 0.21875 \\ 0 & 000 & 1111 & \rightarrow \quad 0.234375 \end{array}$$

There are, indeed, 15 subnormal numbers.

1.1.5 Exercise 05

Calculate the smallest positive normal number and the largest positive subnormal number in the IEEE 32-bit system.

SOLUTION

The smallest positive number is given by the formula

$$\beta^{e_{\min}+1},$$

where $\beta = 2$ and $[e_{\min}, e_{\max}] = [-127, 128]$ for the IEEE 32-bit system. The smallest positive normal number is thus

$$2^{-127+1} = 2^{-126} \approx 1.175494350 \times 10^{-38}.$$

To calculate the largest positive subnormal number, we could proceed similarly to what done with the IEEE 8-bit toy system (see previous exercise). But this would absorb too much time. Rather, and more conveniently, we can simply subtract an appropriate power of 2 from the smallest normal number, and obtain the same result. Such a value is

$$2^{e_{\min}+1} \times 2^{-m},$$

where m is the number of bits in the significand. For the 32-bit system we have $e_{\min} = -127$ and $m = 23$ and therefore the appropriate power of 2 is

$$2^{-126} \times 2^{-23} = 2^{-149} \approx 1.401298464324817 \times 10^{-45}.$$

This number is, as expected, very small and therefore the largest subnormal number is very close to the largest normal number. This is even more the case for the IEEE 64-bit system.

1.1.6 Exercise 06

Verify that there are $n = 2^m$ normal numbers between any two consecutive powers of 2, in the IEEE system, where the two consecutive numbers are represented as

$$2^p, 2^{p+1}, \qquad e_{\min+1} \leq p \leq e_{\max} - 1.$$

In particular, find out how many normal numbers exist between 2 and 4 in the IEEE 32-bit and 64-bit systems.

SOLUTION

The number of normal numbers between any two consecutive powers of 2 is determined by the number of bits in the significand, and it is independent from the specific value of p. This is due to the fact that each bit of the significand can only take two values, 0 and 1. Therefore, between any two consecutive powers of 2 there will be 2^m normal numbers. For the 32-bit system there will be $2^{23} = 8388608$ (millions) numbers, while for the 64-bit system, where $m = 52$, there will be $2^{52} = 4503599627370496$ (million of billions) numbers. In both cases we are excluding the largest power of 2, i.e. 2^{p+1}.

1.1.7 Exercise 07

Represent $\sqrt{2}$ and $\sqrt{3}$ in the IEEE 8-bit toy system. For both numbers, calculate the absolute and relative errors, due to round off.

SOLUTION

Let us start with $\sqrt{2} \approx 1.414213562$. First, the number is transformed into a binary number. The integer is 1, and this corresponds to 1 in base 2 too. For the decimal part we have:

0.4144213562	×	2	=	0.8288427124	take integer :	0	value for next line :	0.8288427124
0.8288427124	×	2	=	1.6576854248	take integer :	1	value for next line :	0.6576854248
0.6576854248	×	2	=	1.3153708496	take integer :	1	value for next line :	0.3153708496
0.3153708496	×	2	=	0.6307416992	take integer :	0	value for next line :	0.6307416992
0.6307416992	×	2	=	1.2614833984	take integer :	1	value for next line :	0.2614833984
0.2614833984	×	2	=	0.5229667968	take integer :	0	value for next line :	0.5229667968
0.5229667968	×	2	=	1.0459335936	take integer :	1	value for next line :	0.0459335936

which yields

$$\sqrt{2} \approx 1.0110101.$$

The algorithm for conversionn does not stop in this case because an irrational number has an infinite number of digits even in base 10. It is important, though, to go beyond the number m of the significand, in order to apply the rules for rounding correctly. The fourth bit after the floating point is rounded, in this case, to 1. The result is the following number in the 8-bit representation:

$$\sqrt{2} \ \rightarrow \ 1.0111 \times 2^0 \ \rightarrow \ 0 \ 011 \ 0111 = 1.4375$$

Therefore, $\sqrt{2}$ is represented by 1.4375, in the 8-bit system. The absolute error due to this representation (round off error) is

$$E_a = |x - \tilde{x}| \approx |1.4144 - 1.4375| = 0.0231.$$

The relative error is

$$E_r = |x - \tilde{x}|/|x| \approx 0.0231/1.4144 \approx 0.0163.$$

This is, as expected, smaller than the machine precision ϵ_{mach} which, for the 8-bit system, is $2^{-m} = 2^{-4} = 0.0625$.

1.1.8 Exercise 08

Find the sum and difference of the two base 2 numbers, $110.011, 110.010$, once they have been represented in the IEEE 8-bit toy system. Work out the absolute and relative errors of both the addition and subtraction.

SOLUTION

The two numbers are

$$110.011 = 6.375, \quad 110.010 = 6.25.$$

Therefore, the correct results for sum and difference are

$$110.011 + 110.010 = 6.375 + 6.25 = 12.625, \quad 110.011 - 110.010 = 6.375 - 6.25 = 0.125.$$

These operations can be carried out without passing through the base 10 system, and we will proceed this way after having transformed preliminarly the numbers in the 8-bit system.

The first number cannot be represented exactly:

$$110.011 \;\rightarrow\; 1.1010 \times 2^2 \quad \text{corresponding to } 6.5$$

The second number can be represented exactly:

$$110.010 \;\rightarrow\; 1.1001 \times 2^2 \quad \text{corresponding to } 6.25.$$

We expect, therefore, the sum to give 12.75 and the difference to give 0.25. Indeed,

$$\begin{array}{rcll} 1.1010 & \times & 2^2 & + \\ 1.1001 & \times & 2^2 & = \\ \hline 11.0011 & \times & 2^2. & \end{array}$$

This number is equal to 1100.11, which is 12.75 in base 10, as expected. For the difference we have:

$$\begin{array}{rcll} 1.1010 & \times & 2^2 & - \\ 1.1001 & \times & 2^2 & = \\ \hline 0.0001 & \times & 2^2. & \end{array}$$

This number is equal to 0.01, which is 0.25 in base 10, also as expected.

The absolute errors for both results are:

$$E_a = |12.625 - 12.75| = 0.125, \quad E_a = |0.125 - 0.25| = 0.125.$$

The corresponding relative errors are:

$$E_r = E_a/|12.625| \approx 0.00990, \quad E_r = E_a/|0.125| = 1.$$

The relative error is significantly high for subtractions (here 100%), when the two numbers are similar in magnitude, as it is the case in this exercise.

1.1.9 Exercise 09

Calculate the absolute and relative error when $\sin(x)$ is replaced by the truncated expansion

$$\sin(x) \approx x - x^3/6,$$

for $x = 0.1$ and $x = \pi/6$ (all values in radians).

SOLUTION

If $y = \sin(x)$ and $\tilde{y} = x - x^3/6$, the absolute and relative errors are

$$E_a = |y - \tilde{y}| = |\sin(x) - x + x^3/6|, \quad E_r = E_a/|y| = |\sin(x) - x + x^3/6|/|\sin(x)|.$$

For $x = 0.1$ the values are

$$E_a = |\sin(0.1) - 0.1 + 0.1^3/6| \approx 0.000000083$$

and

$$E_r = E_a/|\sin(0.1)| = 0.000000835$$

Such a small value for E_r means that calculations with $x - x^3/3$ replacing $\sin(x)$ will be accurate when x is close to 0.1 radians. For $x = \pi/6$ (corresponding to 30 degrees), the truncation does not yield a similar precision, but it is still advantageous:

$$E_a = |\sin(\pi/6) - \pi/6 + (\pi/6)^3/6| \approx 0.000325821$$

and

$$E_r = E_a/|\sin(\pi/6)| = E_a/0.5 \approx 0.000651641$$

2 Chapter 02

2.1 Exercises on R vectors

2.1.1 Exercise 01

Generate and display:

1. A vector of values $-2, -1, 0, 1, 2, 3, 4, 5, 6$.
2. A vector of values $10, 9, 8, 7, 6$.
3. A vector of values $0, 0.5, 1, 1.5, 2, 2.5, 3$, starting from the vector with values $0, 1, 2, 3, 4, 5, 6$.

SOLUTION

1. As the interval between a number in the series of components is of length 1, we can use the R syntax `Vini:Vfin`, where `Vini` and `Vfin` are the initial and final values of the series.

```
Vini <- 0
Vfin <- 6
V <- Vini:Vfin
print(V)
#> [1] 0 1 2 3 4 5 6
```

2. The series goes backward, but the interval between values has still length 1. Therefore, we can use the `Vini:Vfin` syntax here too.

```
Vini <- 10
Vfin <- 6
V <- Vini:Vfin
print(V)
#> [1] 10  9  8  7  6
```

3. The series required can be obtained from the first one, simply dividing, elementwise, by 2.

```
# First series
W <- 0:6

# Second series
V <- W/2
print(V)
#> [1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0
```

2.1.2 Exercise 02

Generate a regular grid between $-\pi$ and $+\pi$ of 100 points. Find out the length Δx between two contiguous points of the grid.

SOLUTION

A regular grid can be generated between any two values x_L, x_R, with $x_R > x_L$, by simply requesting to the function `seq` the number of grid points needed. The function works out the regular spacing by itself. For the specific example suggested:

```
# Extremes of the interval
xL <- -pi
xR <- pi

# Grid
xgrid <- seq(xL,xR,length=100)
```

```
# First three grid points
print(xgrid[1:3])
#> [1] -3.141593 -3.078126 -3.014660

# Spacing of the grid
print(xgrid[2]-xgrid[1])
#> [1] 0.06346652
```

The length requested is $\Delta x \approx 0.06346652$.

2.1.3 Exercise 03

Create the following pattern,

1 1 1 2 2 3

using function `rep`.

SOLUTION

The three numbers repeated are $1, 2, 3$. The first is repeated three times, the second tw times, and the third just one time. Thus:

```
x <- rep(c(1,2,3),times=c(3,2,1))
print(x)
#> [1] 1 1 1 2 2 3
```

2.1.4 Exercise 04

Create the numeric pattern,

1 2 3 2 2 1 2 3 2 2

using function `rep`.

SOLUTION

This is a two-times repetition of 1 2 3 2 2. Therefore:

```
x <- rep(c(1,2,3,2,2),times=2)
print(x)
#>  [1] 1 2 3 2 2 1 2 3 2 2
```

2.1.5 Exercise 05

Create the numeric pattern,

1 1 1 2 2 3 1 1 1 2 2 3 1 1 1 2 2 3

using function `rep`.

SOLUTION

This is a repetition, three-times, of the same pattern of Exercise 03. We can therefore use `rep` in a nested way.

```
x <- rep(rep(c(1,2,3),times=c(3,2,1)),times=3)
print(x)
#>  [1] 1 1 1 2 2 3 1 1 1 2 2 3 1 1 1 2 2 3
```

2.1.6 Exercise 06

Create a vector `x` of length 30 using the following expression:

```
set.seed(123)
x <- sample(seq(0,1,length=101),size=30,replace=TRUE)
```

Print the value of the 9th, 18th and 27th component of the vector `x`. The `set.seed` function fixes the generation of the pseudo-random numbers so that the above code will always output the same numbers.

SOLUTION

This is the code involved, in one chunk.

```
# Creation of the vector
set.seed(123)
x <- sample(seq(0,1,length=101),size=30,replace=TRUE)

# Access and print value of 9th, 18th and 27th component
print(x[c(9,18,27)])
#> [1] 1.00 0.08 0.35
```

Therefore, the components requested are 1.00, 0.08 and 0.35.

2.1.7 Exercise 07

Consider the vector `x` with the following components:

$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$$

Select and print only the components of `x` which contain even numbers. Then print the other components.

SOLUTION

The *filtered* selection of a vector's components can be carried out using appropriate indices. For example, `x[c(2,4,6)]` selects the second, fourth and sixth component of `x`, while `x[-1]` selects all components of `x` with the exception of the first. For the exercise given here, we can create a set of indices with only odd integers, and use them both as they are and with a minus sign to carry out the filtered selection.

```
# Create vector
x <- 0:9

# Index for direct and filtered selection
idx <- c(1,3,5,7,9)

# Straight selection
print(x[idx])
#> [1] 0 2 4 6 8

# Filtered selection
print(x[-idx])
#> [1] 1 3 5 7 9
```

2.1.8 Exercise 08

Using a regular grid `x` of 361 values in the range $[0, 2\pi]$, calculate the corresponding values of the function

$$2\sin(x) - \cos(x),$$

and store them in a vector called `y`. Find the indices of `x` corresponding to 0, π and 2π, and verify that the values of `y` at these positions are -1, 1 and -1.

SOLUTION

The grid suggested essentially covers the whole range between 0 degrees and 360 degrees, with each grid point associated to an integer degree. So the grid starts as $0, 1, 2, 3, \ldots$, where these values have to be transformed in radians.

```
# Grid (in radians)
x <- seq(0,2*pi,length=361)
print(x[1:5])
#> [1] 0.00000000 0.01745329 0.03490659 0.05235988 0.06981317

# Each grid point is one degree (remember the conversion)
print(x[1:5]*180/pi)
#> [1] 0 1 2 3 4
```

We can, next, calculate the values, bearing in mind that operations on vectors act in a parallel fashion.

```
# Values of function at grid points
y <- 2*sin(x)-cos(x)

# Print first few values
print(y[1:5])
#> [1] -1.0000000 -0.9649429 -0.9295918 -0.8939576 -0.8580511
```

The indices of the values to check suggested are easily derivable from the fact that the grid has a regular step size of one degree. Therefore 0 is associated with component 1, π, which is half-way in the full range, is associated with component 181 and 2π with component 361.

```
# Values requested
idx <- c(1,181,361)
print(y[idx])
#> [1] -1  1 -1
```

They are, indeed, equal to $-1, 1, -1$, as expected.

2.1.9 Exercise 09

Consider the two vectors `x` and `y` of different lengths:

$$x: \quad 1\ 2\ 3\ 4\ 5, \qquad y: \quad 2\ 4$$

What do you expect the result of `x+y` to be? Justify your answer and verify its correctness with R.

***SOLUTION**

As vector `y` is shorter than vector `x`, it will be recycled to match the length of `x`. As this last vector has length 5, recycling `y` yields the vector

$$2\ 4\ 2\ 4\ 2$$

Therefore we expect `x+y` to return

$$1+2=3,\ 2+4=6,\ 3+2=5,\ 4+4=8,\ 5+2=7$$

Indeed:

```
x <- 1:5
y <- c(2,4)
print(x+y)
#> Warning in x + y: longer object length is not a multiple of shorter object
#> length
#> [1] 3 6 5 8 7
```

The warning is automatically returned by R to make sure the user is conscious about the recycling of components.

2.2 Exercises on simple graphics

2.2.1 Exercise 10

Using the function `plot`, draw an empty square with vertices at $(0,0)$, $(0,1)$, $(1,1)$ and $(1,0)$, in black. Draw also the two diagonals in red.

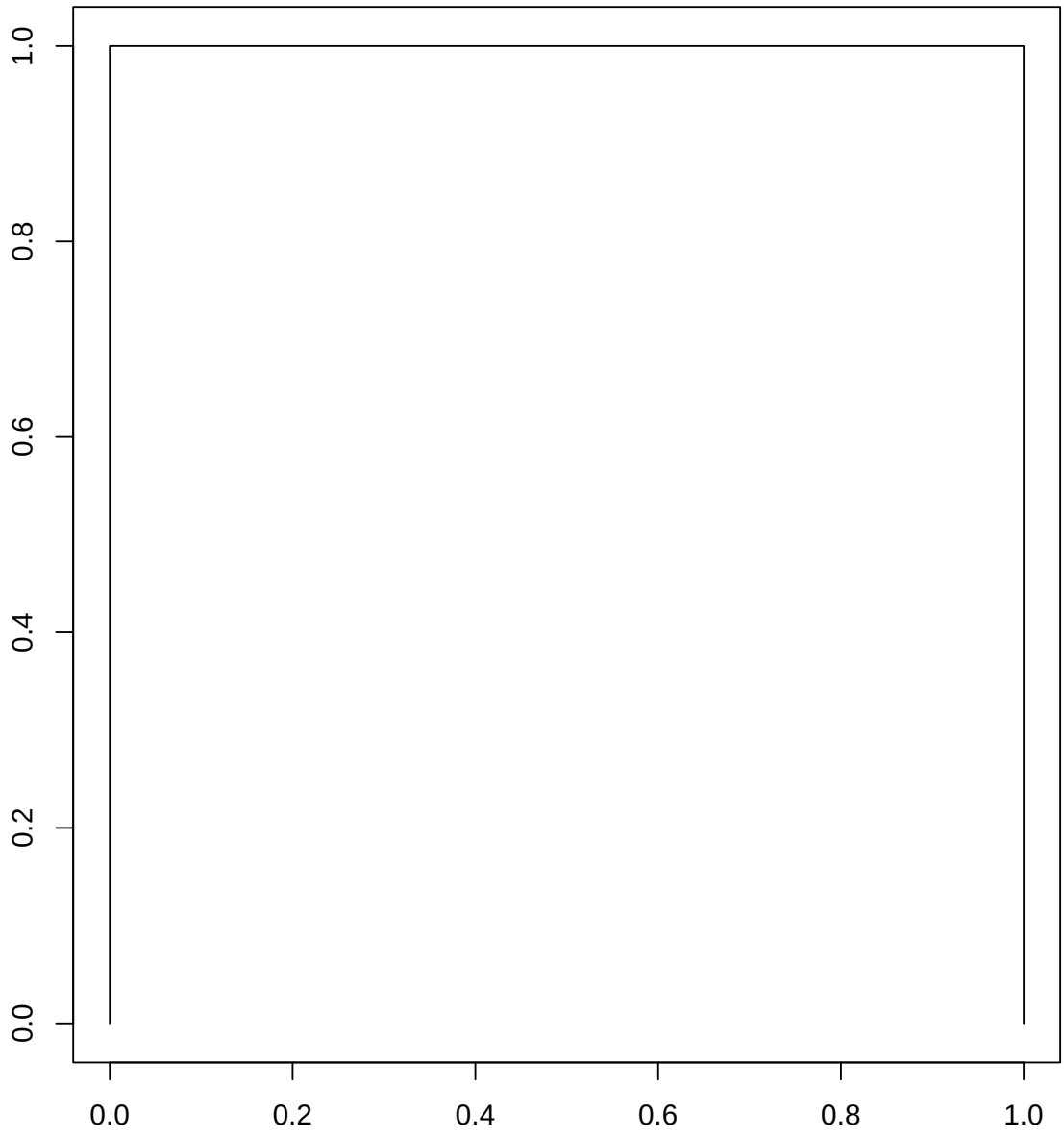
Repeat the same pattern two times, using function `rect` first, and `polygon` second. In both cases, the diagonals can be drawn using the function `segments`.

Make sure to use inline help and/or information on the internet for all functions needed.

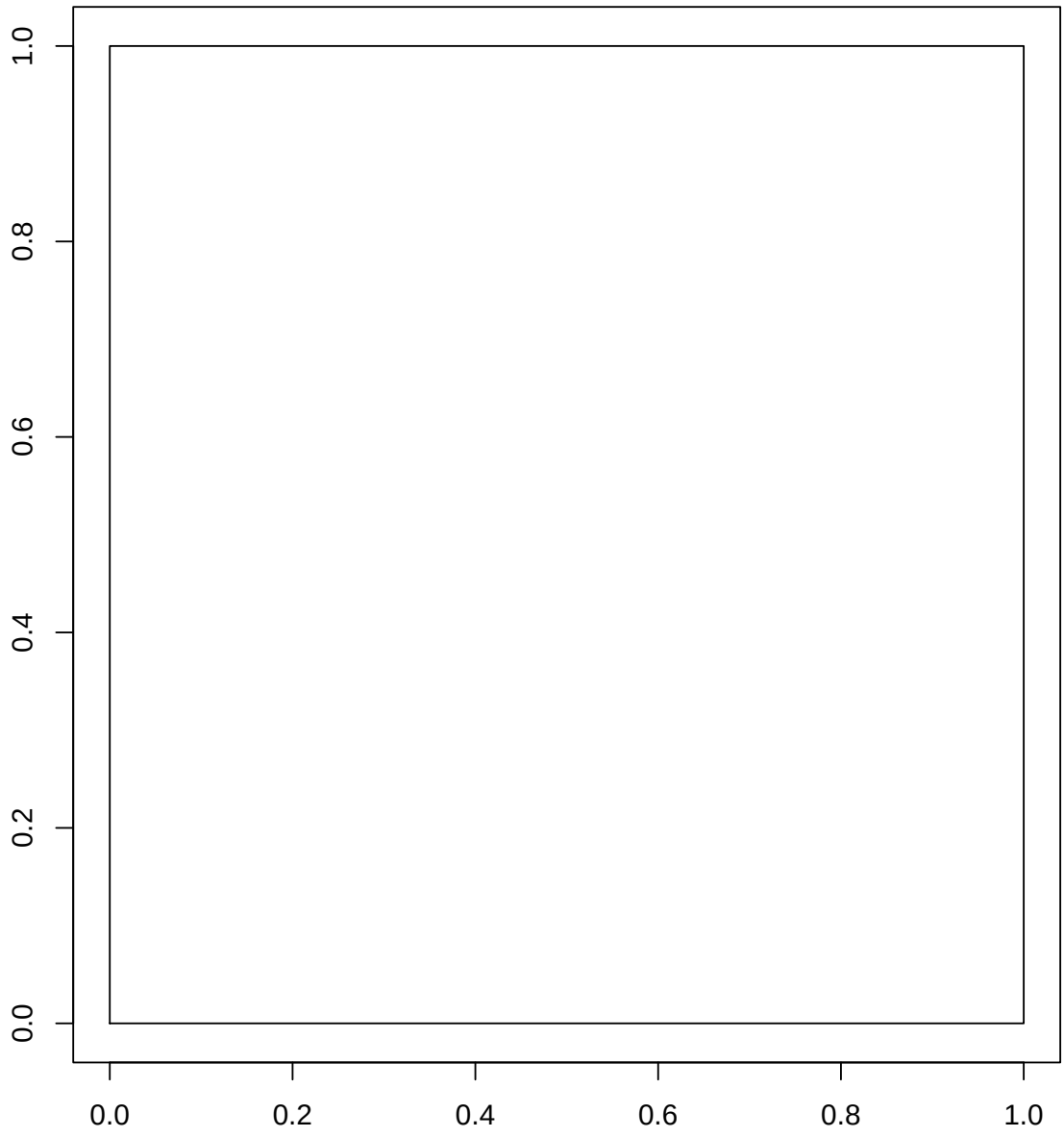
SOLUTION

When using `plot`, we need to decompose the box into four lines. Each line is a straight segment between two points. We can therefore use `plot` with just four points and with `type` set to `l` ("el' ').

```
# x and y are the coordinates of the four points
# They need to be in the same order
x <- c(0,0,1,1)
y <- c(0,1,1,0)

# Now plot with type="l". Surprise: it's open
plot(x,y,type="l",xlab="",ylab="")
```



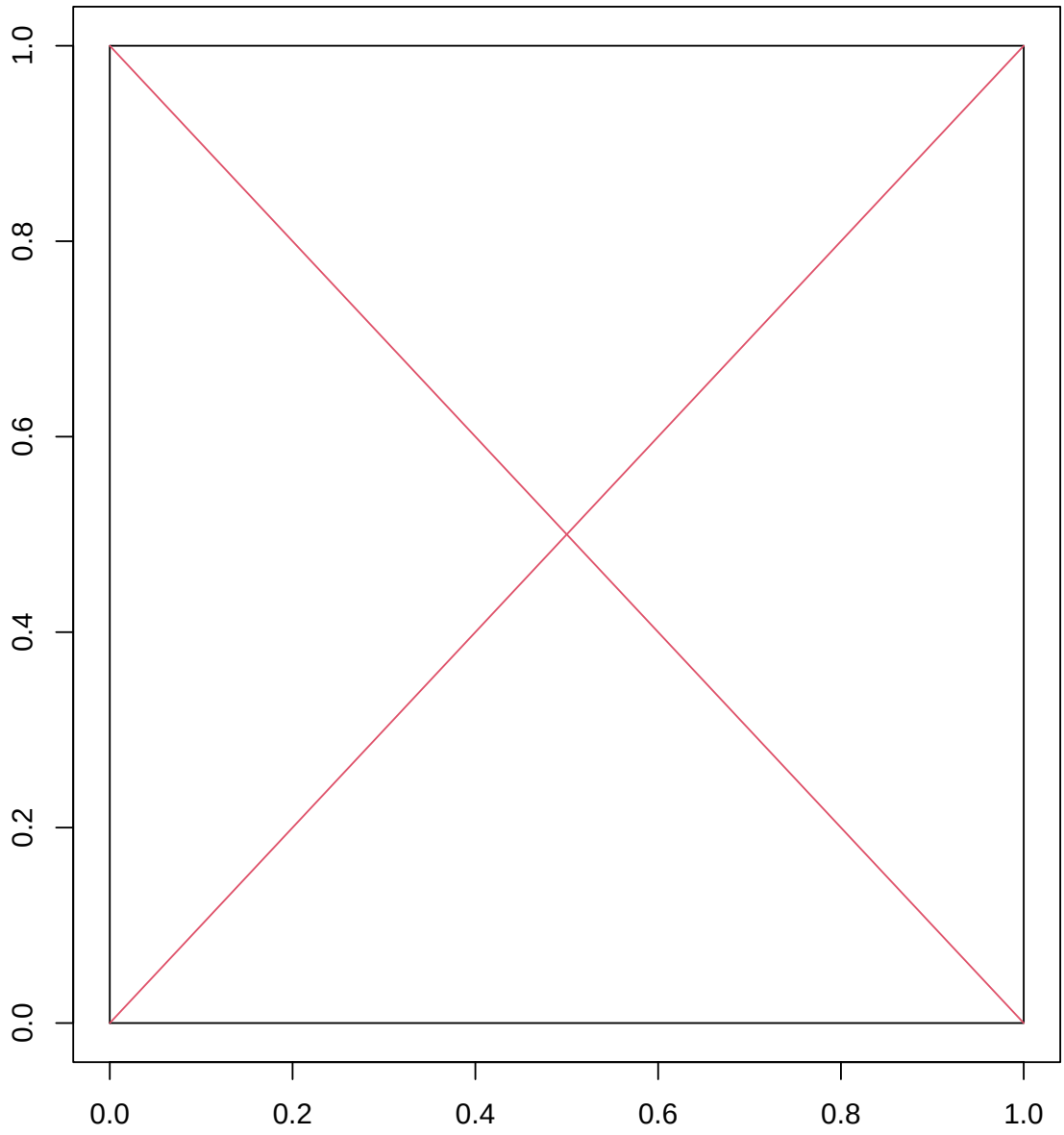

```r
# To close the square, repeat first point after the last
x <- c(0,0,1,1,0)
y <- c(0,1,1,0,0)
plot(x,y,type="l",xlab="",ylab="")
```



Finally, we can add the *cross* in red using the same concept of points joined by segments.

```
# Repeat what done earlier
x <- c(0,0,1,1,0)
y <- c(0,1,1,0,0)
plot(x,y,type="l",xlab="",ylab="")

# Plot each segment of cross, in turn
# Notice that as the plot has been already created,
```

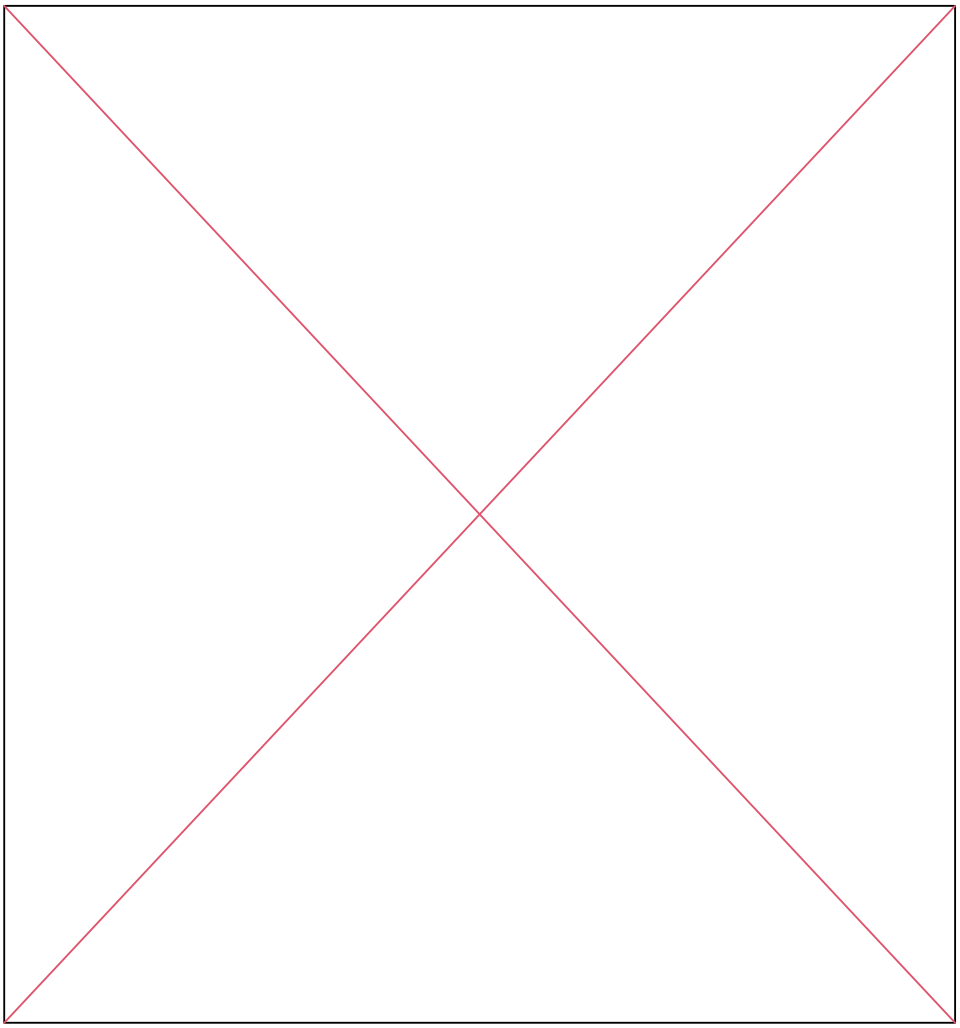
```r
# we use "points", rather than "plot"
points(x=c(0,1),y=c(0,1),type="l",col=2)
points(x=c(1,0),y=c(0,1),type="l",col=2)
```



A square can also be drawn using the function `rect`. A rectangle with sides parallel to the x-axis and y-axis is completely defined once the bottom-left and top-right corners are defined. The colour of the border of the rectangle is given by `border`. A plot should already be present before using `rect`. This is accomplished using `plot.new`. Observe that in this case the plot is not annotated. The segments for the cross can also be

produced using `segments`.

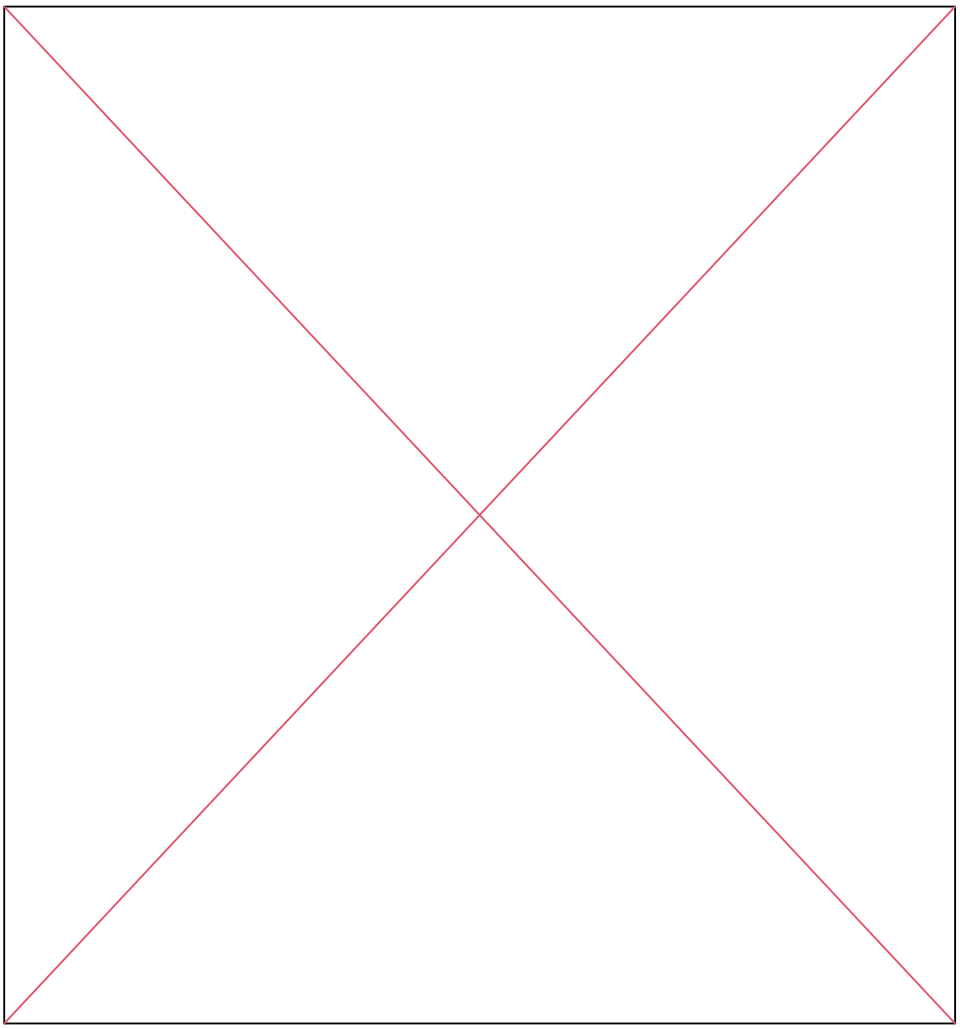
```
# Create an empty plot
plot.new()

# The whole square using "rect"
rect(xleft=0,ybottom=0,xright=1,ytop=1,border="black")

# Red cross
segments(x0=0,y0=0,x1=1,y1=1,col=2)
segments(x0=1,y0=0,x1=0,y1=1,col=2)
```



Everything can be repeated using `polygon`, which can be used only if a plot already exists (one can use `plot.new` for that). This acts similar to `plot`, the way we have used it to draw the rectangle. We can then add the cross using, again, `segments`.

```
# Create an empty plot
plot.new()

# Polygon (in this case a square)
polygon(x=c(0,0,1,1,0),y=c(0,1,1,0,0),border="black")
```

```r
# Red cross
segments(x0=0,y0=0,x1=1,y1=1,col=2)
segments(x0=1,y0=0,x1=0,y1=1,col=2)
```



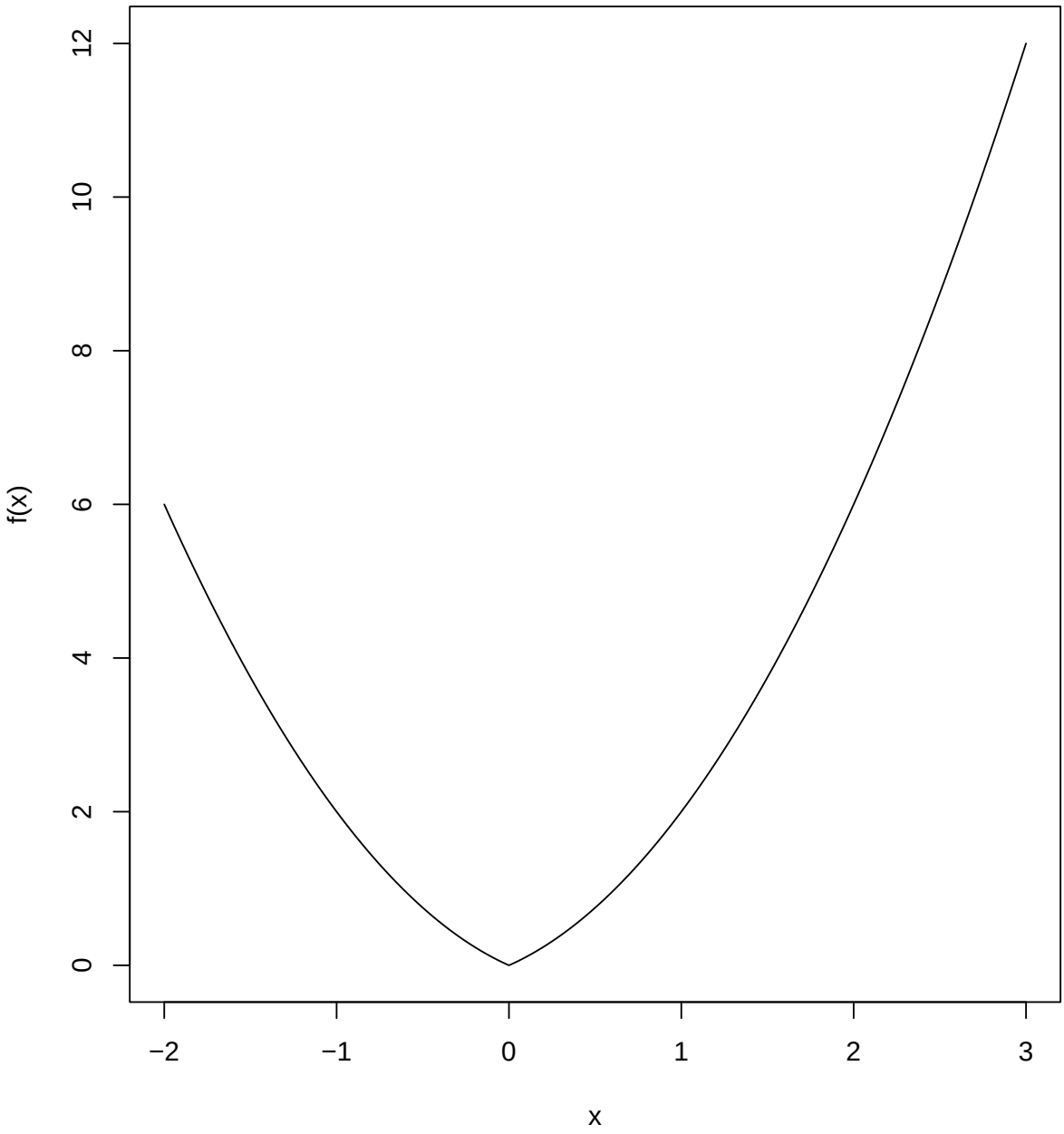
2.2.2 Exercise 11

Plot the function $f(x) = |x| + x^2$ in the interval $x \in [-2, 3]$, on two separate plots using, respectively, `plot` and `curve`. For the second plot, use a green, dashed line.

SOLUTION

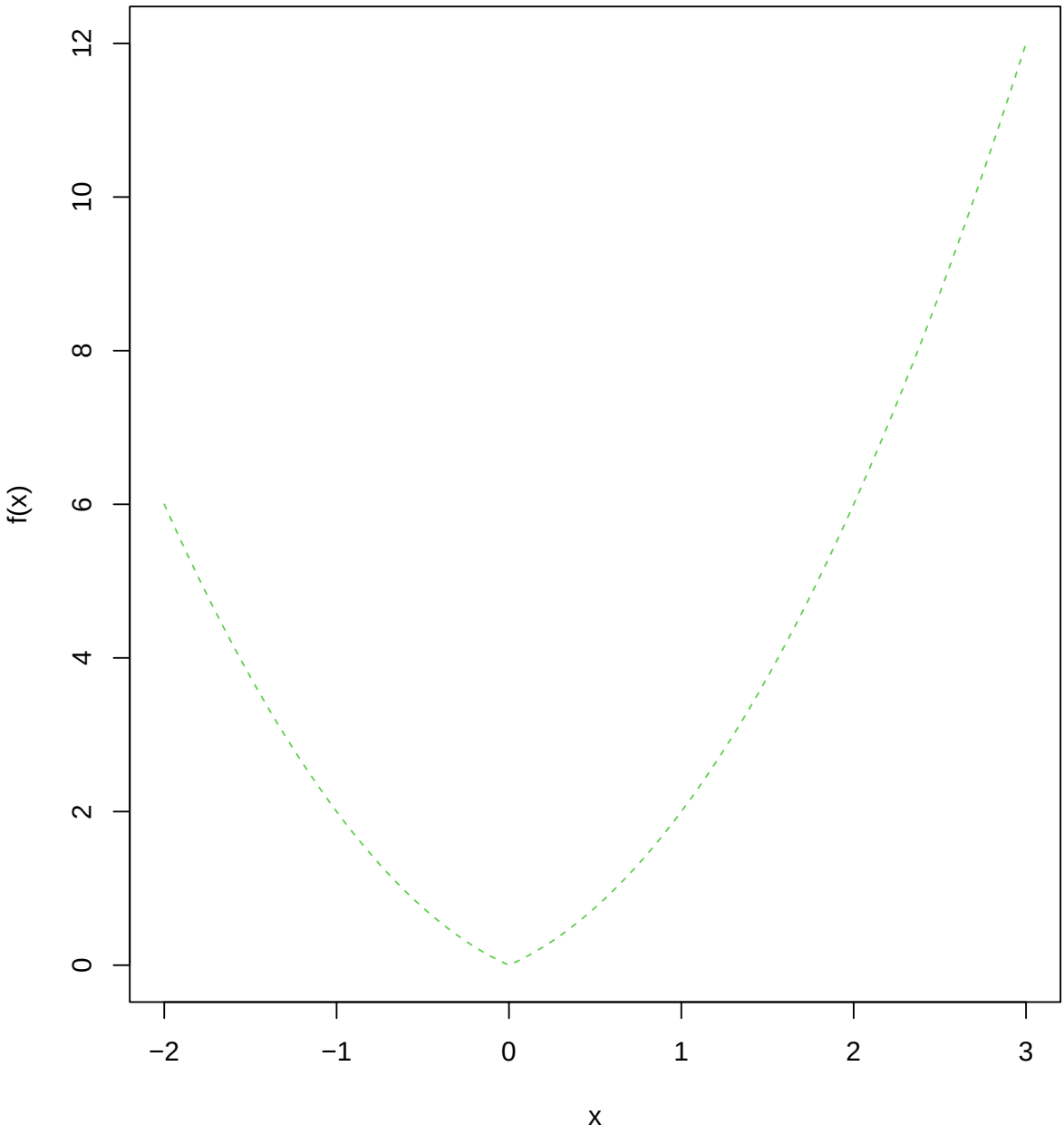
Without `curve`, we need first to create a grid in $[-2, 3]$ and calculate the function at all grid points.

```
# x grid
x <- seq(-2,3,length=1000)

# Values of function
y <- abs(x)+x^2

# Plot
plot(x,y,type="l",xlab="x",ylab="f(x)")
```



With `curve` there is no need to create the grid.

```
# For a gree dashed line we use "col" and "lty"
curve(abs(x)+x^2,from=-2,to=3,col=3,lty=2,xlab="x",ylab="f(x)")
```



2.2.3 Exercise 12

In the interval $x \in [-\pi, \pi]$, draw the following three experimental points,

x	$y = f(x)$
$-\pi$	-1
0	0
π	1

as black upward triangles and the following three curves:

a. $f(x) = \sin(x)$, in red

b. $f(x) = x^3 + (1/\pi + \pi^2)x$, in green

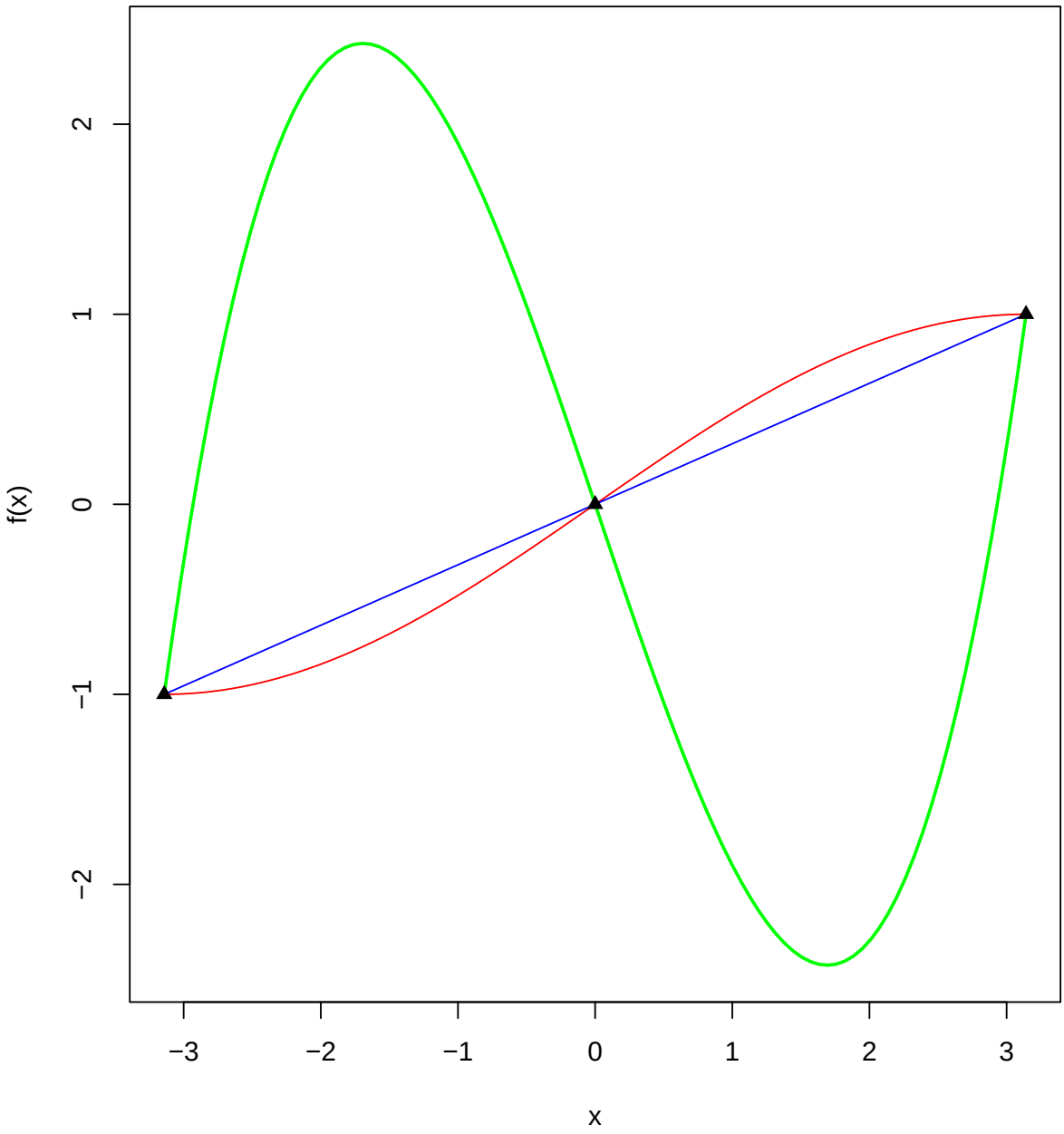
c. $f(x) = x/\pi$, in blue.

Make sure that the second curve as a line twice as thick as the other two curves.

SOLUTION

We can plot the curves with `curve` and add the triangles with `points`. But we first need to know the range of the three functions, so to represent all of them completely within one plot.

```
# Range of three functions
x <- seq(-pi,pi,length=1000)
y1 <- sin(x/2)
y2 <- 0.25*x^3+(1/pi-0.25*pi^2)*x
y3 <- x/pi
yy <- range(y1,y2,y3)

# Use "curve" to draw the functions
curve(sin(x/2),from=-pi,to=pi,col="red",xlab="x",ylab="f(x)",ylim=yy)
curve(0.25*x^3+(1/pi-0.25*pi^2)*x,from=-pi,to=pi,col="green",lwd=2,add=TRUE)
curve(x/pi,from=-pi,to=pi,col="blue",add=TRUE)

# Add points - The triangle is pch=17
points(c(-pi,0,pi),c(-1,0,1),pch=17,col="black")
```



2.3 Exercises on R functions

2.3.1 Exercise 13

Write a function with input `n` that outputs the first n integers as a vector.

SOLUTION

As the input is `n` the function's definition will contain `function(n)`. A simple implementation of the function requested is:

```r
# Definition
f <- function(n) {
  v <- 1:n

  return(v)
}
```

This is then easy to use:

```r
# Generate and print vector of first 3 numbers
v <- f(3)
print(v)
#> [1] 1 2 3

# Generate and print vector of first 5 numbers
v <- f(5)
print(v)
#> [1] 1 2 3 4 5
```

2.3.2 Exercise 14

Modify the previous function (remember to change the name in order to preserve the previous function) to create a vector between two integer numbers.

SOLUTION

This task can be achieved using `n1:n2` rather than `1:n`.

```r
# New function
g <- function(n1,n2) {
  v <- n1:n2

  return(v)
}

# Use the function
v <- g(-3,5)
print(v)
#> [1] -3 -2 -1  0  1  2  3  4  5
```

2.3.3 Exercise 15

Write a function `spc` to sum the sine and cosine of any angle `x` and a function `smc` that calculates the difference between the sine and cosine of any angle `x`. Finally, write a function that calculates the ratio

$$T(x) = \frac{\sin(x) + \cos(x)}{\sin(x) - \cos(x)},$$

for any given angle. Verify that the result is identical to the one returned by the function

$$\frac{\tan(x) + 1}{\tan(x) - 1}.$$

Finally, use the input `x <- c(0,pi/6,pi/4,pi/3,pi/2)` in the last function. What do you think is happening?

SOLUTION

The creation of the first two functions is implemented in the following code:

```r
# Function spc
spc <- function(x) {
  y <- sin(x)+cos(x)

  return(y)
}

# Function smc
smc <- function(x) {
  y <- sin(x)-cos(x)

  return(y)
}
```

These two functions, like any new function, must be tested to check they return the expected output. We can try them now on an angle, $\pi/6$, for which it is easy to calculate both the sine and the cosine. In the first case we expect:

$$\sin\left(\frac{\pi}{6}\right)+\cos\left(\frac{\pi}{6}\right)=\frac{1}{2}+\frac{\sqrt{3}}{2}=\frac{\sqrt{3}+1}{2}\approx 1.366$$

and

$$\sin\left(\frac{\pi}{6}\right)-\cos\left(\frac{\pi}{6}\right)=\frac{1}{2}-\frac{\sqrt{3}}{2}=\frac{1-\sqrt{3}}{2}\approx -0.366.$$

```r
# Test on x=pi/6
print(spc(pi/6))
#> [1] 1.366025
print(smc(pi/6))
#> [1] -0.3660254
```

So, the function defined, work. Next, we define a new function, `T`, that uses the previously-defined functions. As these are present in the working memory, they will be recognised:

```r
# New (ratio) function
T <- function(x) {
  y <- spc(x)/smc(x)

  return(y)
}
```

To test this function we can still use the angle $\pi/6$. We should have:

$$\frac{\sin(\pi/6)+\cos(\pi/6)}{\sin(\pi/6)-\cos(\pi/6)}=\frac{(1+\sqrt{3})/2}{(1-\sqrt{3})/2}=\frac{1+\sqrt{3}}{1-\sqrt{3}}\approx -3.732.$$

Indeed:

```r
# Test with x=pi/6
print(T(pi/6))
#> [1] -3.732051
```

So, this function works. Incidentally, dividing the original expression with sines and cosines by $\cos(x)$, we obtain an expression with tangents:

$$\frac{\sin(x)+\cos(x)}{\sin(x)-\cos(x)}=\frac{\tan(x)+1}{\tan(x)-1}.$$

This should give us again the same result when $x=\pi/6$:

```
# Use tangents
print((tan(pi/6)+1)/(tan(pi/6)-1))
#> [1] -3.732051
```

Let us complete the exercise on the values given:

```
# Values given
x <- c(0,pi/6,pi/4,pi/3,pi/2)

# Feed them to T
y <- T(x)

# Print
print(y)
#> [1] -1.000000e+00 -3.732051e+00 -1.273810e+16  3.732051e+00  1.000000e+00
```

The function built acts in parallel fashion on the five values provided. The third value returned is particularly high; this is in line with $\sin(\pi/4) - \cos(\pi/4)$ being zero and the ratio being, accordingly, infinity. The function has not returned `Inf` because `pi/4` is stored with finite precision.

2.4 Exercises on R objects and data handling

2.4.1 Exercise 16

a. Create the following objects in R:

```
a <- 42
b <- "42"
c <- TRUE
d <- charToRaw("Hello")
e <- as.raw(c(0x48, 0x65, 0x6C, 0x6C, 0x6F))
```

b. Use `typeof` to determine the storage type of each object. What is the difference between `a` and `b`? What is the difference between `d` and `e`? For the raw objects `d` and `e`: convert them back to a character string using `rawToChar`.

c. Use `typeof` on a data frame and on one of its columns. Why are the results different? What does this tell you about how R stores data?

SOLUTION

a. The first task is easily carried out simply by typing in all variables in a console:

```
# Clear workspace to check next set of objects is created
# (This is an interesting way of doing it)
rm(list=ls(all=TRUE))
ls()
#> character(0)

# Create objects
a <- 42
b <- "42"
c <- TRUE
d <- charToRaw("Hello")
e <- as.raw(c(0x48, 0x65, 0x6C, 0x6C, 0x6F))
ls()
#> [1] "a" "b" "c" "d" "e"
```

b. Check each type:

```
# a
print(typeof(a))
#> [1] "double"

# b
print(typeof(b))
#> [1] "character"

# c
print(typeof(c))
#> [1] "logical"

# d
print(typeof(d))
#> [1] "raw"
print(d)
#> [1] 48 65 6c 6c 6f

# e
print(typeof(e))
#> [1] "raw"
print(e)
#> [1] 48 65 6c 6c 6f
```

So, although `a` and `b` hve been typed with a same '`42''`, the first is a number in double precisionwhile the second is a character; they are two completely different objects.cis clearly a logical object. The last two are raw objects. By printing them out one can see their hexadecimal representation, which coincides. So, even if generated differently,dandecontain the same raw data. Let try and convert them to characters using the built in functionrawToChar':

```
# Convert raw data to character strings
Cd <- rawToChar(d)
print(Cd)
#> [1] "Hello"
Ce <- rawToChar(e)
print(Ce)
#> [1] "Hello"
```

Being `d` and `e` the same raw data, when converted they produce the same character string, the word "Hello' '.

c. Here we could use one of the existing data frames, say `mtcars`.

```
# typeof applied to data frame
print(typeof(mtcars))
#> [1] "list"

# mtcars column name
print(colnames(mtcars))
#>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#> [11] "carb"

# typeof applied to data frame columns
print(typeof(mtcars$mpg))
#> [1] "double"
```

Thus, a data frame is a list with some attributes that make it a data frame. The command `typeof` in essence

reveals how data are stored. So, data are stored as in a list for the whole data frame, but as numbers for one of its columns.

2.4.2 Exercise 17

Type in the following (exact) syntax in a console:

```
G <- list(mtcars,diag(5),solve(diag(5)),LETTERS[1:10])
```

a. What is the `typeof` for `G` and `G[[1]]`? Explain the difference.

b. What data structure have `G[[2]]` and `G[[3]]`? Print the objects to verify this is true.

c. What is the data structure and data type of `G[[4]]`?

SOLUTION

Typing is straightforward:

```
# Clear workspace to check next set of objects is created
rm(list=ls(all=TRUE))
ls()
#> character(0)

# Create object provided
G <- list(mtcars,diag(5),solve(diag(5)),LETTERS[1:10])
ls()
#> [1] "G"
```

a. Data types are assessed with `typeof`:

```
# Data type of G
print(typeof(G))
#> [1] "list"

# Data type of G[[1]]
print(typeof(G[[1]]))
#> [1] "list"
```

Both objects are lists. But the second is, in fact, a built-in data frame, `mtcars`, which is recognised as a list data type because all data frames are lists with additional attributes.

```
# Attributes reveal internal structure
print(attributes(G))
#> NULL
print(attributes(G[[1]]))
#> $names
#>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#> [11] "carb"
#>
#> $row.names
#>  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
#>  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
#>  [7] "Duster 360"          "Merc 240D"           "Merc 230"
#> [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
#> [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
#> [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
#> [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
#> [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
```

```
#> [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
#> [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
#> [31] "Maserati Bora"       "Volvo 142E"
#>
#> $class
#> [1] "data.frame"
```

So, `G` has no attributes, it's a pure list. Differently, one of the attributes of `G[[1]]` is `class` which, in this specific instance, '`classify''`mtcars' as a data frame.

b.

```
# Data structures are revealed by str
# G[[2]]
str(G[[2]])
#>  num [1:5, 1:5] 1 0 0 0 0 0 1 0 0 0 ...

# G[[3]]
str(G[[3]])
#>  num [1:5, 1:5] 1 0 0 0 0 0 1 0 0 0 ...
```

So, both objects are 2D arrays (matrices), containing numbers. Being a '`matrix'' is made possible by the attribute`dim':

```
# Attributes of G[[2]] and G[[3]]
print(attributes(G[[2]]))
#> $dim
#> [1] 5 5
print(attributes(G[[3]]))
#> $dim
#> [1] 5 5

# Print actual objects
print(G[[2]])
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    0    0    0    0
#> [2,]    0    1    0    0    0
#> [3,]    0    0    1    0    0
#> [4,]    0    0    0    1    0
#> [5,]    0    0    0    0    1
print(G[[3]])
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    0    0    0    0
#> [2,]    0    1    0    0    0
#> [3,]    0    0    1    0    0
#> [4,]    0    0    0    1    0
#> [5,]    0    0    0    0    1
```

These two matrices are identical, the 5×5 identity matrix, I_5. The function `solve`, when applied to a matrix, returns its inverse, and the inverse of I_5 is still I_5.

c. Let us explore `G[[4]]` now:

```
print(typeof(G[[4]]))
#> [1] "character"
print(attributes(G[[4]]))
#> NULL
```

```
# What's in G[[4]]?
print(G[[4]])
#>  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

`G[[4]]` is a vector of characters. As all vectors, it does not have attributes, to start with. In fact, the object `LETTERS` is a built-in vector containing the whole alphabet in uppercase:

```
# Uppercase alphabet
print(LETTERS)
#>  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
#> [20] "T" "U" "V" "W" "X" "Y" "Z"
```

2.4.3 Exercise 18

Select a built-in data frame different from **mtcars** and transform it into a matrix. Is it possible straight away? Or, you need to take some decisions on the variables contained, first?

SOLUTION

One of the data frames seen earlier is **iris**. It includes five columns, the last one being a column of factors.

```
# Data frame iris
str(iris)
#> 'data.frame':    150 obs. of  5 variables:
#>  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
#>  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
#>  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
#>  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

unique(iris$Species)
#> [1] setosa     versicolor virginica
#> Levels: setosa versicolor virginica
```

There are only three factor levels that could be turned into the integers $1, 2, 3$. We can therefore start by creating a vector of integers to transform the last column of `iris`

```
# Turn factors into integers
new_factors <- as.integer(iris$Species)

# Replace last column with integers (keep iris safe)
M <- cbind(iris[,1:4],new_factors)
str(M)
#> 'data.frame':    150 obs. of  5 variables:
#>  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
#>  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
#>  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
#>  $ new_factors : int  1 1 1 1 1 1 1 1 1 1 ...
```

The new data frame `M` contains only numbers and can be turned into a matrix with one command:

```
# Turn data frame into mtrix
M <- as.matrix(M)
str(M)
#>  num [1:150, 1:5] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  - attr(*, "dimnames")=List of 2
```

```
#>   ..$ : NULL
#>   ..$ : chr [1:5] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" ...
```

There are some left over attributes. What are they?

```
# Attributes of M
print(attributes(M))
#> $dim
#> [1] 150   5
#>
#> $dimnames
#> $dimnames[[1]]
#> NULL
#>
#> $dimnames[[2]]
#> [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "new_factors"
```

Besides the `dim` attribute that characterises a matrix, there is another attribute, `dimnames`, which is a left over from when `M` was a data frame. We can get rid of it:

```
# Get rid of unnecessary attributes
attributes(M)$dimnames <- NULL

# New M (a pure matrix)
str(M)
#>  num [1:150, 1:5] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
```

The same task could have been achieved using different procedures but factors and attributes must be always taken into account.

2.4.4 Exercise 19

Create an $2 \times 3 \times 4$ array and fill it naturally (i.e. following the built-in order of your machine) with the first 24 integers.

SOLUTION

An array with three dimensions can be thought as a `cubic matrix'' made of slices, each one of them being a`square matrix''. Filling the array, if no special arrangements are made, follows whatever filling order is hard-coded in the language. Let us fill the array, which we can call `A`, and discover how the filling is done.

```
# Create the array
A <- array(1:24,c(2,3,4))

# Print to discover order of filling
print(A)
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
#> [2,]    8   10   12
```

```
#>
#> , , 3
#>
#>      [,1] [,2] [,3]
#> [1,]   13   15   17
#> [2,]   14   16   18
#>
#> , , 4
#>
#>      [,1] [,2] [,3]
#> [1,]   19   21   23
#> [2,]   20   22   24
```

In order to identify which part of the array is being filled, we can use the letters `i,j,k`. We can then see that the first to be filled is the bottom slice (`k=1`); the filling of this slice/matrix follows the columns order: rows (index `i`) change fast, columns (index `j`) change slow. This means that a different filling order will have to be thought appropriately.

The different filling ordering can be also visualised by extracting the indices from the array, with the command `arrayInd`:

```
# Extract indices from array
idx <- arrayInd(seq_along(A), dim(A))

# Merge these with array content.
# This work because it's a natural filling order
T <- cbind(idx,value=A)
print(T)
#>              value
#>  [1,] 1 1 1     1
#>  [2,] 2 1 1     2
#>  [3,] 1 2 1     3
#>  [4,] 2 2 1     4
#>  [5,] 1 3 1     5
#>  [6,] 2 3 1     6
#>  [7,] 1 1 2     7
#>  [8,] 2 1 2     8
#>  [9,] 1 2 2     9
#> [10,] 2 2 2    10
#> [11,] 1 3 2    11
#> [12,] 2 3 2    12
#> [13,] 1 1 3    13
#> [14,] 2 1 3    14
#> [15,] 1 2 3    15
#> [16,] 2 2 3    16
#> [17,] 1 3 3    17
#> [18,] 2 3 3    18
#> [19,] 1 1 4    19
#> [20,] 2 1 4    20
#> [21,] 1 2 4    21
#> [22,] 2 2 4    22
#> [23,] 1 3 4    23
#> [24,] 2 3 4    24
```

We can appreciate the fastest-changing index `i` (`121212`), the intermediate-changing index `j` (`112233`), and

the slowest-changing index `k` (`111111`).

2.4.5 Exercise 20

Generate an Hermitian matrix. Recall: an Hermitian matrix, H, obeys the property

$$H^\dagger = H,$$

where † indicates a complex-conjugate transpose of the original matrix. This property means that H has real numbers along the diagonal, while for the other components h_{ij} (in general complex numbers), we have:

$$h^*_{ij} = h_{ji}.$$

SOLUTION

One way of creating the Hermitian matrix is indicated here.

```
# Fix random seed for results replication
set.seed(1234)

# Random set of 9 complex numbers with real
# and imaginary parts between -2 and 2
h <- complex(real=sample(-2:2,size=9,replace=TRUE),
             imaginary=sample(-2:2,size=9,replace=TRUE))

# Create a matrix with complex values
A <- matrix(h,ncol=3)

# Hermitian is sum of A and transpose of complex conjugate
H <- 0.5*(A+t(Conj(A)))
print(H)
#>           [,1]      [,2]    [,3]
#> [1,] 1.0+0.0i  0.0+0.5i  1.5+0i
#> [2,] 0.0-0.5i -2.0+0.0i  0.5+1i
#> [3,] 1.5+0.0i  0.5-1.0i -1.0+0i
```

The matrix is Hermitian. Besides having all values along the diagonal as real, all off-diagonal components enjoy $h^*_{ij} = h_{ji}$. For example:

$$h_{23} = \frac{1}{2} + \mathrm{i} \;\Rightarrow\; h^*_{23} = \frac{1}{2} - \mathrm{i} = h_{32}.$$

2.4.6 Exercise 21

A quick way of generating all combinations of finite factors is with the command `expand.grid`. Use the help facility in R to understand how this command works. Then create a data frame for all possible combinations of:

- `colour`: values `"g","r","b"`;
- `size`: values `"xs","s","m","l","xl"`;
- `gender`: value `"m","f"`.

Finally, create a list of the three vectors `colour`, `size`, and `gender`.

SOLUTION

The `expand.grid` command is quite easy to use. Simply, one include all vectors (with relative values) as arguments. In our case:

```r
# Content of data frame
colour <- c("g","r","b")
size <- c("xs","s","m","l","xl")
gender <- c("m","f")

# Generate data frame of all combinations of jumpers
jumpers <- expand.grid(colour=colour,size=size,gender=gender)
str(jumpers)
#> 'data.frame':    30 obs. of  3 variables:
#>  $ colour: Factor w/ 3 levels "g","r","b": 1 2 3 1 2 3 1 2 3 1 ...
#>  $ size  : Factor w/ 5 levels "xs","s","m","l",..: 1 1 1 2 2 2 3 3 3 4 ...
#>  $ gender: Factor w/ 2 levels "m","f": 1 1 1 1 1 1 1 1 1 1 ...
#>  - attr(*, "out.attrs")=List of 2
#>   ..$ dim     : Named int [1:3] 3 5 2
#>   .. ..- attr(*, "names")= chr [1:3] "colour" "size" "gender"
#>   ..$ dimnames:List of 3
#>   .. ..$ colour: chr [1:3] "colour=g" "colour=r" "colour=b"
#>   .. ..$ size  : chr [1:5] "size=xs" "size=s" "size=m" "size=l" ...
#>   .. ..$ gender: chr [1:2] "gender=m" "gender=f"
print(jumpers[1:5,])
#>   colour size gender
#> 1      g   xs      m
#> 2      r   xs      m
#> 3      b   xs      m
#> 4      g    s      m
#> 5      r    s      m
```

The three vectors of characters (they are transformed into factors by `expand.grid`) can be easily accommodated into a list with the standard construction:

```r
# Build a list of factors
features <- list(colour=colour,size=size,gender=gender)
print(features)
#> $colour
#> [1] "g" "r" "b"
#>
#> $size
#> [1] "xs" "s"  "m"  "l"  "xl"
#>
#> $gender
#> [1] "m" "f"
```

2.4.7 Exercise 22

Consider the dataset `penguins`. A `help(penguins)` reveals the following description:

Data on adult penguins covering three species found on three islands in the Palmer Archipelago, Antarctica, including their size (flipper length, body mass, bill dimensions), and sex.

Use `aggregate` to find the average of the penguins' flipper length, body mass, and bill dimension, aggregating data by island, species, and sex. Where can you find the penguins with highest body mass index? And are they male or female?

SOLUTION

It is always a good idea to explore the content of a data frame using `str`.

```
# What;s in penguins?
str(penguins)
#> 'data.frame':    344 obs. of  8 variables:
#>  $ species    : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1 ...
#>  $ island     : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
#>  $ bill_len   : num  39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
#>  $ bill_dep   : num  18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
#>  $ flipper_len: int  181 186 195 NA 193 190 181 195 193 190 ...
#>  $ body_mass  : int  3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
#>  $ sex        : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
#>  $ year       : int  2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
```

So, we can see that `penguins` is a data frame with 344 observations of 8 variables. Three of these variables are factors, ideal for the aggregation. They are `species`, `island`, and `sex`. These are exactly the factors requested for aggregation in the question. We can then move to formulating the aggregation using `aggregate`.

```
# Aggregation by species, island, and sex
df <- aggregate(cbind(bill_len,bill_dep,flipper_len,body_mass) ~
                species+island+sex,data=penguins,FUN=mean)
print(df)
#>       species    island    sex bill_len bill_dep flipper_len body_mass
#> 1      Adelie    Biscoe female 37.35909 17.70455    187.1818  3369.318
#> 2      Gentoo    Biscoe female 45.56379 14.23793    212.7069  4679.741
#> 3      Adelie     Dream female 36.91111 17.61852    187.8519  3344.444
#> 4   Chinstrap     Dream female 46.57353 17.58824    191.7353  3527.206
#> 5      Adelie Torgersen female 37.55417 17.55000    188.2917  3395.833
#> 6      Adelie    Biscoe   male 40.59091 19.03636    190.4091  4050.000
#> 7      Gentoo    Biscoe   male 49.47377 15.71803    221.5410  5484.836
#> 8      Adelie     Dream   male 40.07143 18.83929    191.9286  4045.536
#> 9   Chinstrap     Dream   male 51.09412 19.25294    199.9118  3938.971
#> 10     Adelie Torgersen   male 40.58696 19.39130    194.9130  4034.783
```

The data frame returned contains all aggregated data (*via* mean). So, for example, the average body mass index of the male Gentoo penguins in Biscoe island, is 5484.836. In this case this is also the highest value of body mass index in our data. Therefore, the penguins with the highest body mass index are the Gentoo living in Biscoe island, and they are male.

2.4.8 Exercise 23

Carry out the same aggregation of the previous exercise, this time using function `by`.

SOLUTION

According to the `by` syntax explained in the text, the same aggregation is obtained as follows:

```
# Aggregation using by
ltmp <- by(penguins[,c("bill_len","bill_dep",
                       "flipper_len","body_mass")],
           list(penguins$species,penguins$island,penguins$sex),
           colMeans)
```

We know that `by` does not return a data frame but a list. We could see what's in the list using `str`:

```
# What's in the list?
str(ltmp)
#> List of 18
#>  $ : Named num [1:4] 37.4 17.7 187.2 3369.3
```

```
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : Named num [1:4] 45.6 14.2 212.7 4679.7
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : Named num [1:4] 36.9 17.6 187.9 3344.4
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : Named num [1:4] 46.6 17.6 191.7 3527.2
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : Named num [1:4] 37.6 17.6 188.3 3395.8
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : NULL
#>  $ : Named num [1:4] 40.6 19 190.4 4050
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : Named num [1:4] 49.5 15.7 221.5 5484.8
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : Named num [1:4] 40.1 18.8 191.9 4045.5
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : Named num [1:4] 51.1 19.3 199.9 3939
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : Named num [1:4] 40.6 19.4 194.9 4034.8
#>   ..- attr(*, "names")= chr [1:4] "bill_len" "bill_dep" "flipper_len" "body_mass"
#>  $ : NULL
#>  $ : NULL
#>  - attr(*, "dim")= int [1:3] 3 3 2
#>  - attr(*, "dimnames")=List of 3
#>   ..$ : chr [1:3] "Adelie" "Chinstrap" "Gentoo"
#>   ..$ : chr [1:3] "Biscoe" "Dream" "Torgersen"
#>   ..$ : chr [1:2] "female" "male"
#>  - attr(*, "call")= language by.data.frame(data = penguins[, c("bill_len", "bill_dep", "flipper_len"
#>  - attr(*, "class")= chr "by"
```

So, the output is more complicted than the one from `aggregate`. We can output specific elements of the list, but we need a mapping between the specific element and the combination of `species`, `island` and `sex`. This could be studied considering that the ordering of the list follows the factors grid created as

$$\text{species} \times \text{island} \times \text{sex}.$$

Thus, as the first element of `species` is `"Adelie"`, the first of `"island"` is `"Biscoe"` and the first of `sex` is `"female"` (you can read that at the bottom of the previous output), the first element of the list `ltmp` will be the one corresponding to this combination. This should coincide with the same combination as derived with aggregate. Indeed:

```
# Element 9
print(ltmp[[1]])
#>    bill_len    bill_dep flipper_len   body_mass
#>    37.35909    17.70455   187.18182  3369.31818

# Corresponding combination with aggregate
print(df[df$species == "Adelie" & df$island == "Biscoe" &
           df$sex == "female",])
#>   species island    sex bill_len bill_dep flipper_len body_mass
```

```
#> 1  Adelie Biscoe female 37.35909 17.70455    187.1818  3369.318
```

The aggregate means are the same. The second element corresponds to the combination `"Chinstrap"`, `"Biscoe"`, and `"female"` (that is, the first factor is the fastest changing, the second is intermediate, and the third is the slowest changing). There exist no data for this combination. This is also evident by the `aggregate` data frame not having this combination. So the list `ltmp` returns `NULL` as second element:

```
# Data frame
print(df)
#>      species    island    sex bill_len bill_dep flipper_len body_mass
#> 1     Adelie    Biscoe female 37.35909 17.70455    187.1818  3369.318
#> 2     Gentoo    Biscoe female 45.56379 14.23793    212.7069  4679.741
#> 3     Adelie     Dream female 36.91111 17.61852    187.8519  3344.444
#> 4  Chinstrap     Dream female 46.57353 17.58824    191.7353  3527.206
#> 5     Adelie Torgersen female 37.55417 17.55000    188.2917  3395.833
#> 6     Adelie    Biscoe   male 40.59091 19.03636    190.4091  4050.000
#> 7     Gentoo    Biscoe   male 49.47377 15.71803    221.5410  5484.836
#> 8     Adelie     Dream   male 40.07143 18.83929    191.9286  4045.536
#> 9  Chinstrap     Dream   male 51.09412 19.25294    199.9118  3938.971
#> 10    Adelie Torgersen   male 40.58696 19.39130    194.9130  4034.783

# Element 2
print(ltmp[[2]])
#> NULL
```

A quick mapping can be done using some expressions in a single line.

```
# dimnames is an attribute of ltmp
print(dimnames)
#> function (x)  .Primitive("dimnames")

# A grid created systematicaly will follow the (natural)
# order of the list. This is the mapping needed.
print(expand.grid(dimnames(ltmp)))
#>          Var1      Var2   Var3
#> 1     Adelie    Biscoe female
#> 2  Chinstrap    Biscoe female
#> 3     Gentoo    Biscoe female
#> 4     Adelie     Dream female
#> 5  Chinstrap     Dream female
#> 6     Gentoo     Dream female
#> 7     Adelie Torgersen female
#> 8  Chinstrap Torgersen female
#> 9     Gentoo Torgersen female
#> 10    Adelie    Biscoe   male
#> 11 Chinstrap    Biscoe   male
#> 12    Gentoo    Biscoe   male
#> 13    Adelie     Dream   male
#> 14 Chinstrap     Dream   male
#> 15    Gentoo     Dream   male
#> 16    Adelie Torgersen   male
#> 17 Chinstrap Torgersen   male
#> 18    Gentoo Torgersen   male

# Mapping
```

```
ltmp_map <- expand.grid(dimnames(ltmp))
print(ltmp_map)
#>          Var1      Var2   Var3
#> 1     Adelie    Biscoe female
#> 2  Chinstrap    Biscoe female
#> 3     Gentoo    Biscoe female
#> 4     Adelie     Dream female
#> 5  Chinstrap     Dream female
#> 6     Gentoo     Dream female
#> 7     Adelie Torgersen female
#> 8  Chinstrap Torgersen female
#> 9     Gentoo Torgersen female
#> 10    Adelie    Biscoe   male
#> 11 Chinstrap    Biscoe   male
#> 12    Gentoo    Biscoe   male
#> 13    Adelie     Dream   male
#> 14 Chinstrap     Dream   male
#> 15    Gentoo     Dream   male
#> 16    Adelie Torgersen   male
#> 17 Chinstrap Torgersen   male
#> 18    Gentoo Torgersen   male
```

It is clear that `by` creates a structure which is different from a data frame, differently from `aggregate`. Therefore, `by` can be used for different tasks.

3 Chapter 03

3.1 Exercises on Linear interpolation

The `comphy` package is loaded in order to make all its functions available to this exercises session.

```
library(comphy)
```

3.1.1 Exercise 01

A function $f(x)$ is known only at the values here tabulated:

x	$f(x)$
-2	3
-1	0
0	-1
1	0
2	3

Use the `linpol` function to calculate the linear interpolation corresponding to the grid $\{x_0 = -2 + 0.1i,\ i = 0, 1, \ldots, 40\}$. Plot all values and highlight the known values in red.

SOLUTION

The known values of the function are first saved as vectors `x` and `f`. Then we need to generate the grid, which is a vector, `x0`.

```
# Known values
x <- c(-2,-1,0,1,2)
```

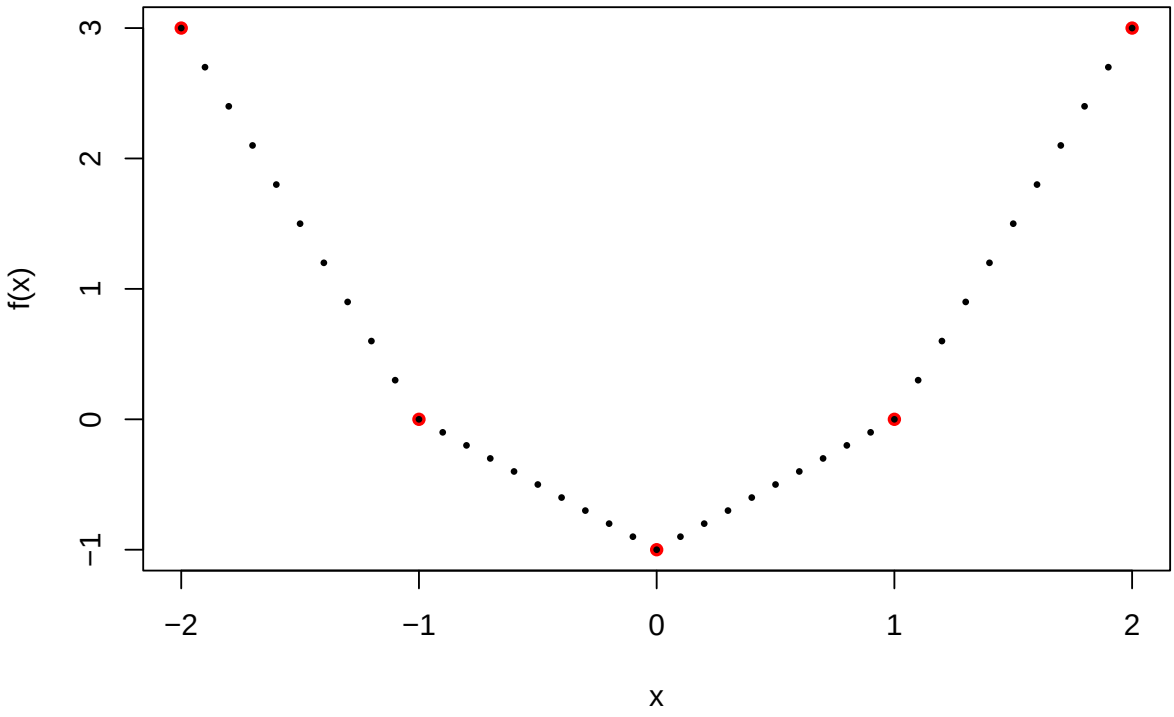
```
f <- c(3,0,-1,0,3)

# Interpolation grid
x0 <- seq(-2,2,by=0.1)
print(x0)
#>  [1] -2.0 -1.9 -1.8 -1.7 -1.6 -1.5 -1.4 -1.3 -1.2 -1.1 -1.0 -0.9 -0.8 -0.7 -0.6
#> [16] -0.5 -0.4 -0.3 -0.2 -0.1  0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9
#> [31]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0
```

Next, the interpolated values are generated using `linpol`. The output is a vector with length equal to the length of `x0` and containing the corresponding interpolated values; this vector can be called `f0`.

```
# Interpolated values
f0 <- linpol(x,f,x0)
print(f0)
#>  [1]  3.0  2.7  2.4  2.1  1.8  1.5  1.2  0.9  0.6  0.3  0.0 -0.1 -0.2 -0.3 -0.4
#> [16] -0.5 -0.6 -0.7 -0.8 -0.9 -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1
#> [31]  0.0  0.3  0.6  0.9  1.2  1.5  1.8  2.1  2.4  2.7  3.0
```

Finally, the five known values are plotted in red and the interpolated values are subsequently added to the plot, with a smaller symbol size.

```
# First plot the known points
plot(x,f,pch=16,col="red",xlab="x",ylab="f(x)")

# Then the interpolated values
points(x0,f0,pch=16,cex=0.5)
```



3.1.2 Exercise 02

Assume that the function used in Exercise 01 is $f(x) = x^2 - 1$. Plot in $x \in [-2, 2]$ the error,

$$\Delta f(x) \equiv f(x) - f_{\text{int}}(x),$$

where f_{int} is the linear approximation to $f(x)$. Verify that $\Delta f(x)$ satisfies equation (3.4).

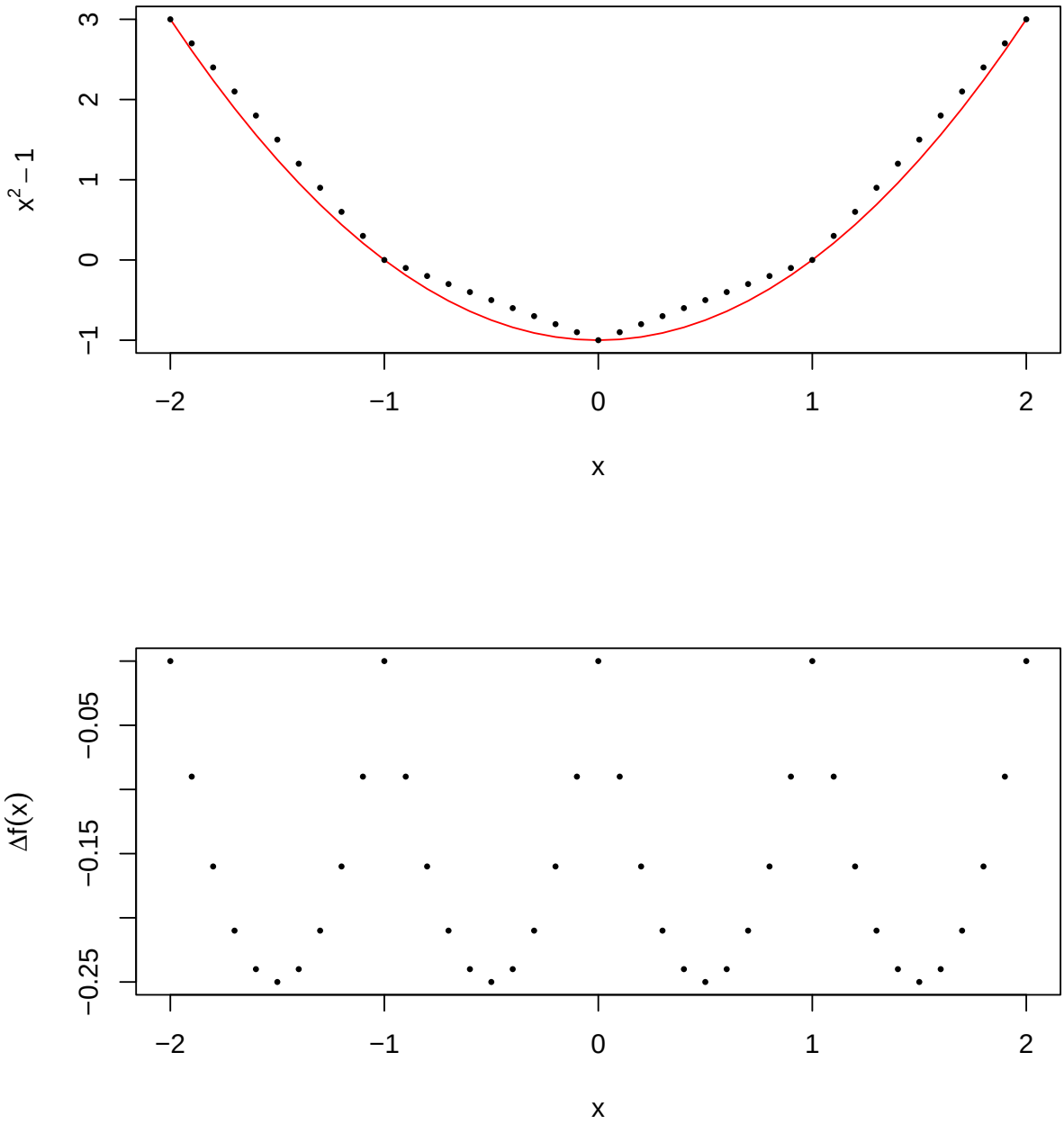
SOLUTION

The linear approximation $f_{\text{int}}(x)$ has been already computed in Exercise 01. To calculate the error $\Delta f(x) = f(x) - f_{\text{int}}(x)$ we need to sample $f(x) = x^2 - 1$ at the same points of the interpolating grid `x0`; the vector containing the correct values is called `ftrue`. The error is then easily obtained as difference between vector `ftrue` and vector `f0`.

```
# Function x^2-1 is sampled at x0
ftrue <- x0^2-1

# Error
Deltaf <- ftrue-f0
summary(Deltaf)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>  -0.250  -0.240  -0.160  -0.161  -0.090   0.000
```

As the summary statistics clearly show, the error is a non-positive function as its maximum value is zero. The reason for this can be appreciated by looking at the plot of $f(x) = x^2 - 1$ overlapped to its linear interpolation.

```r
# Two plots, one on top of the other
par(mfrow=c(2,1))

# Plot x^2-1 (in red)
plot(x0,ftrue,type="l",xlab="x",ylab=expression(x^2-1),col="red")

# Plot linearly interpolated values (in black)
points(x0,f0,pch=16,cex=0.5)


# Plot difference
plot(x0,Deltaf,pch=16,cex=0.5,xlab="x",ylab=
        expression(paste(Delta,"",f(x))))
```



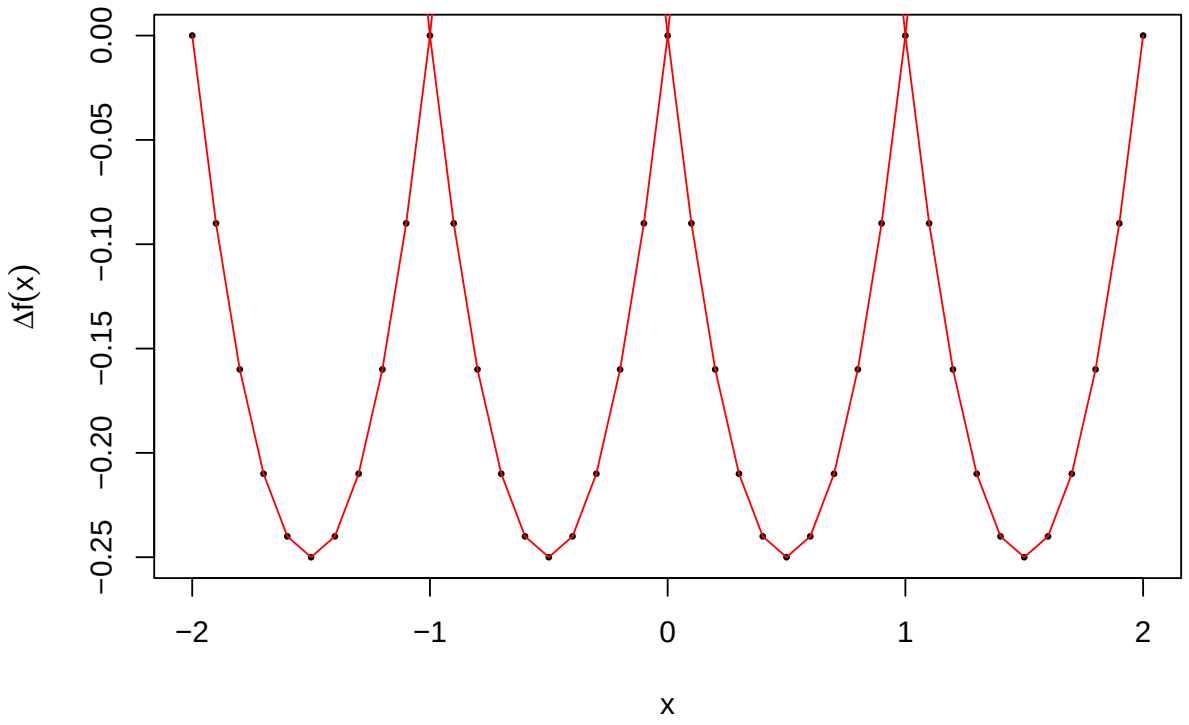
The function is always below its linear interpolation, for this specific example, so that their difference is negative or zero. It is possible, using equation (3.4) to estimate the largest value of the interpolation error. From the picture, this seems to be equal to $1/4 = 0.25$. Formula (3.4) yields

$$\Delta f(x) = \frac{f''(\xi)}{2}(x - x_1)(x - x_2),$$

where x_1 and x_2 are the interpolation interval's extremes. For the function under study, the second derivative is a constant, $f''(\xi) = 2$. Therefore the error coincides, for each interval, with $(x - x_1)(x - x_2)$. This seems

to agree with the error plot, where four parabolas are depicted, one for each interpolationn interval. We can also verify that the parabolas are of the form $(x - x_1)(x - x_2)$.

```
# First plot the error as previously sampled
plot(x0,Deltaf,pch=16,cex=0.5,xlab="x",
     ylab=expression(paste(Delta,"",f(x))))

# Next, loop over four interpolation intervals
# and draw the error curves
for (i in 1:4) {
   init <- i
   ifin <- init+1
   x1 <- x[init]
   x2 <- x[ifin]
   ferr <- (x0-x1)*(x0-x2)
   points(x0,ferr,type="l",col="red")
}
```



Perfect agreement! One has to admit a certain satisfaction in observing the agreement of theory with the numerical application. Looking at the analytic expression of the error, it is clear that the largest contribution comes when x is in the midle of each interval. So, considering for instance the first interval $-2 \leq x \leq -1$, the expression $(x+2)(x+1)$ for the error is highest when $x = -3/2$. For this value we have $|\Delta f(-3/2)| = |(-3/2+2)(-3/2+1)| = 1/4 = 0.25$. Thus, the largest error has magnitude 0.25, as previously observed empirically.

3.1.3 Exercise 03

Sample the function

$$f(x) = 2\sin(x) - \cos(2x)$$

at 20 random points in the interval $(0, 2\pi)$. Then find all linear interpolations in the 21 interpolation intervals created in $[0, 2\pi]$. Finally, plot $f(x)$ and the linear interpolation in the same plot.

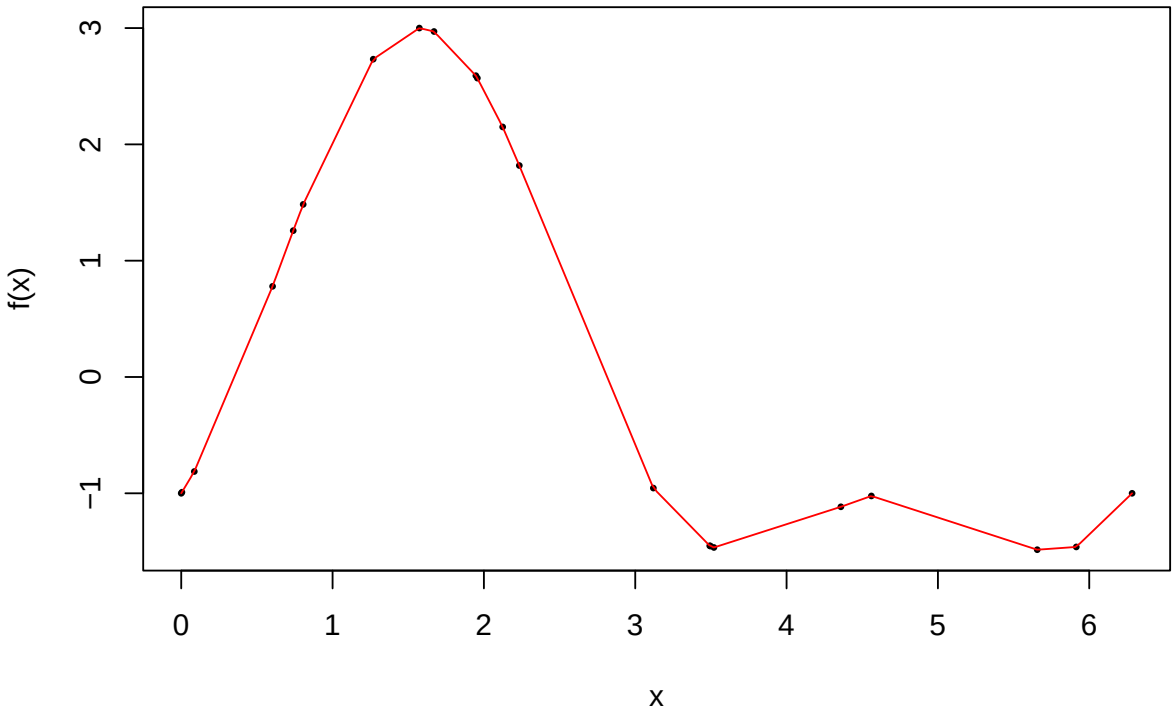
SOLUTION

The 20 random points can be found using `runif` to generate random deviated from a uniform distribution between 0 and 2π.

```
set.seed(1960)  # Fix random seed to reproduce results exactly

# Generate points at which function is known
# Note: points don't need to be sorted.
x <- runif(n=20,min=0,max=2*pi)

# Make sure 0 and 2*pi are not in the sample
print(x)
#>  [1] 3.49232114 3.49681503 2.23393832 1.26866630 0.00457501 1.94685465
#>  [7] 0.80588243 2.12394305 4.35883368 4.56042859 1.95670734 5.91431193
#> [13] 0.60356949 3.51987432 1.57347511 0.08639646 3.11960028 5.65634513
#> [19] 0.74008075 1.67035781

# x0 is augmented to include 0 and 2*pi
x <- c(x,0,2*pi)

# function values sampled at the previous points
f <- 2*sin(x)-cos(2*x)
```

Next, a uniform grid is created between 0 and 2π and linear interpolation performed. Both known and interpolated values are displayed in a plot between 0 and 2π.

```
# Uniform grid (201 points)
x0 <- seq(0,2*pi,length.out=201)

# Linear interpolation
f0 <- linpol(x,f,x0)

# Plot points and linear interpolations
plot(x,f,pch=16,cex=0.5,xlab="x",ylab="f(x)")
points(x0,f0,type="l",col="red")
```



3.1.4 Exercise 04

Consider the function and linear interpolation found in Exercise 03. Write a function called `which_max` that takes in the correct function values and the related linear interpolations, and returns the value at which the interpolation error

$$\Delta f(x) = f(x) - f_{\text{int}}(x)$$

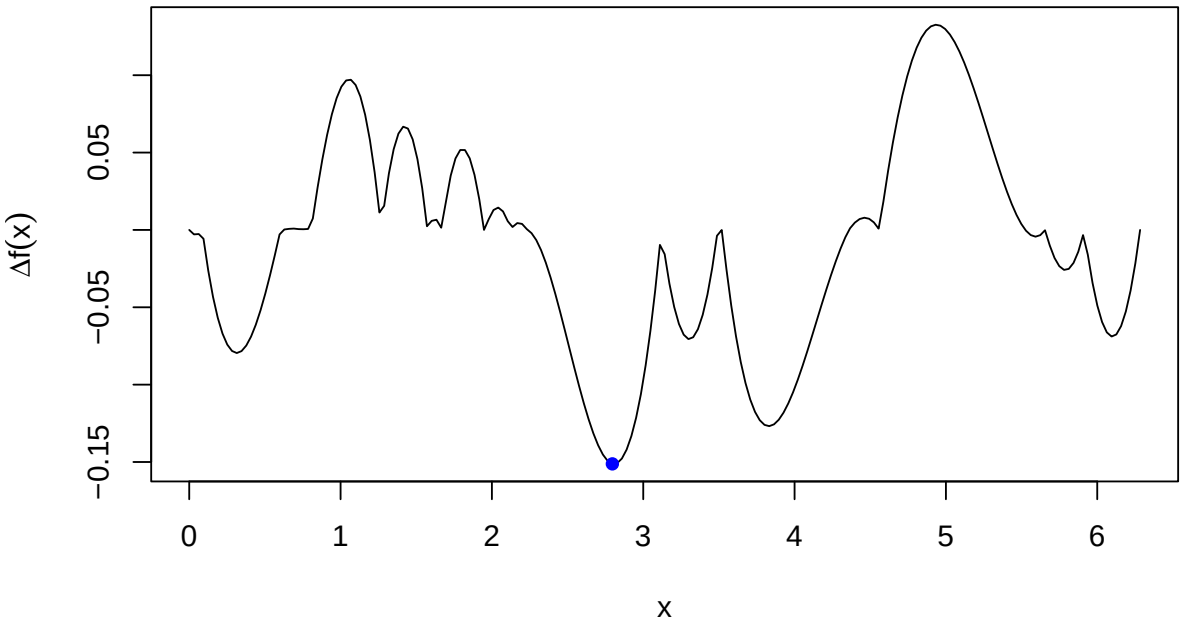
is the largest. Can you justify the value found, using formula (3.4)?

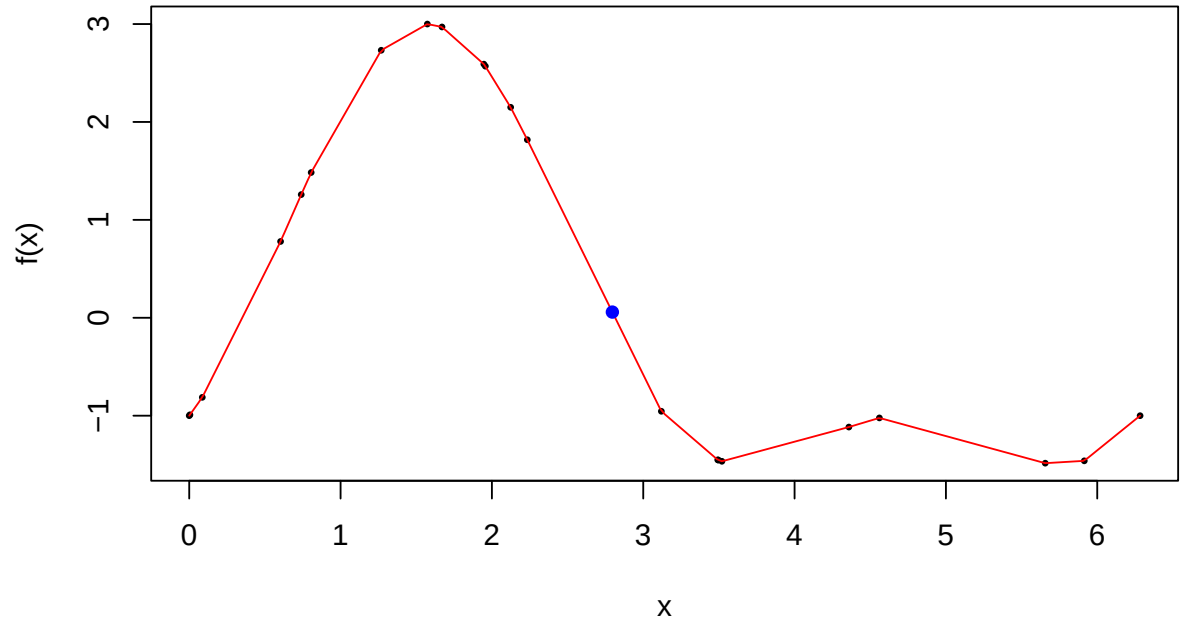
SOLUTION

The function can be design so to take in `x0` and `f0` (from Exercise 03) and return the index `idx` for which `x0[idx]` yields the largest error. The following is a possible design.

```
# Definition of the function
which_max <- function(x0,f0) {
   # Correct function sampled at x0
   ftrue <- 2*sin(x0)-cos(2*x0)

   # Error (magnitude, i.e. absolute values)
   Deltaf <- abs(ftrue-f0)

   # Index corresponding to largest error
   idx <- which(Deltaf == max(Deltaf))

   # Return index
   return(idx)
```

```
}

# Now use function
idx <- which_max(x0,f0)
print(idx)
#> [1] 90
print(c(x0[idx],f0[idx]))
#> [1] 2.79601746 0.05817791

# Let's check if the selected point makes sense visually
par(mfrow=c(2,1))
ftrue <- 2*sin(x0)-cos(2*x0)
Deltaf <- ftrue-f0
plot(x0,Deltaf,type="l",xlab="x",
     ylab=expression(paste(Delta,"",f(x))))
points(x0[idx],Deltaf[idx],pch=16,col="blue")

# Overlap function and known points for analysis
plot(x,f,pch=16,cex=0.5,xlab="x",ylab="f(x)")
points(x0,f0,type="l",col="red")
points(x0[idx],f0[idx],pch=16,col="blue")
```





The largest error is a negative error at 2.7960175 radians. This is found in correspondence to the 13th interpolation interval, which is also the widest one. The interpolation error, according to formula (3.4), is indeed proportional to the product $(x - x_{13})(x - x_{14})$, where x_{13} and x_{14} are the extremes for the 13th

interpolation interval. For a wide interval the product is more likely to return a large value. The list of 21 products corresponding to the 21 interpolation intervals for this example is worked out in the following snippet.

```
# Sort x and f to identify intervals
jdx <- order(x)
x <- x[jdx]
f <- f[jdx]

# Loop to measure product (x-x1)*(x-x2)
xw <- c()    # Intervals' width
prd <- c()   # Products
for (i in 1:(length(x)-1)) {
   # Search many products in  given interval
   # and select the larges
   xx <- seq(x[i],x[i+1],length.out=100)
   prdcts <- (xx-x[i])*(xx-x[i+1])
   xw <- c(xw,x[i+1]-x[i])
   prd <- c(prd,max(abs(prdcts)))
   msg <- sprintf("%2d  %6.4f %6.4f %6.4f %10.6f\n",
                  i,x[i],x[i+1],xw[i],prd[i])
   cat(msg)
}
#>  1  0.0000 0.0046 0.0046   0.000005
#>  2  0.0046 0.0864 0.0818   0.001674
#>  3  0.0864 0.6036 0.5172   0.066860
#>  4  0.6036 0.7401 0.1365   0.004658
#>  5  0.7401 0.8059 0.0658   0.001082
#>  6  0.8059 1.2687 0.4628   0.053537
#>  7  1.2687 1.5735 0.3048   0.023225
#>  8  1.5735 1.6704 0.0969   0.002346
#>  9  1.6704 1.9469 0.2765   0.019111
#> 10  1.9469 1.9567 0.0099   0.000024
#> 11  1.9567 2.1239 0.1672   0.006991
#> 12  2.1239 2.2339 0.1100   0.003024
#> 13  2.2339 3.1196 0.8857   0.196079
#> 14  3.1196 3.4923 0.3727   0.034727
#> 15  3.4923 3.4968 0.0045   0.000005
#> 16  3.4968 3.5199 0.0231   0.000133
#> 17  3.5199 4.3588 0.8390   0.175945
#> 18  4.3588 4.5604 0.2016   0.010159
#> 19  4.5604 5.6563 1.0959   0.300228
#> 20  5.6563 5.9143 0.2580   0.016635
#> 21  5.9143 6.2832 0.3689   0.034013
```

The width of the 13th interpolation interval is 0.885662 and it gives a product $(x - x_{13})(x - x_{14})$ which has the largest magnitude, 0.1960793. This is, in fact, the second largest product as the first relates to the 19th interval. In fact, the second largest error, and not the first, falls in the 19th interval. The reason for this is that even if the product $(x - x_{19})(x - x_{20})$ has largest magnitude (0.3002276 is larger than 0.1960793), the actual interpolation error depends also on the second derivative term, $f''(\xi)/2$ in expression (3.4). The second derivative for the 13th interval is larger than the one for the 19th interval, as one can verify.

3.1.5 Exercise 05

A linear interpolation of $f(x) = x^2$ is carried out between $x = 0$ and $x = 1$ using a grid of equally-spaced values, $x_i = (i-1)d\ ,\ i = 1, \ldots, n+1,\ d > 0$. What value should be assigned to d in order to keep the interpolation error $|\Delta f(x)| \equiv |f(x) - f_{\rm int}(x)|$ smaller than an assigned positive number ϵ?

**SOLUTION*

The formula for the interpolation error is,

$$\Delta f(x) = \frac{f''(\xi)}{2}(x - x_1)(x - x_2),$$

where x_1 and x_2 are respectively the left and right extremes of each interpolation interval. A generic interpolation interval has extremes,

$$x_{i-1} = (i-1)d \qquad \text{and} \qquad x_i = id$$

The second derivative of $f(x) = x^2$ is 2. Therefore the analytic expression for the interpolation error, for the generic interpolation interval, is,

$$\Delta f(x) = [x - (i-1)d][x - id]$$

The largest value (positive or negative) of this expression can be found setting its first derivative equal to zero. The result is,

$$x - id + x - (i-1)d = 2x - (2i-1)d = 0 \ \Rightarrow\ x = \left(i - \frac{1}{2}\right) d$$

Thus, the largest error happens in the middle of each interpolation interval. The extent of the error can be found replacing $(i - 1/2)d$ in the expression for $\Delta f(x)$ and obtaining,

$$\Delta f(x) = \left[\left(i - \frac{1}{2}\right) d - (i-1)d\right] \left[\left(i - \frac{1}{2}\right) d - id\right] = -\frac{1}{4}d^2$$

If the absolute value of the error, $|\Delta f(x)|$, has to be kept smaller than ϵ:

$$|\Delta f(x)| < \epsilon \ \Rightarrow\ \frac{1}{4}d^2 < \epsilon \ \Rightarrow\ d < 2\sqrt{\epsilon}$$

So once ϵ has been fixed, the interpolation interval's width, d, will have to be smaller than $2\sqrt{\epsilon}$.

3.2 Exercises on Lagrangian interpolation and the Neville-Aitken algorithm

3.2.1 Exercise 06

Find the four basic Lagrangian polynomials $L_{3,i}\ ,\ i = 1, 2, 3, 4$ for the function $f(x) = x^3 - 5x^2 + 3x + 2$ where the four interpolating points are,

$$x_1 = 1\ ,\ x_2 = -1\ ,\ x_3 = 0\ ,\ x_4 = 2$$

Then write the expression for $f(x)$ as a linear expansion in term of the basic Lagrange polynomials. Finally, plot $f(x)$ between $x = -2$ and $x = 3$ and plot the interpolating points in red.

SOLUTION

Each basic Lagrange polynomial is a third-degree polynomial expressed as product of three factors of the

form $x - x_i$. The four basic Lagrange polynomials for the interpolating points given are:

$$\begin{aligned} L_{3,1}(x) &= \frac{(x+1)x(x-2)}{(2)(1)(-1)} = -\frac{x(x+1)(x-2)}{2} \\ L_{3,2}(x) &= \frac{(x-1)x(x-2)}{(-2)(-1)(-3)} = -\frac{x(x-1)(x-2)}{6} \\ L_{3,3}(x) &= \frac{(x-1)(x+1)(x-2)}{(-1)(1)(-2)} = \frac{(x-1)(x+1)(x-2)}{2} \\ L_{3,4}(x) &= \frac{(x-1)(x+1)x}{(1)(3)(2)} = \frac{x(x-1)(x+1)}{6} \end{aligned}$$

These polynomials satisfy the property:

$$L_{n,i}(x_j) = \begin{cases} 0 & \text{for} \quad i \neq j \\ 1 & \text{for} \quad i = j \end{cases},$$

as it is easy to verify. The function can be written as linear combination of the four polynomials just introduced, where the coefficients are the values of the function itself at the interpolation points:

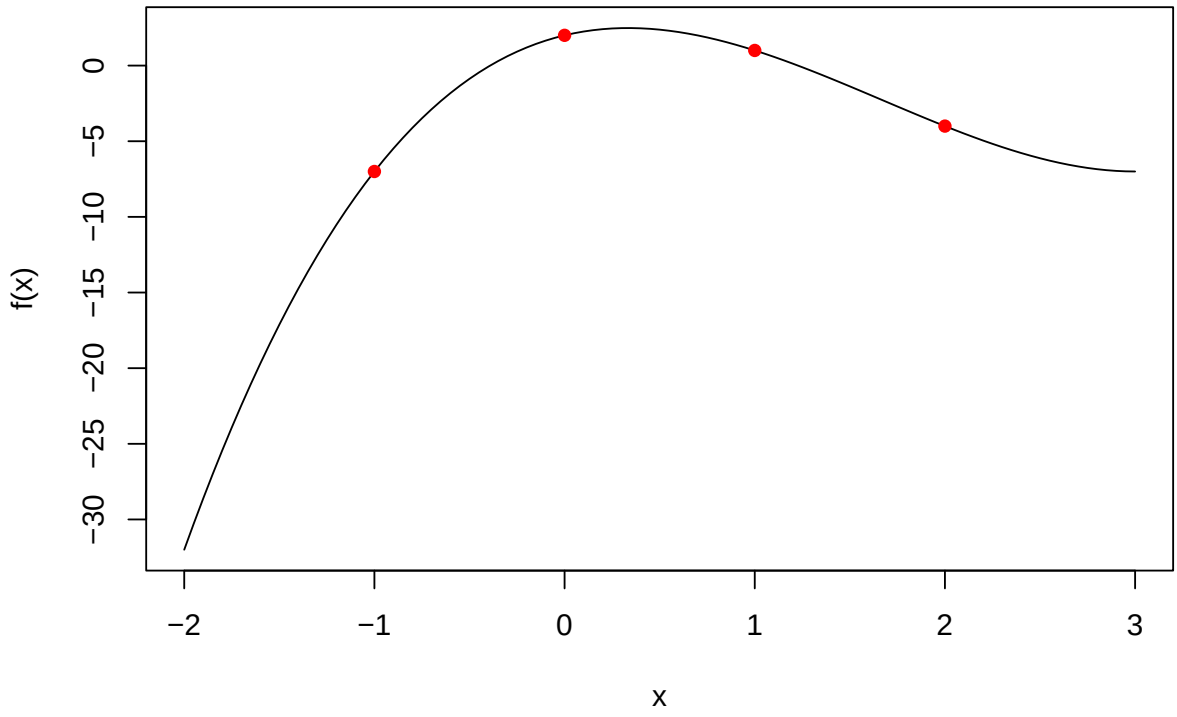
$$f(x) = \sum_{i=1}^{4} f_i L_{3,i}(x)$$

The coefficients are listed in the following table, obtained by replacing the four x_i in the expression $f(x) = x^3 - 5x^2 + 3x + 2$:

x	f_i
1	1
-1	-7
0	2
2	-4

Finally, the required plot can be readily created using the following R code snippet.

```
# Four interpolating points
x <- c(1,-1,0,2)
f <- c(1,-7,2,-4)

# Function curve
curve(x^3-5*x^2+3*x+2,from=-2,to=3,
      xlab="x",ylab="f(x)")

#Known points
points(x,f,pch=16,col="red")
```



3.2.2 Exercise 07

Write the analytic form of the Lagrange polynomial of degree 4, $P_4(x)$, that interpolates the five points

$$x_1 = 0 \ , \ x_2 = \pi/6 \ , \ x_3 = \pi/4 \ , \ x_4 = \pi/3 \ , \ x_5 = \pi/2$$

of the function $f(x) = \sin(x)$ in the interval $x \in [0, \pi/2]$. Plot the curves corresponding to both $f(x)$ and $P_4(x)$, and highlight in red the five points of the interpolation.

SOLUTION

The values of $f(x)$ at the five interpolating points are $f_1 = 0$, $f_2 = 1/2$, $f_3 = \sqrt{2}/2$, $f_4 = \sqrt{3}/2$, $f_5 = 1$. The analytic form of the polynomial is, therefore:

$$\begin{aligned} P_4(x) \quad = \quad & \frac{(x-\pi/6)(x-\pi/4)(x-\pi/3)(x-\pi/2)}{(-\pi/6)(-\pi/4)(-\pi/3)(-\pi/2)} 0 + \\ & \frac{x(x-\pi/4)(x-\pi/3)(x-\pi/2)}{(\pi/6)(\pi/6-\pi/4)(\pi/6-\pi/3)(\pi/6-\pi/2)} (1/2) + \\ & \frac{x(x-\pi/6)(x-\pi/3)(x-\pi/2)}{(\pi/4)(\pi/4-\pi/6)(\pi/4-\pi/3)(\pi/4-\pi/2)} (\sqrt{2}/2) + \\ & \frac{x(x-\pi/6)(x-\pi/4)(x-\pi/2)}{(\pi/3)(\pi/3-\pi/6)(\pi/3-\pi/4)(\pi/3-\pi/2)} (\sqrt{3}/2) + \\ & \frac{x(x-\pi/6)(x-\pi/4)(x-\pi/3)}{(\pi/2)(\pi/2-\pi/6)(\pi/2-\pi/4)(\pi/2-\pi/3)} (1) \end{aligned}$$

This is quite a lengthy expression, but the advantage of it is that it can be written down quite straightforwardly.

A shorter analytic form can only be achieved after having solved a system of 5 equations with five unknowns. That is why Lagrangian interpolation is convenient.

A program to exploit the above formula needs to consider a function to return the expression of $P_4(x)$ for each value of x in the given interval. A possible solution is the following in which the long products at the top and bottom of the Lagrange polynomial are coded in a very synthetic fashion using R's parallelisation features.

```
P_4 <- function(x) {
   xp <- c(0,pi/6,pi/4,pi/3,pi/2)
   fp <- c(0,0.5,sqrt(2)/2,sqrt(3)/2,1)

   # Create a vector and then sum its elements
   f <- c()
   for (i in 1:5) {
    # Top
    tt <- prod(x-xp[-i])  # xp without i-th element

    # Bottom
    bb <- prod(xp[i]-xp[-i])

    # Partial sum
    f <- c(f,tt*fp[i]/bb)
   }

   return(sum(f))
}
```

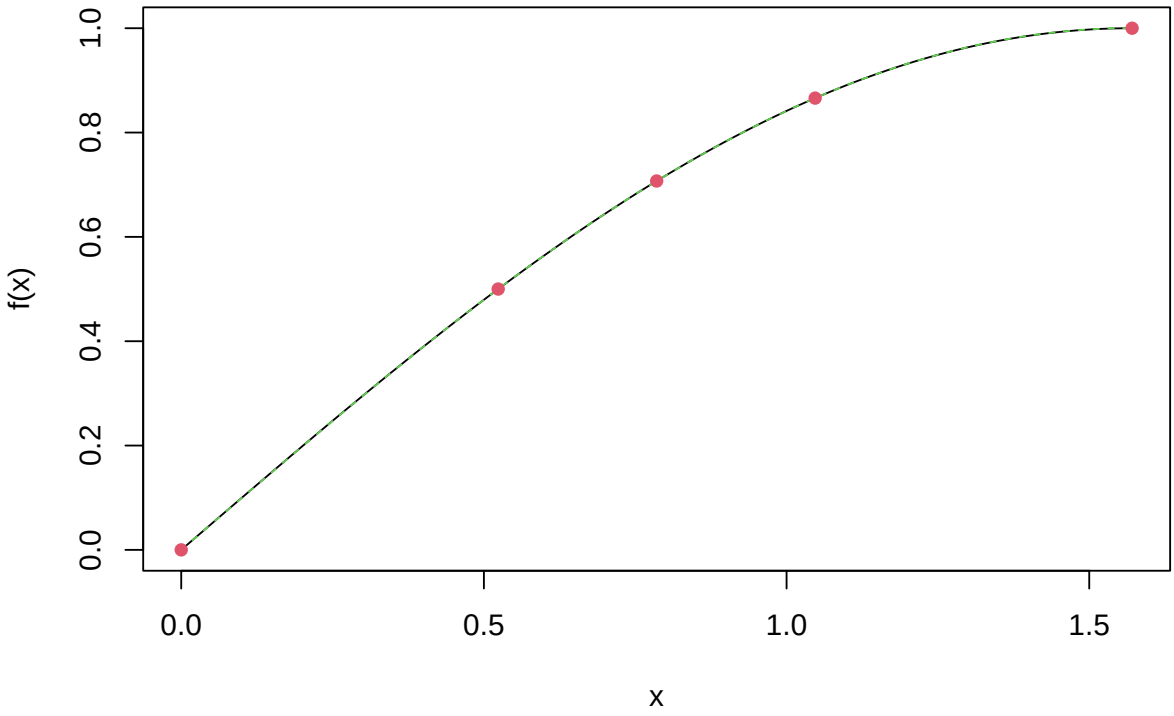
Once created, a function should always be tested to check that it returns the expeccted numerical values. In our case it is worth trying with the 5 interpolation points.

```
# The value of P_4(x) at xp should return fp

# These have to be re-defined because in the previous
# code section they were hidden inside a function
xp <- c(0,pi/6,pi/4,pi/3,pi/2)
fp <- c(0,0.5,sqrt(2)/2,sqrt(3)/2,1)

# Check
for (i in 1:5) {
   print(c(P_4(xp[i]),fp[i]))
}
#> [1] 0 0
#> [1] 0.5 0.5
#> [1] 0.7071068 0.7071068
#> [1] 0.8660254 0.8660254
#> [1] 1 1
```

The exercise can be then completed by plotting $f(x)$, $P_4(x)$ and the five interpolation points. For this task is important to use the R function `sapply` as the function `P_4` created takes only scalar values as input. A vector input like the `x` of the following code chunk would give incorrect results.

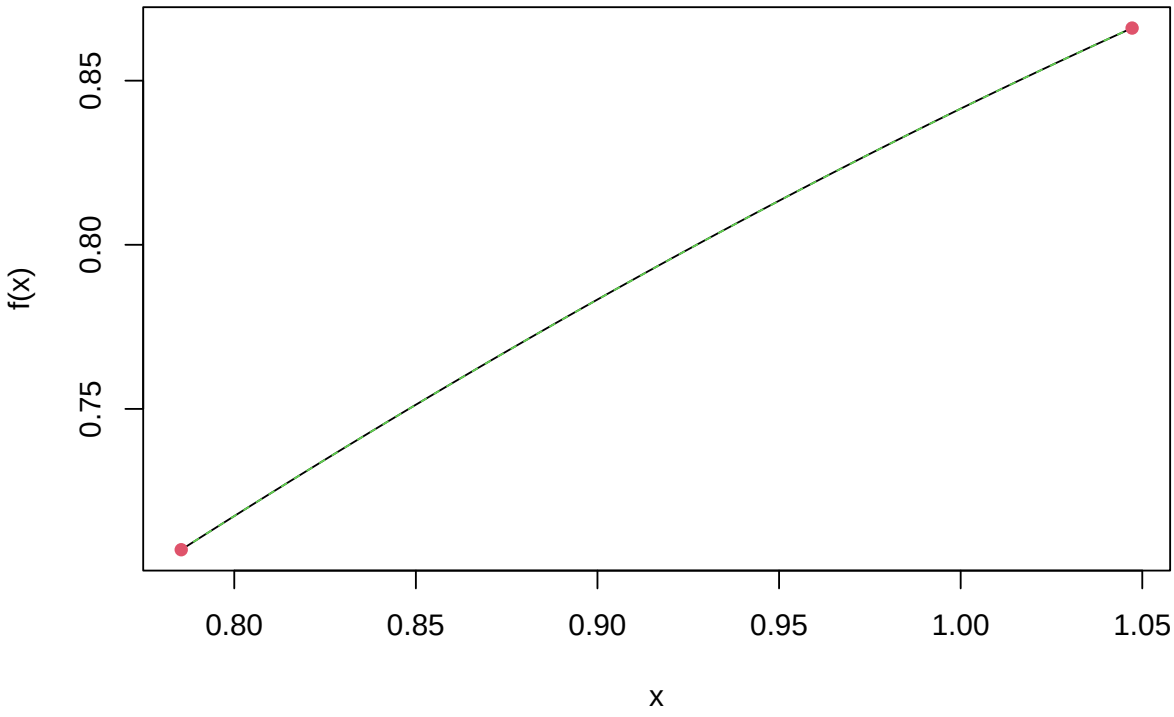
```
x <- seq(0,pi/2,length=1000)
f <- sin(x)

# P_4's argumentis a scalar. Use sapply for more values
```

```
P4 <- sapply(x,P_4)
plot(x,f,type="l",col=1,xlab="x",ylab="f(x)")
points(x,P4,type="l",lty=2,col=3)
points(xp,fp,pch=16,col=2)
```



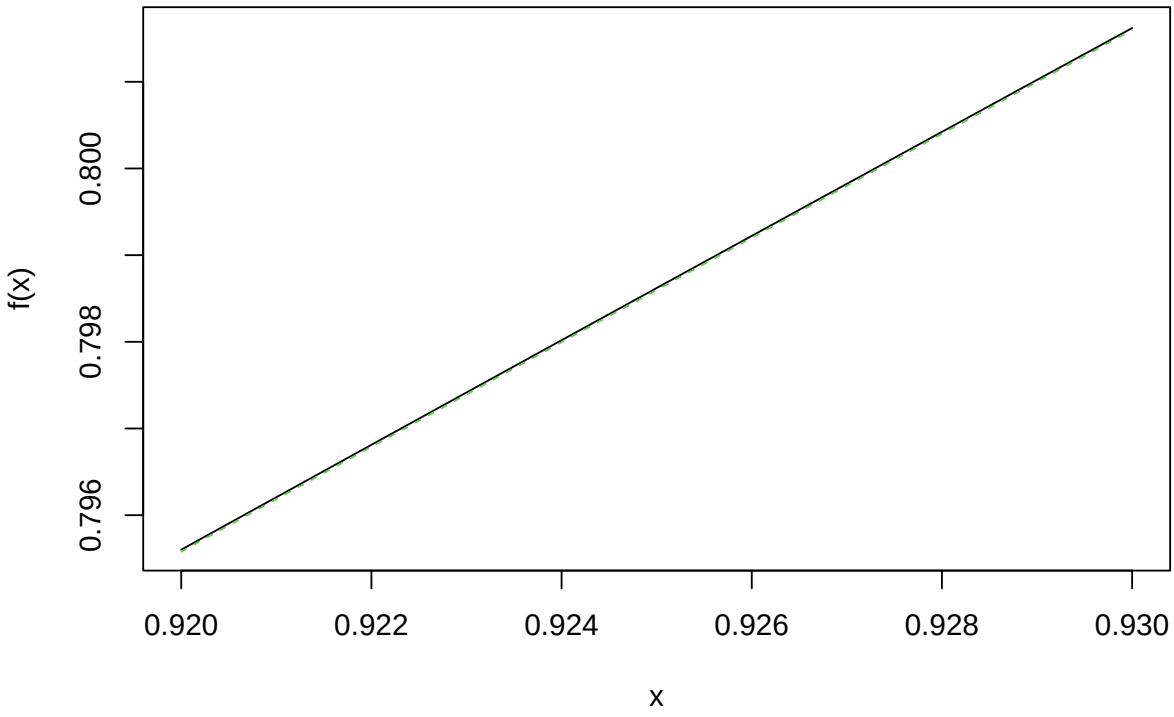
The overlap is a good one, which means the approximation of $f(x)$ with $P_4(x)$ is quite accurate. To appreciate how close the interpolating curve is to $\sin(x)$ we could zoom around a couple of interpolation points, say x_3 and x_4; the visual result is produced using the following code.

```
x <- seq(xp[3],xp[4],length=1000)
f <- sin(x)

# P_4's argumentis a scalar. Use sapply for more values
P4 <- sapply(x,P_4)
plot(x,f,type="l",col=1,xlab="x",ylab="f(x)")
points(x,P4,type="l",lty=2,col=3)
points(xp[3:4],fp[3:4],pch=16,col=2)
```



The approximation cannot be appreciated visually even in this smaller interval. We could zoom in the plot even closer, this time plotting the region between, say, 0.92 and 0.93.

```
x <- seq(0.92,0.93,length=1000)
f <- sin(x)
P4 <- sapply(x,P_4)
plot(x,f,type="l",col=1,xlab="x",ylab="f(x)")
points(x,P4,type="l",lty=2,col=3)
```



The two curves seem to be nearly coinciding. The approximation of that stretch of curve with 5 interpolation points is, therefore, very good. An estimate of the approximation error, which turns out to be around 4×10^{-4}, is given in the next exercise.

3.2.3 Exercise 08

Consider the difference between the function of Exercise 07, $f(x)$, and its approximation using Lagrange polynomials, $P_4(x)$,

$$\Delta P_4(x) = f(x) - P_4(x)$$

Estimate the largest value of $|\Delta P_4(x)|$, ΔP, in the interval $x \in [0, \pi/2]$.

SOLUTION

The error, $\Delta P_n(x)$ of a Lagrangian interpolation is given by the formula,

$$\Delta P_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_1)(x - x_2)\cdots(x - x_{n+1}),$$

where $x_1, x_2, \ldots, x_{n+1}$ are the $n+1$ interpolation points, and ξ is a point in the interpolating interval, with the exclusion of the interpolation points. For $P_4(x)$ we have,

$$\Delta P_4(x) = \frac{f^{(5)}(\xi)}{5!}(x - x_1)(x - x_2)(x - x_3)(x - x_4)(x - x_5)$$

As it is known that $f(x) = \sin(x)$, it follows that $f^{(5)}(x) = \cos(x)$. The point ξ is not determined, but it is in the interval $[0, \pi/2]$. The largest value $\cos(x)$ can have in that interval is 1. Using $\cos(x) = 1$, we can

indicate with $\Delta P(x)$ the following quantity:

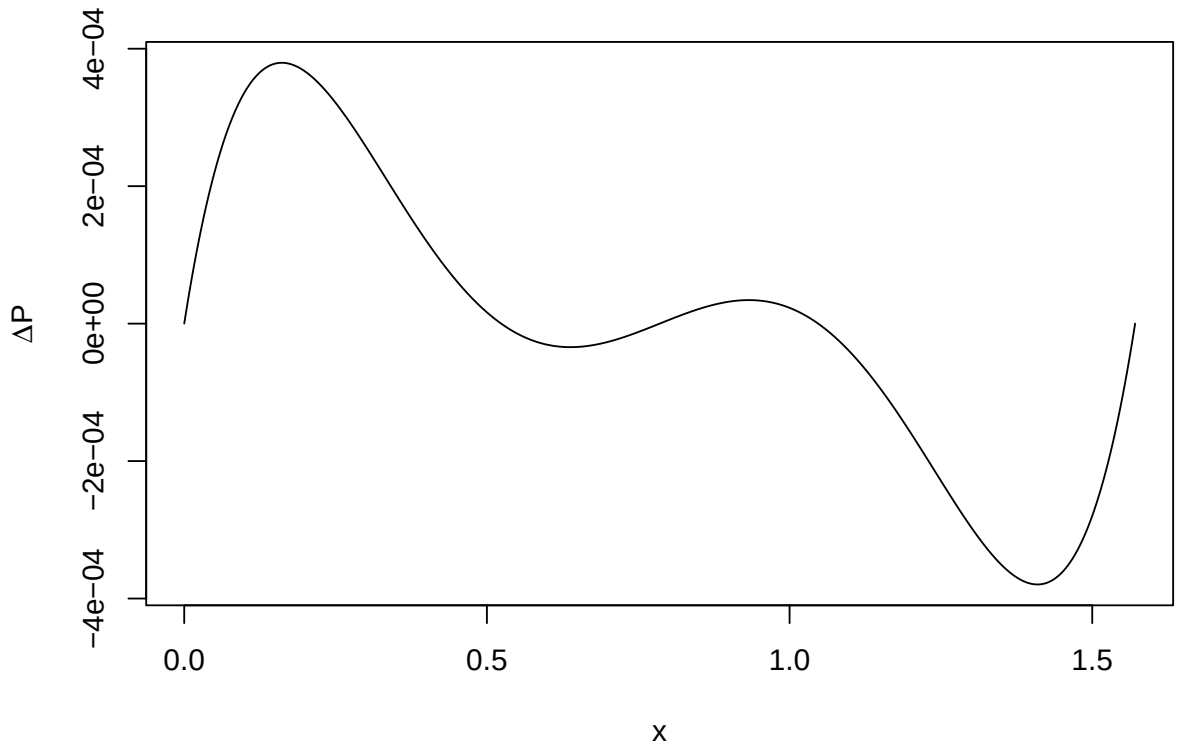
$$\Delta P(x) \equiv \frac{1}{120} x \left(x - \frac{\pi}{6}\right) (x - \pi/4) (x - \pi/3) (x - \pi/2)$$

The value ΔP we are looking for is therefore:

$$\Delta P = max_{x \in [0,\pi/2]} |\{\Delta P(x)\}|$$

The analytic method to find the maximum value of $\Delta P(x)$ consists in taking its derivative and set it to zero. One or more of the solutions of the algebraic equation obtained correspond to the maximum. The solution or solutions might, in fact, also correspond to a minimum, which is equally acceptable. The equation whose roots are wanted is a fourth degree algebraic equation. Such roots could be found with a formula, but we will use here an empirical method that takes advantage of R's quick generation of grids. Essentially, a grid `x` of values between 0 and $\pi/2$ is created and the expression for $\Delta P(x)$ is calculated for all the points in the grid. Maxima and minima are then easily found either using a plot or with the `max` or `min` functions.

```
# Grid of 10000 points (for accurate results)
x <- seq(0,pi/2,length.out=10000)

# Delta P(x)
DeltaP <- (1/120)*x*(x-pi/6)*(x-pi/4)*(x-pi/3)*(x-pi/2)

# Plot to see maxima and minima
plot(x,DeltaP,type="l",xlab="x",
     ylab=expression(paste(Delta,"",P)))
```

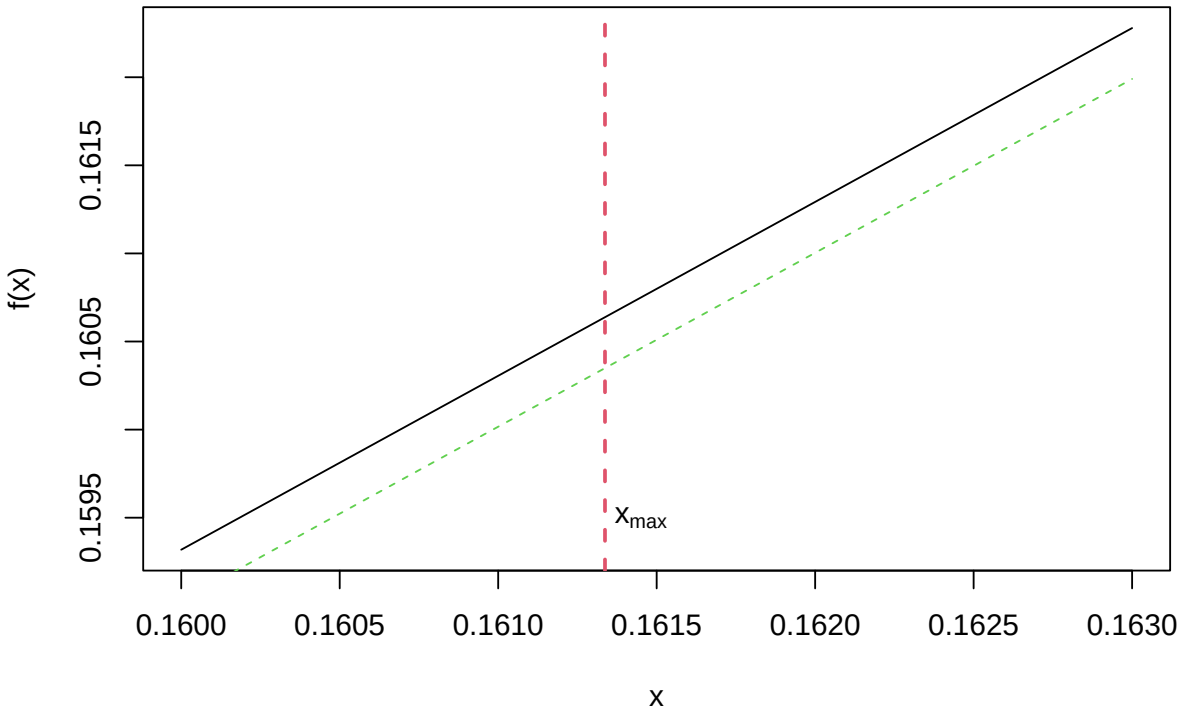


From the plot there appear to be a highest and lowest point, close to 0 and $\pi/2$ respectively. We have indeed,

```r
# Max
idx <- which(DeltaP == max(DeltaP))
xmax <- x[idx]
print(DeltaP[idx])
#> [1] 0.0003795075

# Min
idx <- which(DeltaP == min(DeltaP))
xmin <- x[idx]
print(DeltaP[idx])
#> [1] -0.0003795075

# Maximum (or minimum) approximation error
dp <- abs(DeltaP[idx])
```

The interpolation error, given what has been said, cannot be larger than 0.000380. The segments between two interpolation points, in which the largest discrepancy between $f(x)$ and $P_4(x)$ is observed are the one between 0 and $\pi/6$ and the one between $\pi/3$ and $\pi/2$. More specifically, any small intervall around either of the points 0.1613369 or 1.4094594, would show the discrepancy better. This is what is done in the following code, where such a small interval is with $x \in [0.160, 0.163]$.

```r
x <- seq(0.160,0.163,length=1000)
f <- sin(x)
P4 <- sapply(x,P_4)
plot(x,f,type="l",col=1,xlab="x",ylab="f(x)")
points(x,P4,type="l",lty=2,col=3)
abline(v=xmax,lty=2,lwd=2,col=2)
text(0.16145,0.1595,label=expression(x[max]))
```



In the plot, the vertical red, dashed line indicates the point where the discrepancy is highest.

3.2.4 Exercise 09

Consider the following three known points for a given function $f(x)$:

x	$f(x)$
0	0
0.5	0.125
1	1

Find all interpolated values between 0 and 1 using Lagrangian interpolation ($P_2(x)$) and interpolation with the Neville-Aitken algorithm, on a grid `x0` of 20 points. Plot both interpolations and the known points to show that they coincide.

SOLUTION

A judicious use of the R function `sapply` can help using the `cry` function `nevaitpol` on the many values of the grid vector `x0`. First, let us define the function $P_2(x)$ representing Lagrangian interpolation.

```
P_2 <- function(x) {
   xp <- c(0,0.5,1)
   fp <- c(0,0.125,1)

   # Create a vector and then sum its elements
   f <- c()
```

```
  for (i in 1:3) {
   # Top
   tt <- prod(x-xp[-i])  # xp without i-th element

   # Bottom
   bb <- prod(xp[i]-xp[-i])

   # Partial sum
   f <- c(f,tt*fp[i]/bb)
  }

  return(sum(f))
}
```

Then the grid `x0` is created.

```
x0 <- seq(0,1,length.out=20)
```

The first set of interpolated points, `int01`, is created applying the function `P_2` over `x0` using `sapply`.

```
int01 <- sapply(x0,P_2)
print(int01)
#>  [1]  0.000000000 -0.022160665 -0.036011080 -0.041551247 -0.038781163
#>  [6] -0.027700831 -0.008310249  0.019390582  0.055401662  0.099722992
#> [11]  0.152354571  0.213296399  0.282548476  0.360110803  0.445983380
#> [16]  0.540166205  0.642659280  0.753462604  0.872576177  1.000000000
```

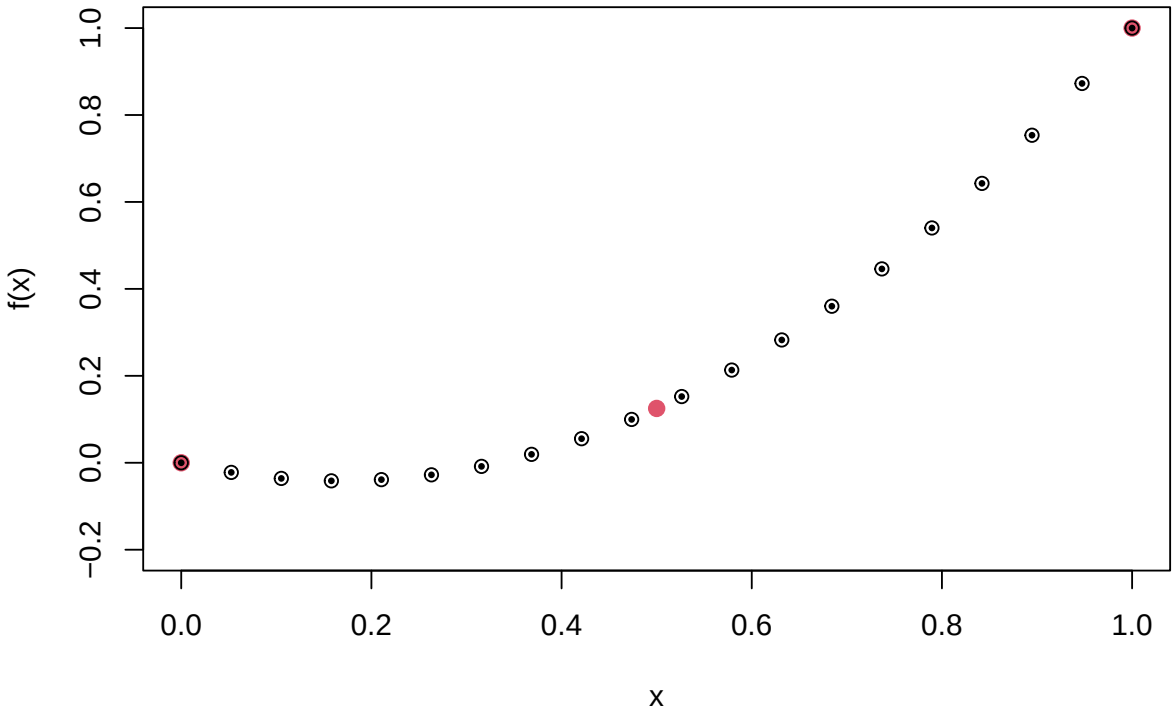
The application of `nevaitpol`, needed to obtain the interpolating point with Neville-Aitken, is more complicated. What is complicated is not so much the use of `sapply`, but how to access the interpolating point afterwards. Let's start with `sapply`. As `x0` has 20 elements and as `nevaitpol` creates a matrix, the result of `sapply` is a list of length 20 and in which each element is the matrix returned by `nevaitpol`.

```
# Known points
xp <- c(0,0.5,1)
fp <- c(0, 0.125,1)

# sapply for nevaitpol over x0
# Additional arguments are those needed by
# nevaitpol, x and f.
# simplify=FALSE helps keeping the matrix
# structure returned by nevaitpol
l20 <- sapply(x0,nevaitpol,x=xp,f=fp,
              simplify=FALSE)

print(length(l20))
#> [1] 20
print(class(l20[[1]]))
#> [1] "matrix" "array"
print(dim(l20[[1]]))
#> [1] 3 3

# Each component of the list is the
# matrix containing the N-A coefficients
print(l20[[1]])
#>       [,1]  [,2] [,3]
#> [1,] 0.000  0.00    0
```

```
#> [2,] 0.125 -0.75    0
#> [3,] 1.000  0.00    0
print(l20[[10]])
#>       [,1]       [,2]       [,3]
#> [1,] 0.000 0.11842105 0.09972299
#> [2,] 0.125 0.07894737 0.00000000
#> [3,] 1.000 0.00000000 0.00000000
print(l20[[20]])
#>       [,1] [,2] [,3]
#> [1,] 0.000 0.25    1
#> [2,] 0.125 1.00    0
#> [3,] 1.000 0.00    0
```

The object created, `l20`, is a list of matrices. We are interested in the component $(1,3)$ of each matrix in the list. How can we extract that into a vector `int02`, similar to the vector `int01` of interpolating values? This operation can be accomplished with an operation which is 'pure R style'. Basically, we need to use `sapply` again. The function needed inside `sapply` should be something to extract the $(1,3)$ matrix component of each matrix in the list. This turns out to be the `[` symbol (which in this case has the effect of an operator), followed by the two indices 1 and 3.

```
# Extract from each matrix what's in
# position (1,3)
int02 <- sapply(l20,`[`,1,3)
print(class(int02))   # It's a vector of
#> [1] "numeric"
print(length(int02))  # 20 numbers
#> [1] 20
attributes(int02) <- NULL   # Remove attributes to make printing nicer
print(int02)
#>  [1]  0.000000000 -0.022160665 -0.036011080 -0.041551247 -0.038781163
#>  [6] -0.027700831 -0.008310249  0.019390582  0.055401662  0.099722992
#> [11]  0.152354571  0.213296399  0.282548476  0.360110803  0.445983380
#> [16]  0.540166205  0.642659280  0.753462604  0.872576177  1.000000000

# Plot
plot(xp,fp,pch=16,cex=1.3,xlab="x",ylab="f(x)",
     ylim=c(-0.2,1),col=2)
points(x0,int01)
points(x0,int02,pch=16,cex=0.5)
```



Clearly, both interpolations yield an identical set of points.

3.2.5 Exercise 10

Consider the following six points,

$$x_1 = 0 \ , \ x_2 = 0.2 \ , \ x_3 = 0.3 \ , \ x_4 = 0.6 \ , \ x_5 = 0.7 \ , \ x_6 = 1,$$

part of the third degree polynomial,

$$f(x) = x^3 - 5x^2 + 2x + 1.$$

Verify that the values obtained with the Neville-Aitken algorithm start to be identical at its fourth level. Explain why this is the case.

SOLUTION

The exercise does not suggest any interpolating point. We can then pick a couple of points, say 0.35 and 0.75, and calculate the 6×6 Neville-Aitken triangular matrices, obtained using the six known points.

```
# Known points
xp <- c(0,0.2,0.3,0.6,0.7,1)
fp <- xp^3-5*xp^2+2*xp+1

# Using the interpolating point 0.35
P1 <- nevaitpol(xp,fp,0.35)

# Display the triangular matrix
print(P1)
```

```
#>         [,1]   [,2]    [,3]     [,4]     [,5]     [,6]
#> [1,]  1.000 1.3640 1.12775 1.130375 1.130375 1.130375
#> [2,]  1.208 1.1615 1.13225 1.130375 1.130375 0.000000
#> [3,]  1.177 1.0835 1.12600 1.130375 0.000000 0.000000
#> [4,]  0.616 1.4235 1.18725 0.000000 0.000000 0.000000
#> [5,]  0.293 1.8015 0.00000 0.000000 0.000000 0.000000
#> [6,] -1.000 0.0000 0.00000 0.000000 0.000000 0.000000

# Using the interpolating point 0.35
P2 <- nevaitpol(xp,fp,0.75)

# Display the triangular matrix
print(P2)
#>         [,1]   [,2]     [,3]     [,4]     [,5]     [,6]
#> [1,]  1.000 1.7800 -0.07625 0.109375 0.109375 0.109375
#> [2,]  1.208 1.0375  0.07225 0.109375 0.109375 0.000000
#> [3,]  1.177 0.3355  0.10600 0.109375 0.000000 0.000000
#> [4,]  0.616 0.1315  0.11125 0.000000 0.000000 0.000000
#> [5,]  0.293 0.0775  0.00000 0.000000 0.000000 0.000000
#> [6,] -1.000 0.0000  0.00000 0.000000 0.000000 0.000000
```

For both interpolating points the values in the `P` matrix start to be equal from the fourth column and above. As $4 = 3 + 1$, this means that the function interpolated is a third degree polynomial. The reason is that a third degree polynomial is completely and uniquely defined by four points. In this exercise, the Neville-Aitken algorithm was performed using six points, two points more than those needed. This is why the values in columns 4, 5 and 6 are all equal.

3.2.6 Exercise 11

Write the code to implement the Lagrange polynomial $P_n(x)$ to interpolate $n+1$ given values of a function $f(x)$. Apply the code created to find:

- $P_2(1.5)$ when $x_1 = 1$, $x_2 = 2$ and $f_1 \equiv \log(x_1)$, $f_2 \equiv \log(x_2)$
- $P_3(1.5)$ when $x_1 = 1$, $x_2 = 2$, $x_3 = 3$ and $f_1 \equiv \log(x_1)$, $f_2 \equiv \log(x_2)$, $f_3 \equiv \log(x_3)$
- $P_4(1.5)$ when $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, $x_4 = 4$ and $f_1 \equiv \log(x_1)$, $f_2 \equiv \log(x_2)$, $f_3 \equiv \log(x_3)$, $f_4 \equiv \log(x_4)$

Verify that the three values found coincide with the values in row 1 of the matrix P found using the Neville-Aitken algorithm on the last set of interpolation points given.

SOLUTION

In order to create the code required, we can start with the code existing to generate $P_2(x)$ (see Exercise 09). The second part of the function `P_2` can be easily generalised, while the $n+1$ points `xp` and $n+1$ values `fp` can be provided as input, besides `x`.

```
# Lagrange polynomial of order n. The input
# consists of the interpolating point x, the
# n+1 known points xp and n+1 known values fp
P_n <- function(x,xp,fp) {
   # Number of points
   n <- length(xp)

   # Create a vector and then sum its elements
   f <- c()
   for (i in 1:n) {
```

```
    # Top
    tt <- prod(x-xp[-i])  # xp without i-th element

    # Bottom
    bb <- prod(xp[i]-xp[-i])

    # Partial sum
    f <- c(f,tt*fp[i]/bb)
   }

   return(sum(f))
}

# Test previous function on the simple
# interpolations f(x)=x (points 0,1) and
# f(x)=x^2 (points 0,0.5,1), at 0.25. The
# correct interpolations return 0.25 and 0.0625
P_n(0.25,c(0,1),c(0,1))
#> [1] 0.25
P_n(0.25,c(0,0.5,1),c(0,0.25,1))
#> [1] 0.0625
```

The function just defined can be now used to acomplish the tasks requested.

```
# Interpolation (known) points
xp <- 1:4
fp <- log(xp)

# P_2(1.5)
P_n(1.5,xp[1:2],fp[1:2])
#> [1] 0.3465736

# P_3(1.5)
P_n(1.5,xp[1:3],fp[1:3])
#> [1] 0.3825338

# P_4(1.5)
P_n(1.5,xp[1:4],fp[1:4])
#> [1] 0.3931525
```

These three values should be equal to those in the first row of matrix `P` from the Neville-Aitken algorithm.

```
P <- nevaitpol(xp,fp,1.5)
print(P[1,])
#> [1] 0.0000000 0.3465736 0.3825338 0.3931525
```

The values match. This demonstrates how parts of the Neville-Aitken algorithm reproduce the interpolation calculated with Lagrange polynomials.

3.2.7 Exercise 12

Consider the set of known points and values of the function $\log(x)$:

$$x_i = i\ ,\ i = 1:100 \qquad , \qquad f_i = log(x_i)\ ,\ i = 1:100$$

Write a program which selects randomly 4 known points in the set $\{x_i\ ,\ i = 2, 99\}$, includes x_1 and x_100 as first and last known point and calculates interpolations $P_5(x_i)$ at the locations of the remaining 94 points,

using the Neville-Aitken algorithm. The program should calculate the two following quantities for each random set of interpolations:

$$\Delta P \equiv \langle |P_{1,5} - P_{2,5}| \rangle \qquad , \qquad \text{Err} \equiv \langle |f_i - P_5(x_i)| \rangle \, ,$$

where <> indicates the average corresponding to the 94 remaining points. Plot ΔP vs Err for 1000 simulations, i.e. 1000 random selections of the 6 known points and 94 remaining points. Repeat the exercise using 18 random points and the first and last sample point as before. What changes do you observe?

SOLUTION

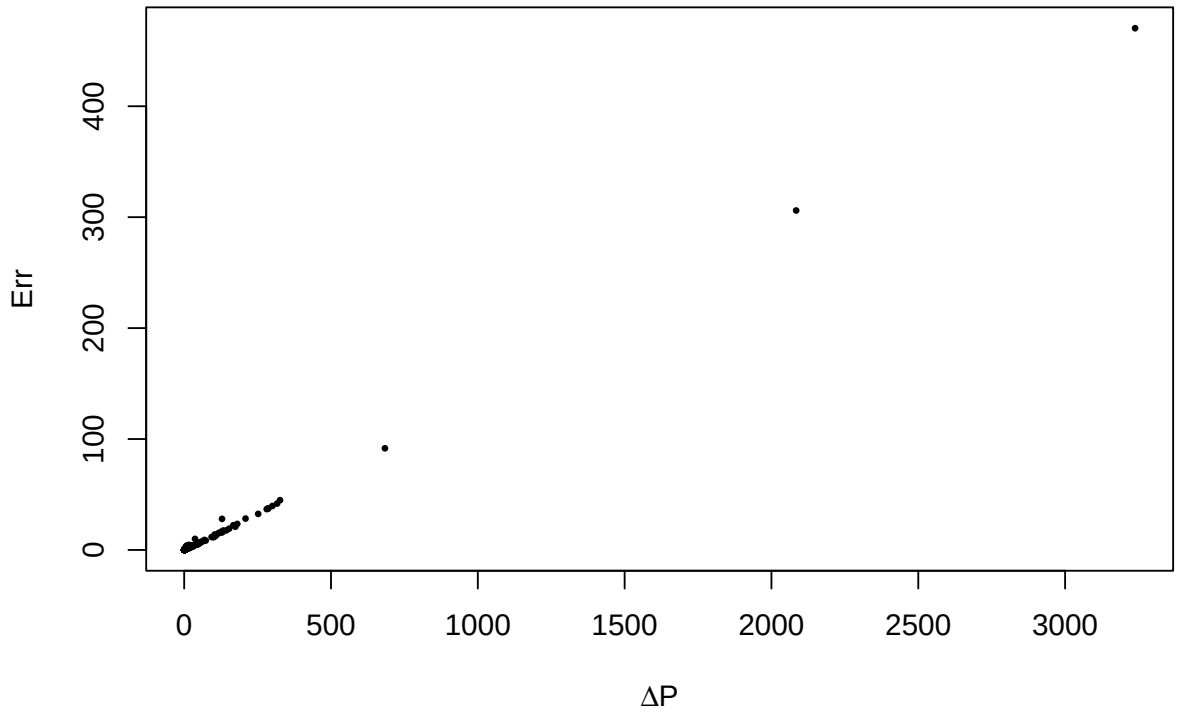
The script that produce the two sets of values is coded as follows.

```
# 100 sampled values for log(x)
xp <- 1:100
fp <- log(xp)

# Number of known points for interpolation
n <- 6

# Create vector of errors and interpolation differences
deltas <- c()
err <- c()

# Loop over 1000 random selections (using 'sample')
# First and last point are fixed.
set.seed(9361)  # To reproduce a "fixed" random simulation
for (i in 1:1000) {
  idx <- c(1,sample(2:99,size=n-2,replace=FALSE),100)
  x <- xp[idx]
  f <- fp[idx]
  x0 <- xp[-idx]
  lNA <- sapply(x0,nevaitpol,x=x,f=f,simplify=FALSE)
  ints <- sapply(lNA,`[`,1,n)
  ups <- sapply(lNA,`[`,1,n-1)
  downs <- sapply(lNA,`[`,2,n-1)
  deltas <- c(deltas,mean(abs(ups-downs)))
  err <- c(err,mean(abs(ints-fp[-idx])))
}

# Both deltas and err include 94 values
str(deltas)
#>  num [1:1000] 0.431 0.823 1.382 0.351 0.287 ...
str(err)
#>  num [1:1000] 0.127 0.188 0.264 0.103 0.149 ...

# Summary statistics
summary(deltas)
#>      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
#>    0.1967    0.2687    0.4207   13.4321    1.8723 3238.4091
summary(err)
#>      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
#>   0.08053   0.11024   0.15868   1.95981   0.40135 470.33461
```

While both sets of values have a high maximum, their median is relatively low. This indicates the maxima as outliers. Those might have happened when, for instance, the six known points were very close with each

other and relatively far from many of the remaining 94 points. In such circumstances the interpolation does not work well. The plot of Err versus ΔP is produced here.

```
plot(deltas,err,pch=16,cex=0.5,xlab=expression(paste(Delta,"",P)),
     ylab="Err")
```

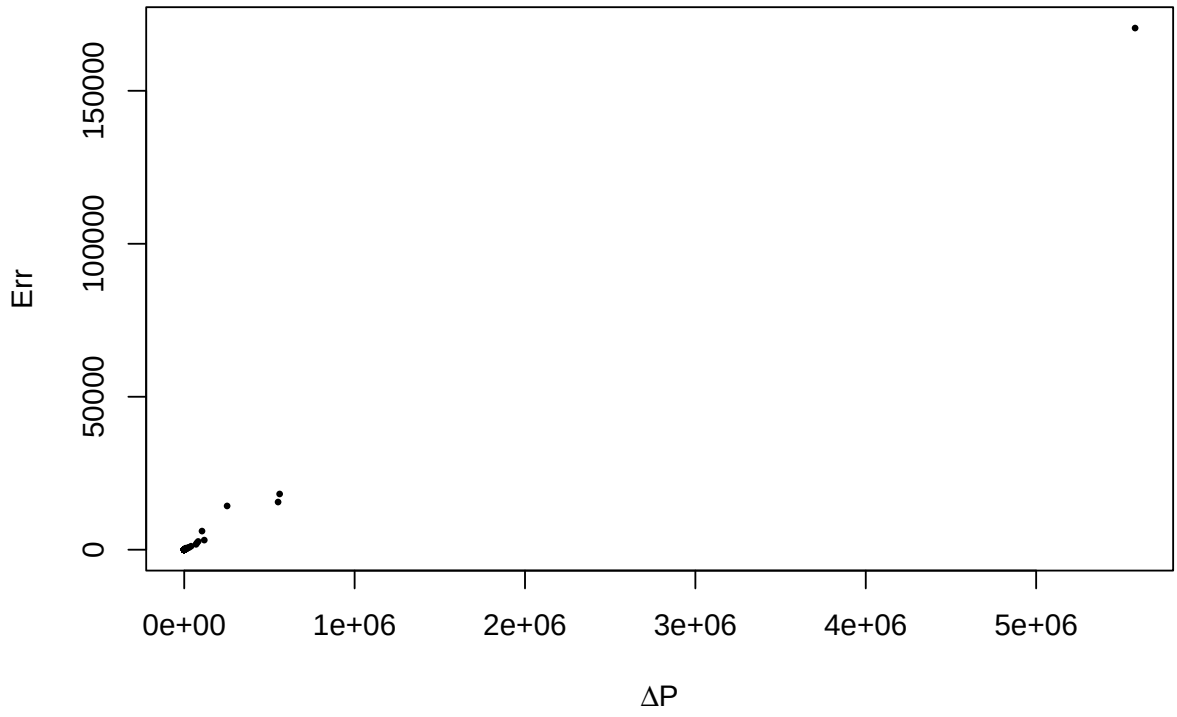


There is a clear direct proportional relation between the two quantities. The difference of the two values in the one-before-last column of the P matrix can therefore give an indication on how the interpolating value is close to the correct one. Further investigations and empirical trials could give a quantitatively better defined measure of such an indication.

We can repeat the previous set of instructions, this time using $n = 20$.

```
# Number of known points for interpolation
n <- 20

# Create vector of errors and interpolation differences
deltas <- c()
err <- c()

# Loop over 1000 random selections (using 'sample')
# First and last point are fixed.
set.seed(9361)  # To reproduce a "fixed" random simulation
for (i in 1:1000) {
  idx <- c(1,sample(2:99,size=n-2,replace=FALSE),100)
  x <- xp[idx]
  f <- fp[idx]
```

```
  x0 <- xp[-idx]
  lNA <- sapply(x0,nevaitpol,x=x,f=f,simplify=FALSE)
  ints <- sapply(lNA,`[`,1,n)
  ups <- sapply(lNA,`[`,1,n-1)
  downs <- sapply(lNA,`[`,2,n-1)
  deltas <- c(deltas,mean(abs(ups-downs)))
  err <- c(err,mean(abs(ints-fp[-idx])))
}

# Both deltas and err include 94 values
str(deltas)
#>  num [1:1000] 0.21917 0.01521 0.00519 0.00672 0.03788 ...
str(err)
#>  num [1:1000] 0.00568 0.00235 0.00227 0.00256 0.01597 ...

# Summary statistics
summary(deltas)
#>      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
#>       0.0       0.0       0.0    7617.5       0.4 5580769.1
summary(err)
#>      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
#> 1.000e-03 3.000e-03 4.000e-03 2.411e+02 1.600e-02 1.705e+05

# Plot
plot(deltas,err,pch=16,cex=0.5,xlab=expression(paste(Delta,"",P)),
     ylab="Err")
```



Values seem to be less spread than before. The median is, in fact, too small for the printing precision of the function summary.

```
print(median(deltas))
#> [1] 0.01472302
print(median(err))
#> [1] 0.004409743
```

These values are smaller than before, although the outliers are bigger because 18 known points clustered very close can yield very wrong interpolations far from where they are located. The result obtained using 20 known points yield interpolations closer to the correct values than interpolation done using only 6 known points.

3.3 Exercises on divided differences

3.3.1 Exercise 13

Find the divided differences for the points tabulated.

x	$f(x)$
1.000	0.000
2.500	1.833
3.000	2.197
4.000	2.773
4.500	3.008

What is the value for $x = 1.5$ using all five points? And using only the first four points?

SOLUTION

The divided differences can be easily found using the function `divdif`.

```
# Tabulated points
x <- c(1,2.5,3,4,4.5)
f <- c(0,1.833,2.197,2.773,3.008)

# Divided differences
P <- divdif(x,f)
print(P)
#>       [,1]  [,2]        [,3]       [,4]         [,5]
#> [1,] 0.000 1.222 -0.24700000 0.04855556 -0.009492063
#> [2,] 1.833 0.728 -0.10133333 0.01533333  0.000000000
#> [3,] 2.197 0.576 -0.07066667 0.00000000  0.000000000
#> [4,] 2.773 0.470  0.00000000 0.00000000  0.000000000
#> [5,] 3.008 0.000  0.00000000 0.00000000  0.000000000
```

Interpolation values can be calculated using `polydivdif`.

```
# Use all five points for the interpolation
x0 <- 1.5
lDD <- polydivdif(x0,x,f)
f0 <- lDD$f0
print(f0)
#> [1] 0.7887143

# Use first 4 tabulated points
lDD <- polydivdif(x0,x[1:4],f[1:4])
f0 <- lDD$f0
print(f0)
#> [1] 0.7709167
```

3.3.2 Exercise 14

Find the coefficients $a_1, \ldots, a_6$ of the function,

$$f(x) = a_1 + a_2(x+1) + a_3(x+1)(x-1) + a_4(x+1)(x-1)(x-2) + a_5(x+1)(x-1)(x-2)(x-4) + a_6(x+1)(x-1)(x-2)(x-4)(x-5),$$

equal to the following polynomial,

$$P_5(x) = x^5 - 2x^4 - x^3 + 3x^2 - 6$$

SOLUTION

The unknown function $f(x)$ is a fifth-degree polynomial, equal to $P_5(x)$. Its coefficients could be found by expanding the products, re-arranging the obtained expression in terms of increasing powers of x and equating the coefficients of equal powers of x. The result is a linear system of six linear equations in six unknowns. This way of proceeding is time consuming. Alternatively, we can proceed using the divided differences because $f(x)$ has an analytical form whose coefficients are the divided differences for the function passing through 6 known points. Five of them are clearly readable from the factorisation in the expression for $f(x)$. The sixth one can be chosen arbitrarily; we could choose, for instance,

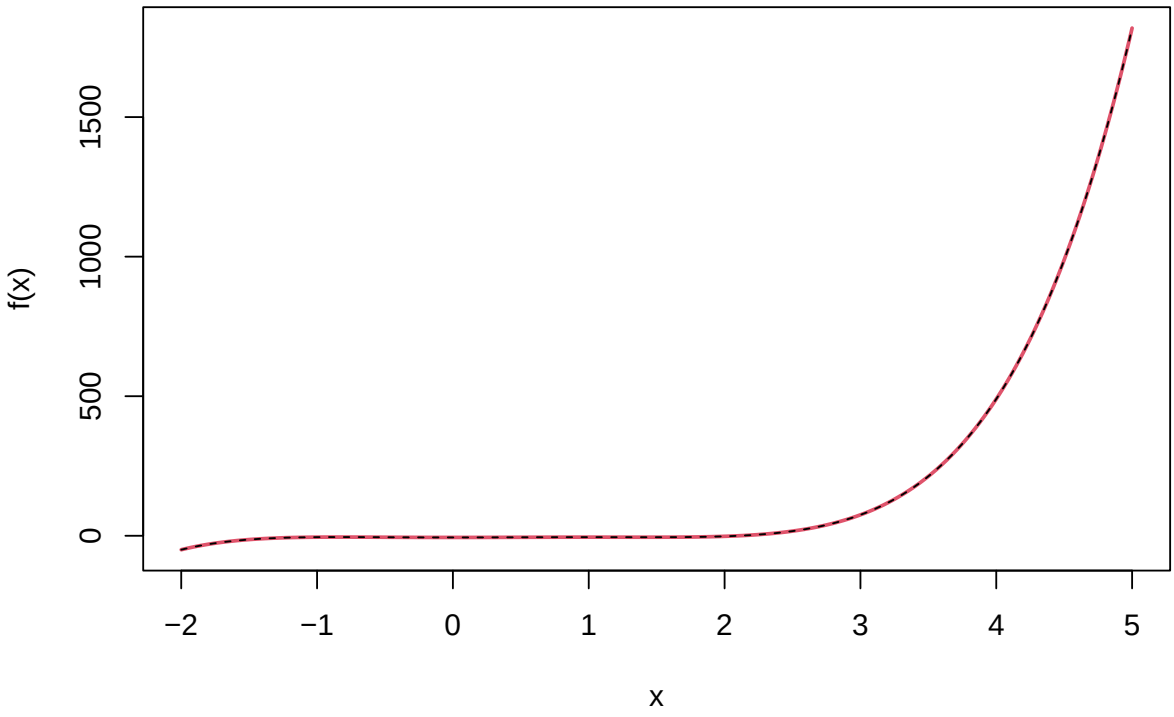
$$x_1 = -2\ ,\ x_2 = -1\ ,\ x_3 = 1\ ,\ x_4 = 2\ ,\ x_5 = 4\ ,\ x_6 = 5$$

The calculation can be done without effort in R.

```
# Tabulated points (using P_5(x))
x <- c(-2,-1,1,2,4,5)
f <- x^5-2*x^4-x^3+3*x^2-6

# Divided differences
P <- divdif(x,f)
print(P)
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]  -50   45  -15    4    2    1
#> [2,]   -5    0    1   16    9    0
#> [3,]   -5    3   81   70    0    0
#> [4,]   -2  246  361    0    0    0
#> [5,]  490 1329    0    0    0    0
#> [6,] 1819    0    0    0    0    0
```

Thus the six coefficients $a_1, \ldots, a_6$ are -50, 45, -15, 4, 2, 1. To verify that these are actually the correct coefficients, let's plot both $f(x)$ and $P_5(x)$ on a grid between $x = -2$ and $x = 5$.

```
# Fine grid
x0 <- seq(-2,5,length.out=1000)
f0 <- x0^5-2*x0^4-x0^3+3*x0^2-6

# Interpolation using divided differences
lDD <- polydivdif(x0,x,f)
f1 <- lDD$f0

# Overlapping plots
plot(x0,f0,type="l",col=2,lwd=2,xlab="x",
     ylab="f(x)")
points(x0,f1,type="l",lty=2,col=1)
```



From the plot it is clear that the coefficients found are correct because the curves overlap.

3.3.3 Exercise 15

The function,

$$f(x) = \cos(x/2) - \sin(x)$$

in the interval $[-2\pi, 2\pi]$ can be interpolated by a four degrees polynomial, $Q_4(x)$, passing through the points x_1, x_2, x_3, x_4, x_6 where,

$$x_1 = -2\pi,\ x_2 = -\pi,\ x_3 = 0,\ x_4 = \pi/2,\ x_6 = 2\pi$$

Using knowledge of $f(x)$ at $x_5 = \pi$, compute the error,

$$\Delta Q_4(x) \equiv f(x) - Q_4(x)$$

at $x = (3/2)\pi$, using the next-term rule.

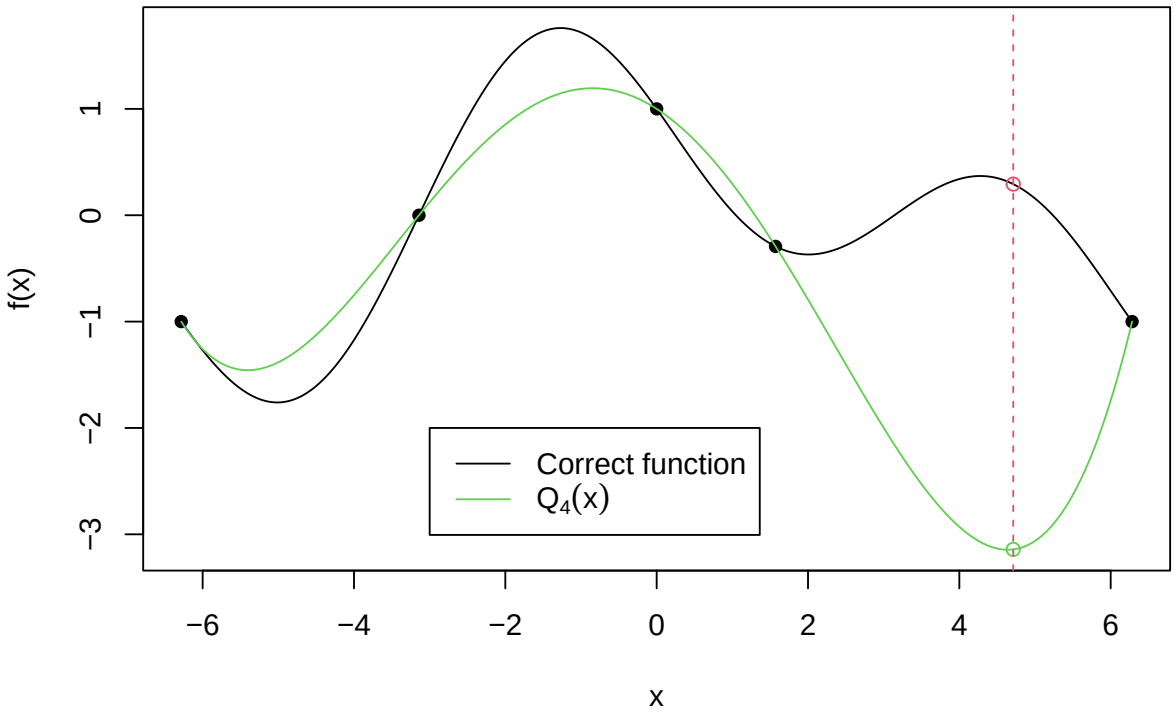
SOLUTION In this exercise we are required to interpolate functions, polynomials and errors at the specific point, $(3/2)\pi$. So, in order to use function `polydivdif`, we need to provide a grid which includes it. First we carry out the calculations for $Q_4(x)$.

```
# Tabulated points (x5 is the added point)
x <- c(-2*pi,-pi,0,pi/2,2*pi,pi)
f <- cos(0.5*x)-sin(x)

# Appropriate fine grid including 1.5pi
x0 <- seq(-2*pi,2*pi,by=0.01*pi)
```

```r
# Interpolation using divided differences
lDD <- polydivdif(x0,x[1:5],f[1:5])
f0 <- lDD$f0  # This is Q4(x)

# Difference between correct value and interpolation
corr_value <- cos(3*pi/4)-sin(3*pi/2)
print(corr_value)
#> [1] 0.2928932
idx <- which(abs(x0-1.5*pi) < 1e-6) # Identify where (3/2)pi is
intQ4_value <- f0[idx]
print(intQ4_value)
#> [1] -3.141751
```

The interpolation seems to provide a value not close to the correct value. This is not surprising because the interpolation at $x = 1.5\pi$ happens far from the two closest tabulated points, $x_4 = \pi/2$ and $x_6 = 2\pi$. A picture of the polynomial $Q_4(x)$ overlapping with the correct function $f(x)$ can further convince us of this.

```r
# Correct function
ftrue <- cos(x0/2)-sin(x0)

# Plot
prange <- range(ftrue,f0)
plot(x0,ftrue,type="l",xlab="x",ylab="f(x)",ylim=prange)
points(x[1:5],f[1:5],pch=16)
points(1.5*pi,cos(1.5*pi/2)-sin(1.5*pi),col=2)
points(x0,f0,type="l",col=3)
points(1.5*pi,f0[idx],col=3)
abline(v=1.5*pi,lty=2,col=2)
legend(-3,-2,legend=c("Correct function",expression(Q[4](x))),
       col=c(1,3),lty=c(1,1))
```



The exact error, $f(x) - \Delta P_4(x)$, at $x = (3/2)\pi$, is 3.4346441. Its estimate using the next-term rule can be calculated when $x_5 = \pi$ is added as new point:

$$\Delta Q_4(x) = f[x_1, x_2, x_3, x_4, x_6, x_5](x - x_1)(x - x_2)(x - x_3)(x - x_4)(x - x_6)$$

Indeed:

```
# Divided differences with the additional points
x <- c(x,1.5*pi)
f <- c(f,cos(1.5*pi/2)-sin(1.5*pi))
P <- divdif(x,f)

# Calculated error with next-term rule at x=(3/2)pi
DelQ4_pi <-
  P[1,6]*(1.5*pi+2*pi)*(1.5*pi+pi)*(1.5*pi)*(1.5*pi-pi/2)*(1.5*pi-2*pi)

# The value obtained is reasonably close
# to the correct value, as expected
print(corr_value-intQ4_value)  # Correct error value
#> [1] 3.434644
print(DelQ4_pi)                # Estimated error value
#> [1] 4.721002
```

3.3.4 Exercise 16

Considering the expression (3.18) for the interpolation using divided differences, prove that the following recurring equation is correct:

$$Q_n(x) = Q_{n-1}(x) + \frac{f_{n+1} - Q_{n-1}(x_{n+1})}{(x_{n+1} - x_1)\cdots(x_{n+1} - x_n)}(x - x_1)\cdots(x - x_n)$$

SOLUTION First of all, the polynomial $Q_n(x)$ is built so to satisfy,

$$Q_n(x_1) = f_1 \ , \ Q_n(x_2) = f_2 \ , \ \ldots \ , \ Q_n(x_n) = f_n \ , \ Q_n(x_{n+1}) = f_{n+1}$$

Second, from the expression (3.18) both for $Q_{n-1}(x)$ and $Q_n(x)$, it follows that,

$$Q_n(x) = Q_{n-1}(x) + f[x_1, \ldots, x_{n+1}](x - x_1)\cdots(x - x_n) \qquad (*)$$

The above expression, written with $x = x_{n+1}$, yields,

$$Q_n(x_{n+1}) = Q_{n-1}(x_{n+1}) + f[x_1, \ldots, x_{n+1}](x_{n+1} - x_1)\cdot(x_{n+1} - x_n)$$

Recalling now that $Q_n(x_{n+1}) = f_{n+1}$ and re-arranging the terms in the last expression, we can write the divided difference as a function of the polynomial of degree $n - 1$:

$$f[x_1, \ldots, x_{n+1} = \frac{f_{n+1} - Q_{n-1}(x_{n+1})}{(x_{n+1} - x_1)\cdots(x_{n+1} - x_n)}$$

If the expression for the divided difference just obtained is inserted in equation (*), the required relation is obtained.

3.3.5 Exercise 17

Consider the following known points of a function $f(x)$:

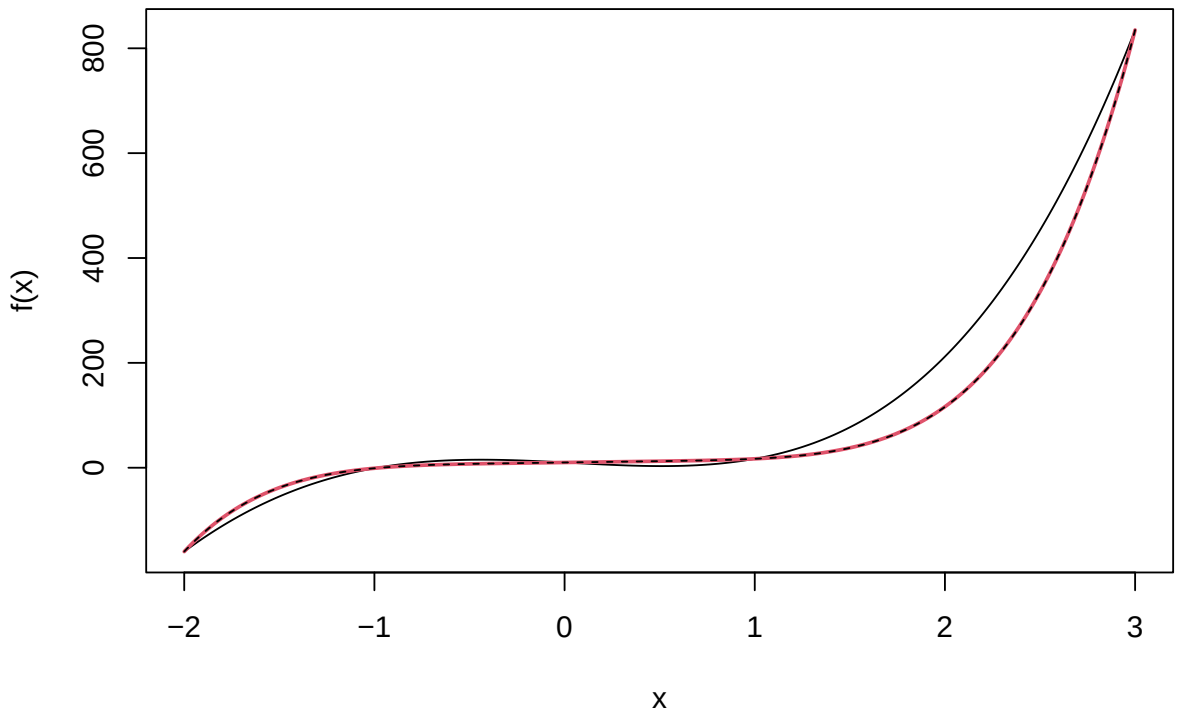
x	$f(x)$
-2	-160
-1	-1
0	10
1	17
3	835
2	116

In a regular grid containing 1000 points between -2 and 3, calculate the fourth-degree interpolating polynomial $Q_4(x)$ using the first five values tabulated and the divided differences. Next, calculate the fifth-degree interpolating poynomial $Q_5(x)$ using the last tabulated value and the formula introduced in the previous exercise. Finally, compute $Q_5(x)$ via divided differences using all six tabulated points. Plot the three curves found and verify that the last two curves coincide.

SOLUTION The first task is straightforward.

```
# Tabulated points
x <- c(-2,-1,0,1,3,2)
f <- c(-160,-1,10,17,835,116)

# x grid
x0 <- seq(-2,3,length.out=1000)
```

```r
# Q4(x) using divided differences
lDD <- polydivdif(x0,x[1:5],f[1:5])
q4 <- lDD$f0

# Q5(x) using new formula

# Need to identify closest point to 2
idx <- which(abs(x0-2) == min(abs(x0-2)))
top <- f[6]-q4[idx]
bot <- (x[6]-x[1])*(x[6]-x[2])*(x[6]-x[3])*(x[6]-x[4])*(x[6]-x[5])
est_DD <- top/bot  # Estimated divided difference
q5new <- q4+est_DD*
         (x0-x[1])*(x0-x[2])*(x0-x[3])*(x0-x[4])*(x0-x[5])

# Q5(x) using divided differences
lDD <- polydivdif(x0,x,f)
q5 <- lDD$f0

# Plot
frange <- range(q4,q5new,q5)
plot(x0,q4,type="l",xlab="x",ylab="f(x)",ylim=frange)
points(x0,q5new,type="l",col=2,lwd=2)
points(x0,q5,type="l",lty=2,col=1)
```



It is clear from the plot that the two curves `q5` and `q5new` overlap, thus showing that the formula given in

the previous exercise is correct. We can also check that the divided difference is equal to the variable `est_DD` calculated with the new formula. In fact, the two variables will not be exactly equal because the estimated divided difference, using the new formula, has been computed with $Q_4(x_{n+1})$ approximately equal to the correct value because in the grid `x0` created, point 2 was not exactly included.

```
# Correct divided difference
P <- divdif(x,f)
print(P[1,6])
#> [1] 4

# Approximate divided difference
print(est_DD)
#> [1] 3.984785
```

3.4 Exercises on cubic splines

3.4.1 Exercise 18

Interpolate, using a grid of 200 points in the interval $x \in [-4, 4]$, the gaussian function,

$$f(x) = \exp\left(-\frac{x^2}{10}\right),$$

where the known points are,

$$x_1 = -4\ ,\ x_2 = -2\ ,\ x_3 = -1\ ,\ x_4 = -1/2\ ,\ x_5 = -1/4$$
$$x_6 = 0\ ,\ x_7 = 1/4\ ,\ x_8 = 1/2\ ,\ x_9 = 1\ ,\ x_{10} = 2\ ,\ x_{11} = 4,$$

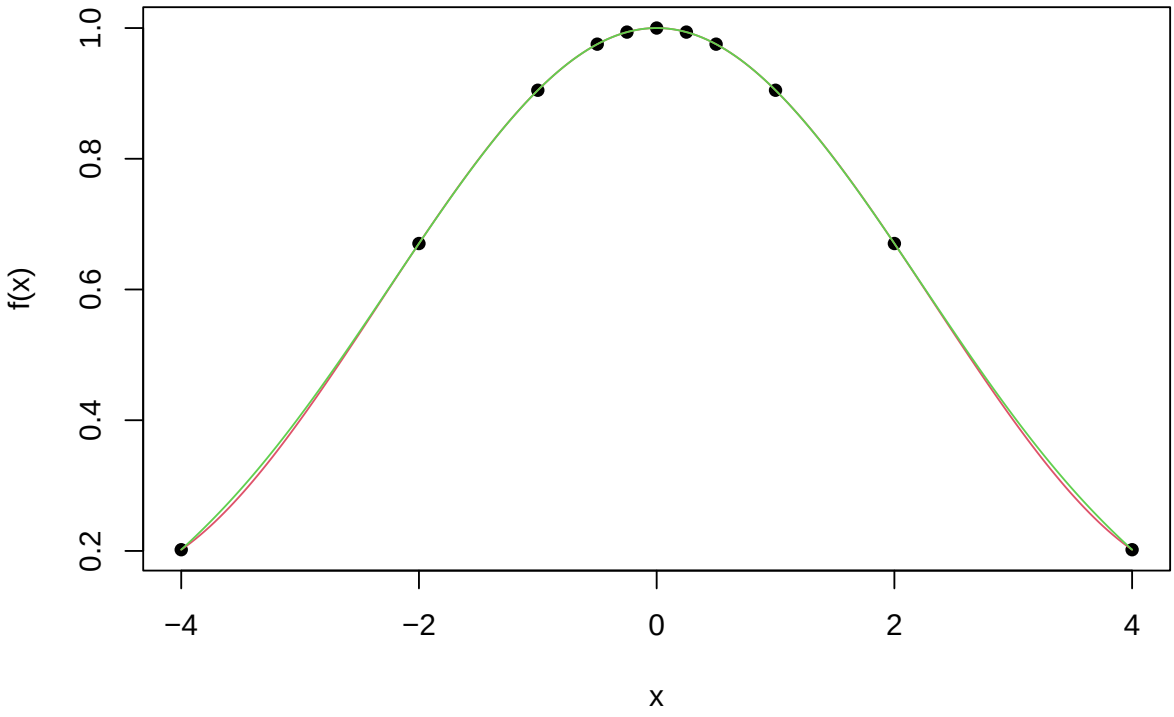
using cubic splines with the Forsythe, Malcolm and Moler method for end segments. Compare graphically the cubic splines with a polynomial fit using divided differences.

SOLUTION First the two vectors, `x,f` of known values will be created. Next, the required grid will also be created using `seq`. Finally, the cubic spline will be calculated using `spline`.

```
# Known points
x <- c(-4,-2,-1,-1/2,-1/4,0,1/4,1/2,1,2,4)
f <- exp(-x^2/10)

# Fine grid with 200 points
x0 <- seq(-4,4,length.out=200)

# Cubic spline (method "fmm" is default)
lCS <- spline(x,f,xout=x0)
```

We can now proceed to compute a polynomial fitting, using `polydivdif`. Then the two interpolating curves can be compared graphically.

```
# Interpolating polynomial using divided differences
lDD <- polydivdif(x0,x,f)

# Compare curves graphically
frange <- range(lCS$y,lDD$f0)
plot(x,f,pch=16,ylim=frange,xlab="x",ylab="f(x)")
points(x0,lCS$y,type="l",col=2)
points(x0,lDD$f0,type="l",col=3)
```



From the picture produced, it is possible to see that in the case presented polynomial interpolation yields results comparable to cubic spline interpolation.

3.4.2 Exercise 19

Given the same known x points of Exercise 18, apply them to function,

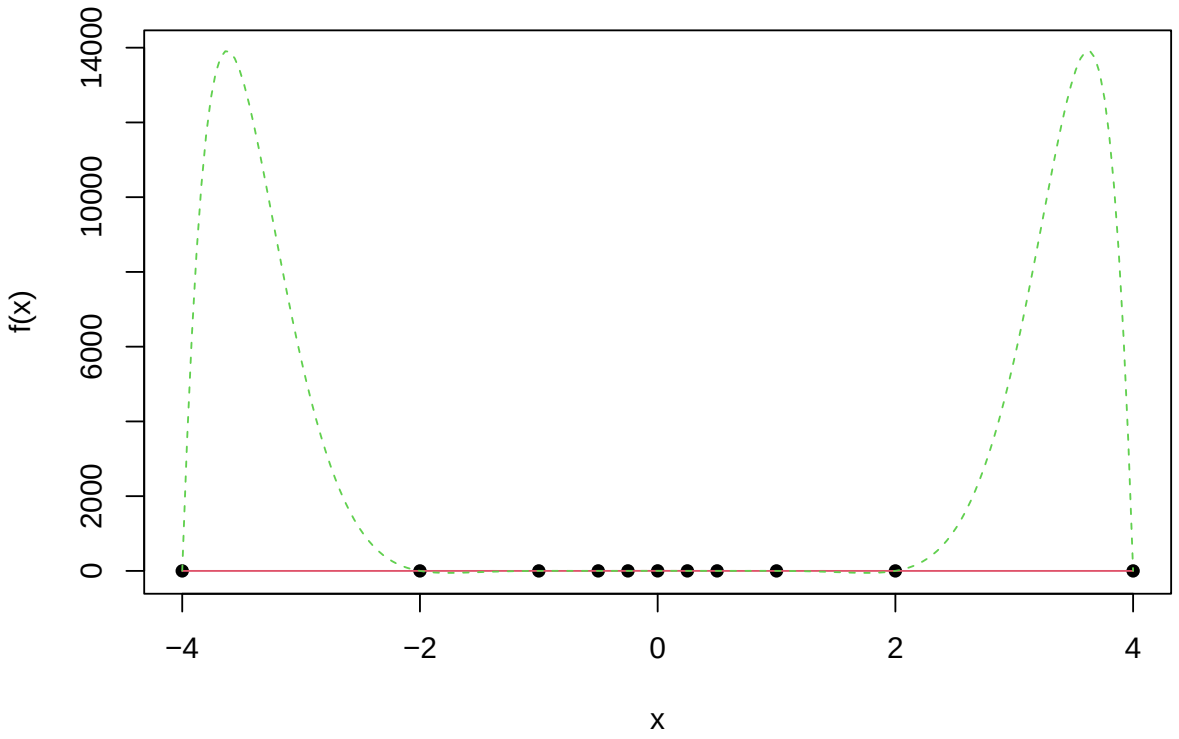
$$f(x) = exp(-x^2/0.1)$$

which has a narrower peak than the gaussian of Exercise 17. Carry out the same interpolation and comparison done in Exercise 17, on this new set of points.

SOLUTION We will proceed as in the previous exercise.

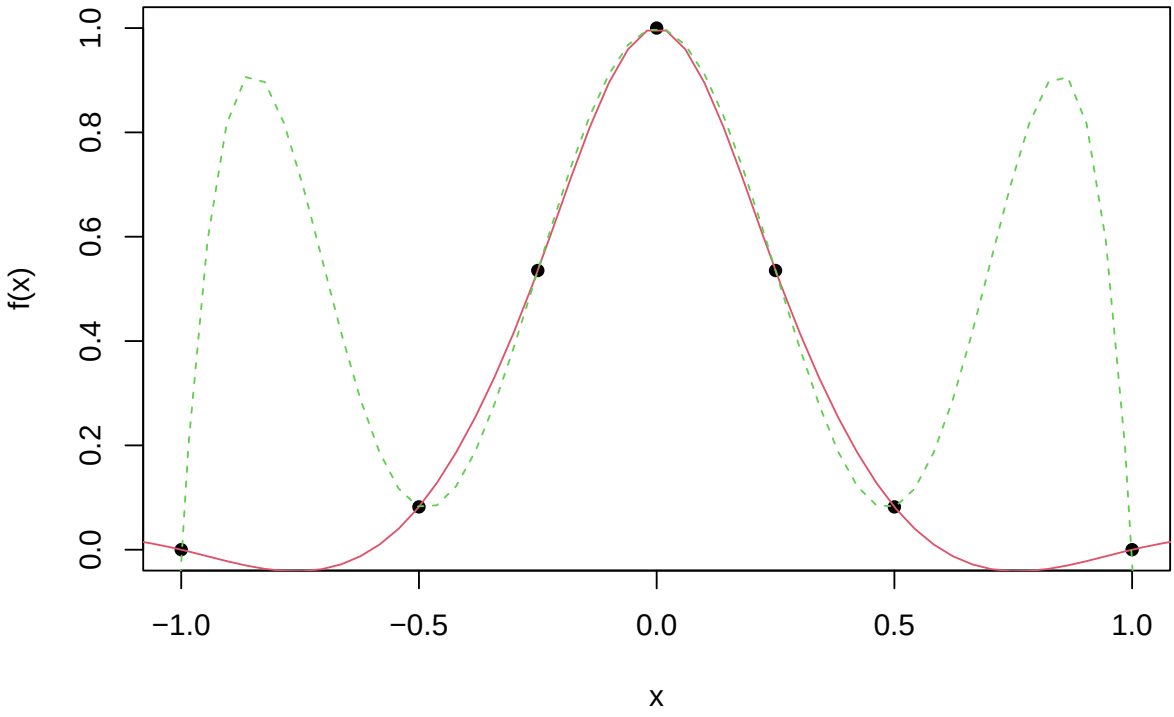
```
x <- c(-4,-2,-1,-1/2,-1/4,0,1/4,1/2,1,2,4)
f <- exp(-x^2/0.1)

# Fine grid with 200 points
x0 <- seq(-4,4,length.out=200)

# Cubic spline (method "fmm" is default)
lCS <- spline(x,f,xout=x0)

# Interpolating polynomial using divided differences
lDD <- polydivdif(x0,x,f)

# Compare curves graphically
```

```
frange <- range(lCS$y,lDD$f0)
plot(x,f,pch=16,ylim=frange,xlab="x",ylab="f(x)")
points(x0,lCS$y,type="l",col=2)
points(x0,lDD$f0,type="l",col=3,lty=2)
```



The picture obtained this time is deceiving because the polynomial interpolation (in dashed green) is swamping the cubic interpolation. This happens because polynomials are not good interpolators for functions mostly flat and with just a few peaks, like the one presented here. A zooming of the interpolation near $x = 0$ can show how good cubic interpolation (in red) actually is and how bad the polynomial one is still close to zero.

```
plot(x,f,pch=16,xlim=c(-1,1),
     xlab="x",ylab="f(x)")
points(x0,lCS$y,type="l",col=2)
points(x0,lDD$f0,type="l",col=3,lty=2)
```



3.4.3 Exercise 20

Using the `comphy` function `polydivdif`, demonstrate visually that the first and last segment of a cubic spline used to interpolate,

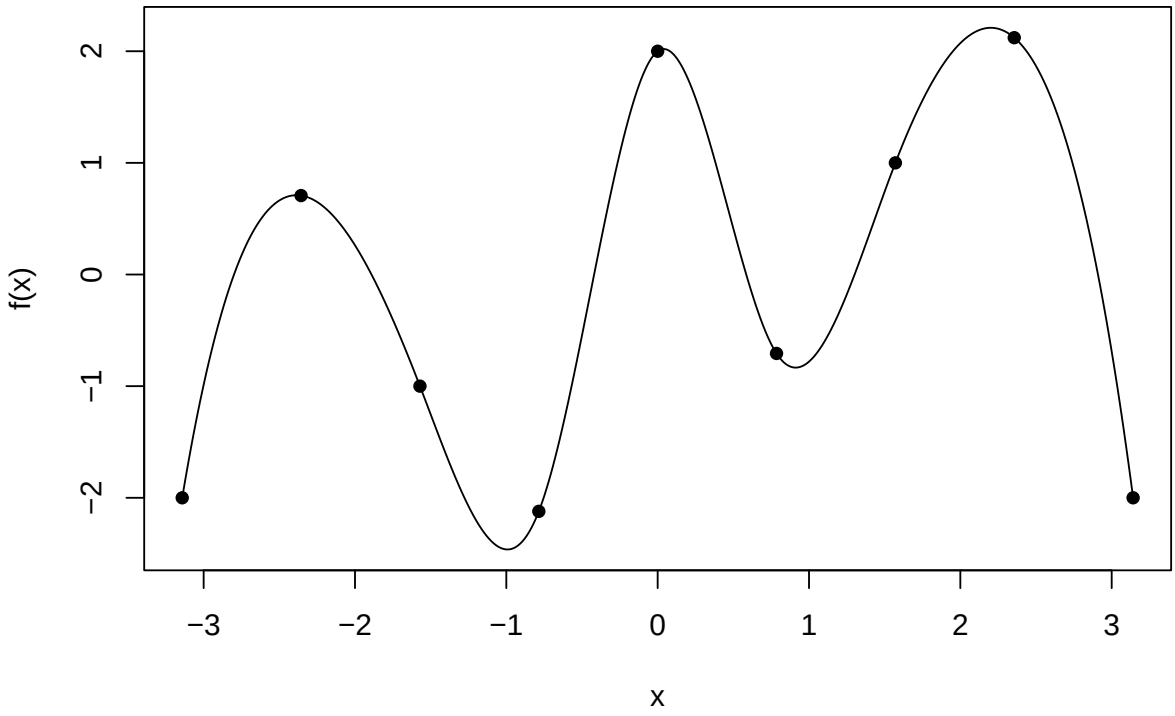
$$x_i = -\pi + (i-1)\pi/4\ ,\ i = 1, \ldots, 9$$
$$f_i = \sin(x_i) + 2\cos(3x_i)\ ,\ i = 1, \ldots, 9$$

with the Forsythe, Malcolm and Moler method, belong to the cubic polynomials passing respectively through the first and last four known points. You can use a regular fine grid containing 1000 points for interpolation.

SOLUTION The cubic spline interpolation is readily done, once the known points are provided. These are, as it is easy to verify, nine points regularly-spaced between $-\pi$ and π.

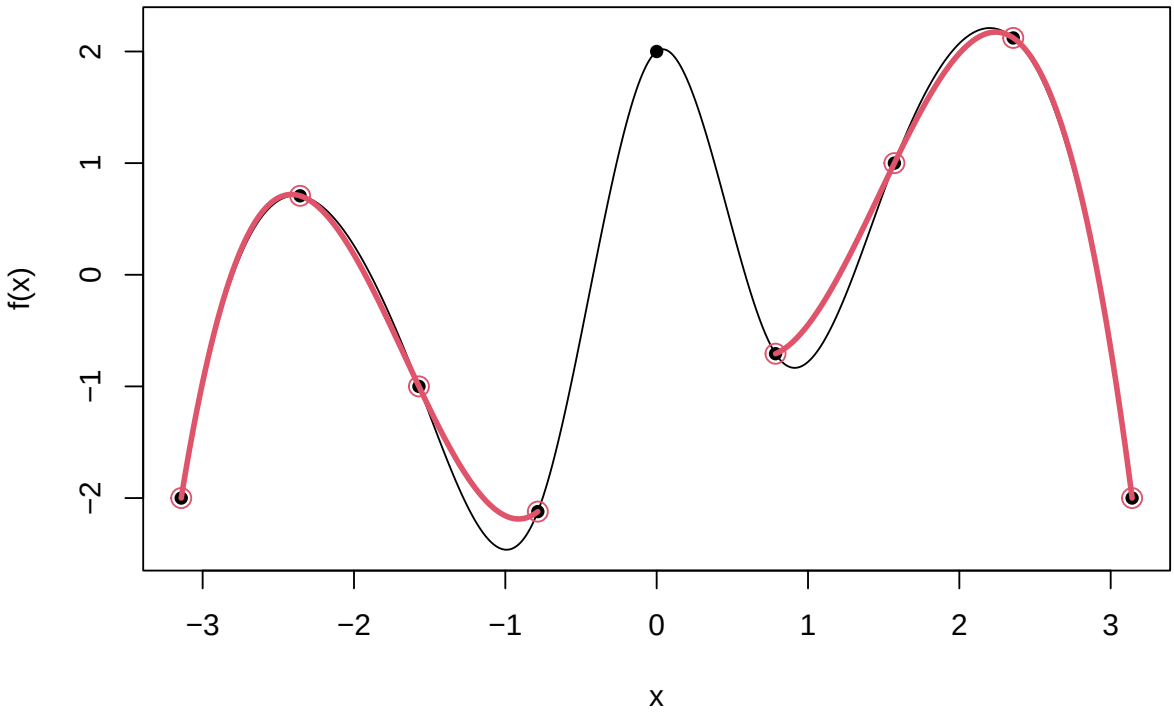
```
# Known points
x <- seq(-pi,pi,length.out=9)
print(x)
#> [1] -3.1415927 -2.3561945 -1.5707963 -0.7853982  0.0000000  0.7853982  1.5707963
#> [8]  2.3561945  3.1415927
f <- sin(x)+2*cos(3*x)

# Fine grid
x0 <- seq(-pi,pi,length.out=1000)

# Cubic spline with "fmm" method
lCS <- spline(x,f,xout=x0)
```

```
# Plot
plot(x0,lCS$y,type="l",xlab="x",ylab="f(x)")
points(x,f,pch=16)
```



To demonstrate what required by the exercise, polynomials calculated with `polydivdif` using the first and last four points can be calculated and plotted as thicker lines on the previous plot. The first and last cubic spline segments should match completely these polynomials.

```
# Polynomial for first four points
x01 <- seq(x[1],x[4],length.out=200)
lDD1 <- polydivdif(x01,x[1:4],f[1:4])

# Polynomial for last four points
x02 <- seq(x[6],x[9],length.out=200)
lDD2 <- polydivdif(x02,x[6:9],f[6:9])

# Do polynomials overlap splines?
plot(x0,lCS$y,type="l",xlab="x",ylab="f(x)")
points(x,f,pch=16)
points(x[1:4],f[1:4],cex=1.5,col=2)
points(x01,lDD1$f0,type="l",lwd=3,col=2)
points(x[6:9],f[6:9],cex=1.5,col=2)
points(x02,lDD2$f0,type="l",lwd=3,col=2)
```



The first and last segment are completely matched by the calculated polynomials. They do not cover the adjacent segments, though, because these are not calculated using the Forsythe, Malcolm and Moler method, but with the regular joining conditions for cubic splines.

4 Chapter 04

4.1 Exercises on systems of linear equations

4.1.1 Exercise 01

Find the solution of the following system of five equations and five unknowns:

$$\begin{cases} 2x_1 + x_2 - 3x_3 - 5x_4 + x_5 &= -4 \\ 7x_1 - x_2 - 5x_5 &= 3 \\ -6x_1 + 8x_2 - 2x_4 - 4x_5 &= -2 \\ -3x_1 + 8x_2 + x_3 + 8x_4 + 6x_5 &= 0 \\ 5x_1 + 2x_2 + 7x_3 + 8x_4 - 2x_5 &= -2 \end{cases}$$

using the function `gauss_elim`. Verify the solution found by substitution in the original system (use R code).

SOLUTION

The system in matrix form,

$$A\mathbf{x} = \mathbf{b},$$

yields,

$$A = \begin{pmatrix} 2 & 1 & -3 & -5 & 1 \\ 7 & -1 & 0 & 0 & -5 \\ -6 & 8 & 0 & -2 & -4 \\ -3 & 8 & 1 & 8 & 6 \\ 5 & 2 & 7 & 8 & -2 \end{pmatrix} \quad , \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \quad , \quad \mathbf{b} = \begin{pmatrix} -4 \\ 3 \\ -2 \\ 0 \\ -2 \end{pmatrix}$$

To use Gaussian elimination we will have to create the augmented matrix,

$$M = (A|\mathbf{b}) = \begin{pmatrix} 2 & 1 & -3 & -5 & 1 & -4 \\ 7 & -1 & 0 & 0 & -5 & 3 \\ -6 & 8 & 0 & -2 & -4 & -2 \\ -3 & 8 & 1 & 8 & 6 & 0 \\ 5 & 2 & 7 & 8 & -2 & -2 \end{pmatrix}$$

The function then returns the solution straight away in terms of components of a vector.

```
# Augmented matrix
raw_data <- c( 2, 1,-3,-5, 1,-4,
               7,-1, 0, 0,-5, 3,
              -6, 8, 0,-2,-4,-2,
              -3, 8, 1, 8, 6, 0,
               5, 2, 7, 8,-2,-2)
M <- matrix(raw_data,ncol=6,byrow=TRUE) # Data above input
                                        # row by row

# Gaussian elimination
sols <- gauss_elim(M)
print(sols) # Numbers displayed with a default precision
#> [1] -0.3293013 -0.6129265 -1.7584402  1.4130710 -0.9384365
```

The solution found, if correct, should make the product $A\mathbf{x}$ as close as possible to $\mathbf{b}$.

```
# Solution as a column vector
x <- matrix(sols,ncol=1)

# Ax should be equal to b
print(M[,6])
#> [1] -4  3 -2  0 -2
print(M[,1:5] %*% x)
#>      [,1]
#> [1,]   -4
#> [2,]    3
#> [3,]   -2
#> [4,]    0
#> [5,]   -2
```

$A\mathbf{x}$ is indeed equal to $\mathbf{b}$.

4.1.2 Exercise 02

Use `gauss_elim` to verify that the system,

$$\begin{cases} 3x_1 + 2x_2 - x_3 - 4x_4 &= 10 \\ x_1 - x_2 + 3x_3 - x_4 &= -4 \\ 2x_1 + x_2 - 3x_3 &= 16 \\ -x_2 + 8x_3 - 5x_4 &= 3 \end{cases}$$

has either no solution, or an infinite number of solutions. Use next the function `transform_upper` and quantitative reasoning to prove that the system has, in fact, no solution.

SOLUTION

All we need to carry out Gaussian elimination is the augmented matrix which is, in this case,

$$M = (A|\mathbf{b}) = \begin{pmatrix} 3 & 2 & -1 & -4 & 10 \\ 1 & -1 & 3 & -1 & -4 \\ 2 & 1 & -3 & 0 & 16 \\ 0 & -1 & 8 & -5 & 3 \end{pmatrix}$$

```
# Augmented matrix
raw_data <- c( 3, 2,-1,-4, 10,
               1,-1, 3,-1, -4,
               2, 1,-3, 0, 16,
               0,-1, 8,-5,  3)
M <- matrix(raw_data,ncol=5,byrow=TRUE)

# Gaussian elimination
sols <- gauss_elim(M)
#> This system has no solution or infinite solutions.
```

Clearly the system has either no solution or an infinite number of solutions, as commented in the output. In order to find out which alternative is true, let us write the system in upper diagonal form, using the function `transform_upper`.

```
transform_upper(M) # Upper diagonal equivalent system
#>      [,1]      [,2]      [,3]          [,4]      [,5]
#> [1,]    3  2.000000 -1.000000 -4.000000e+00 10.000000
#> [2,]    0 -1.666667  3.333333  3.333333e-01 -7.333333
#> [3,]    0  0.000000  6.000000 -5.200000e+00  7.400000
#> [4,]    0  0.000000  0.000000 -4.440892e-16 14.500000
```

Considering the finite precision with which the above numbers are expressed, it is not difficult to re-write the reduced upper triangular system as follows:

$$\begin{cases} 3x_1 + 2x_2 - x_3 - 4x_4 &= 10 \\ -(5/3)x_2 + (10/3)x_3 + (1/3)x_4 &= -22/3 \\ 6x_3 - (26/5)x_4 &= 37/5 \\ 0 &= 29/2 \end{cases}$$

The last equation is never true, therefore the system has no solution. If, on the other hand, the last equation turned out to be $0 = 0$ then we would have been left with three equations and four unknowns. One of the unknowns could have been chosen as a parameter and the other three unknowns expressed as function of that parameter. That would have meant that the system had an infinite number of solutions.

4.1.3 Exercise 03

The LU decomposition of a square matrix A can be used to solve an algebraic system of linear equations. Starting from the system in matrix form,

$$A\mathbf{x} = \mathbf{b}$$

and using the LU decomposition of A we get:

$$L\,U\,\mathbf{x} = \mathbf{b}$$

If a new set of unknowns, $\mathbf{y}$, defined as

$$\mathbf{y} = U\mathbf{x},$$

is introduced, then the original system becomes a lower triangular system,

$$L\mathbf{y} = \mathbf{b}$$

which can be quickly solved using back-substitution. Eventually, also the solution $\mathbf{x}$ can be found quickly by solving the upper triangular system,

$$U\mathbf{x} = \mathbf{y}$$

as the previously unknown $\mathbf{y}$ have now a numeric value.

Write a program that makes use of the LU decomposition and of the considerations written above, to solve the system presented in Exercise 01. Compare the solution found with the one found when solving Exercise 01.

SOLUTION

The program needs the augmented matrix, $M = (A|\mathbf{b})$, as input. Then it will have to call the function `LUdeco` to find the lower and upper triangular matrices for the decomposition of A. The back substitution can be implemented using the second part of the code in the function `gauss_elim`, as this consisted of the back-substitution part of the algorithm. The same code will have to be implemented twice, the first time to solve $L\mathbf{y} = \mathbf{b}$ and the second time to solve $U\mathbf{x} = \mathbf{y}$. The procedure is included in the following function, called `LUsolve`.

```
LUsolve <- function(M) {
  # Size of the augmented matrix
  tmp <- dim(M)
  n <- tmp[1]

  # Divide the augmented matrix into A and b
  A <- M[,1:n]
  b <- M[,n+1]

  # LU decomposition
  ltmp <- LUdeco(A,"doolittle")

  # L and U matrices
  L <- ltmp$L
  U <- ltmp$U

  # New permuted constants
  # (no permutation in this specific case)
  newb <- b[ltmp$ord]

  # Back-substitution (y)
  y <- rep(0,length=n)
  y[1] <- newb[1]
  for (i in 2:n) {
    y[i] <- newb[i]-sum(y[1:(i-1)]*L[i,1:(i-1)]) # No division
                                                 # by L[i,i]
                                                 # because here
                                                 # all L[i,i]=1
  }

  # Back-substitution (x)
  x <- rep(0,length=n)
  x[n] <- y[n]/U[n,n]
  for (i in (n-1):1) {
```

```
    x[i] <- (y[i]-sum(x[(i+1):n]*U[i,(i+1):n]))/U[i,i]
  }

  return(x)
}

# Apply function to case in Exercise 01
raw_data <- c( 2, 1,-3,-5, 1,-4,
               7,-1, 0, 0,-5, 3,
              -6, 8, 0,-2,-4,-2,
              -3, 8, 1, 8, 6, 0,
               5, 2, 7, 8,-2,-2)
M <- matrix(raw_data,ncol=6,byrow=TRUE)
x1 <- LUsolve(M)
print(x1)
#> [1] -0.3293013 -0.6129265 -1.7584402  1.4130710 -0.9384365

# Comparison with gaussian elimination
x2 <- gauss_elim(M)
print(x2)
#> [1] -0.3293013 -0.6129265 -1.7584402  1.4130710 -0.9384365
```

The function returns the same result as the one found with Gaussian elimination, therefore it is an appropriate function for the task required. There are a couple of observation in place here:

1. The LU decomposition in the code has used the Doolittle method that returned the lower triangular matrix with ones along the diagonal. For this reason, there was no need to divide by `L[i,i]` in the back-substitution section. The function would have equally worked if the Crout method had been used. The only difference is that in that case it is the `U[i,i]` to have ones along the diagonals and this has to be considered for a correct back-substitution.
2. The first back-substitution is, in fact, a forward-substitution because the reduction produces a lower triangular system.

4.1.4 Exercise 04

Computer time to solve a linear system is very short with modern processors. It is normally less than a second when the system size includes 100 or less equations. As it is very tedious and time-consuming to fill matrices of size 100 or more, we can test solution time of the `gauss_elim` function using matrices with elements generated randomly. Generate a random matrix of size $n = 100$, using the sampling of integers between -5 and +5 and using a fixed random seed for generation, in order to compare your results with those of the solution presented here. Use the seed 7821 for matrix A and the seed 7659 for the constant vector, $\mathbf{b}$. Use the R function `Sys.time` to find out the execution time. Try your procedure with various values of n, say $n = 100, 500, 1000$.

SOLUTION

To fill an $n \times n$ square matrix we need n^2 integers (between -5 and +5). We can use the function `sample` with the parameter `replace=TRUE` as the number of integers to be sampled is larger than the number of integers between -5 and 5. Then matrix A is easily created. A similar procedure goes towards the creation of $\mathbf{b}$; here we only need n random integers, and the vector is formed as a $n \times 1$ matrix.

```
# Size of system
n <- 100

# A and b
```

```
set.seed(7821) # To reproduce same numbers
itmp <- sample(-5:5,size=n*n,replace=TRUE)
A <- matrix(itmp,ncol=n)
set.seed(7659) # To reproduce same numbers
b <- matrix(sample(-5:5,size=n,replace=TRUE),ncol=1)
M <- cbind(A,b)

## Start time
st <- Sys.time()

# Gaussian elimination
v <- gauss_elim(M)

## End time
et <- Sys.time()

# Duration
print(et-st)
#> Time difference of 0.02167106 secs

# Repeat with n=500
n <- 500
set.seed(7821)
itmp <- sample(-5:5,size=n*n,replace=TRUE)
A <- matrix(itmp,ncol=n)
set.seed(7659)
b <- matrix(sample(-5:5,size=n,replace=TRUE),ncol=1)
M <- cbind(A,b)
st <- Sys.time()
v <- gauss_elim(M)
et <- Sys.time()
print(et-st)
#> Time difference of 1.477706 secs

# Repeat with n=1000
n <- 1000
set.seed(7821)
itmp <- sample(-5:5,size=n*n,replace=TRUE)
A <- matrix(itmp,ncol=n)
set.seed(7659)
b <- matrix(sample(-5:5,size=n,replace=TRUE),ncol=1)
M <- cbind(A,b)
st <- Sys.time()
v <- gauss_elim(M)
et <- Sys.time()
print(et-st)
#> Time difference of 15.1984 secs
```

Note how the augmented matrix is formed from A and $\mathbf{b}$ using the function `cbind` which creates a matrix out of two or more matrices, columnwise. A similar function to combine matrices rowwise is called `rbind`.

4.1.5 Exercise 05

Write a function that returns coefficients and constants for a tridiagonal system $A\mathbf{x} = \mathbf{b}$. A is a square tridiagonal matrix of size n and $\mathbf{b}$ a vector of length n. The non-zero elements of A and the elements of $\mathbf{b}$ have to be sampled randomly (with repetition) among the numbers,

$$-5, -4, -3, -2, -1, 1, 2, 3, 4, 5$$

Use the integer seed 1243 for the random sampling of matrix A and the integer seed 8731 for the random sampling of vector $\mathbf{b}$.

Using $n = 100, 500, 1000$, compare the execution time to find the solution using the functions `gauss_elim` and `solve_tridiag`. As the last function is supposed to exploit the special structure of a tridiagonal system, execution times for it are expected to be shorter than for `gauss_elim`.

SOLUTION

It is here important to know in advance how many non-zero elements have to be sampled. For matrix A the counting goes as follows. There are n elements along the main diagonal and $n-1$ elements along each adjacent diagonal. Therefore the number of non-zero elements for A is,

$$n + 2(n-1) = 3n - 2$$

The vector $\mathbf{b}$ has simply n elements. Thus a function that provides A and $\mathbf{b}$, given the size n, can be written as in the following code.

```
random_tridiag <- function(n,iseed1=1243,iseed2=8731) {
  # Work out number of non-zero elements
  m <- 3*n-2

  # Random integers between -5 and 5 (excluding 0)
  set.seed(iseed1)
  raw_data <- sample(c(-5:-1,1:5),size=m,replace=TRUE)

  # Fill in matrix
  A <- matrix(rep(0,length=n*n),ncol=n)
  A[1,1] <- raw_data[1]
  A[1,2] <- raw_data[2]
  j <- 2
  for (i in 2:(n-1)) {
    A[i,i-1] <- raw_data[j+1]
    A[i,i] <- raw_data[j+2]
    A[i,i+1] <- raw_data[j+3]
    j <- j+3
  }
  A[n,n-1] <- raw_data[j+1]
  A[n,n] <- raw_data[j+2]

  # Fill in vector of constants
  set.seed(iseed2)
  b <- matrix(sample(c(-5:-1,1:5),size=n,replace=TRUE),ncol=1)

  return(list(A=A,b=b))
}
```

We can next use `random_tridiag` to generate A and $\mathbf{b}$, form the augmented matrix M and find out the time to find the solution of the related tridiagonal system using `gauss_elim` and `solve_tridiag`. We can use the three suggested values for n, 100, 500 and 1000.

```r
# n = 100
ltmp <- random_tridiag(n)
M <- cbind(ltmp$A,ltmp$b)
st <- Sys.time()
x1 <- gauss_elim(M)
et <- Sys.time()
print(paste("Time for Gaussian elimination: ",et-st))
#> [1] "Time for Gaussian elimination:  0.174921035766602"
st <- Sys.time()
x2 <- solve_tridiag(M)
et <- Sys.time()
print(paste("Time for Thomas algorithm:     ",et-st))
#> [1] "Time for Thomas algorithm:      0.00685501098632812"

# The two numerical solutions are very close
print(sum(abs(x1-x2)))
#> [1] 1.224499e-11

# n = 500
ltmp <- random_tridiag(n)
M <- cbind(ltmp$A,ltmp$b)
st <- Sys.time()
x1 <- gauss_elim(M)
et <- Sys.time()
print(paste("Time for Gaussian elimination: ",et-st))
#> [1] "Time for Gaussian elimination:  0.250078916549683"
st <- Sys.time()
x2 <- solve_tridiag(M)
et <- Sys.time()
print(paste("Time for Thomas algorithm:     ",et-st))
#> [1] "Time for Thomas algorithm:      0.0173511505126953"

# The two numerical solutions are very close
print(sum(abs(x1-x2)))
#> [1] 1.224499e-11

# n = 1000
ltmp <- random_tridiag(n)
M <- cbind(ltmp$A,ltmp$b)
st <- Sys.time()
x1 <- gauss_elim(M)
et <- Sys.time()
print(paste("Time for Gaussian elimination: ",et-st))
#> [1] "Time for Gaussian elimination:  0.299400091171265"
st <- Sys.time()
x2 <- solve_tridiag(M)
et <- Sys.time()
print(paste("Time for Thomas algorithm:     ",et-st))
#> [1] "Time for Thomas algorithm:      0.0160989761352539"

# The two numerical solutions are very close
print(sum(abs(x1-x2)))
#> [1] 1.224499e-11
```

Clearly the solution is found much faster using Thomas' algorithm. The reason is that this algorithm involves a much smaller number of numerical operations (additions/subtractions and multiplications/divisions). Whenever the system is known to be tridiagonal, a fast algorithm like Thomas' algorithm should be used to find the solution.

4.2 Exercises on matrix decomposition

4.2.1 Exercise 06

Oneway to create arbitrary symmetric matrices is to add a generic matrix to its transpose. Prove that the resulting matrix is, indeed, symmetric and create a function called `symmat` that take the matrix size, n, a set of numbers (not necessarily n) as input and returns a symmetric $n \times n$ matrix, with elements derived from the input numbers.

SOLUTION

Consider the matrix M resulting from the sum of a generic matrix A and its transpose:

$$M = A + A^T$$

It is easy to show that M is symmetric because the transpose of a transpose is the matrix itself:

$$M^T = (A + A^T)^T = A^T + (A^T)^T = A^T + A = A + A^T = M \;\Rightarrow\; M^T = M$$

The R function requested, `symmat`, is relatively straightforward to implement. First a random set of n^2 integers is selected (with the possibility of sampling the same integer more than once) using `sample`. Then the numbers obtained are arranged, column by column, in a generic matrix A. Finally, a symmetric matrix, M, is obtained as sum of A and its transpose.

```
symmat <- function(rrr,n) {
  # rrr is a vector of numbers from which to sample
  #   n is the generic square matrix size

  # Elements to fill generic matrix A
  rtmp <- sample(rrr,size=n*n,replace=TRUE)
  A <- matrix(rtmp,ncol=n)

  # Symmetric matrix
  M <- A+t(A)

  return(M)
}
```

The function just created can now be tested. As no specific size and or type of input numbers were specified, the results shown here are in general going to be different, depending on what input was provided to the function.

```
# First set of numbers (just two values!)
rrr <- c(0,1)

# Test function. Matrix size is n=4
M <- symmat(rrr,4)
print(M)
#>      [,1] [,2] [,3] [,4]
#> [1,]    0    0    1    2
#> [2,]    0    0    0    2
#> [3,]    1    0    2    1
#> [4,]    2    2    1    2
```

```
# Second set of numbers (real between 0 and 1)
rrr <- seq(0,1,length.out=11)

# Test function. Matrix size is n=5
M <- symmat(rrr,5)
print(M)
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]  1.8  0.4  1.8  0.8  1.6
#> [2,]  0.4  1.6  1.4  0.4  0.1
#> [3,]  1.8  1.4  1.0  1.2  0.6
#> [4,]  0.8  0.4  1.2  1.4  0.9
#> [5,]  1.6  0.1  0.6  0.9  0.8
```

The matrices obtained are indeed symmetric. The function created works as expected.

4.2.2 Exercise 07

Apply the Cholesky decomposition to the following symmetric matrix:

$$A = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 2 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Calculate the determinant of A using the result of the decomposition and verify that the result is correct with both `condet` and `det`.

SOLUTION

The matrix is symmetric, but we have to see whether it is positive definite. The leading principal minors are, proceding along the diagonal of A,

$$|2| = 2 \quad , \quad \begin{vmatrix} 2 & 1 \\ 1 & 2 \end{vmatrix} = 3 \quad , \quad \begin{vmatrix} 2 & 1 & 0 \\ 1 & 2 & 0 \\ 0 & 1 & 1 \end{vmatrix} = 3$$

This can also be worked out using R.

```
# Matrix A
A <- matrix(c(2,1,0,1,2,1,0,0,1),ncol=3)
print(A)
#>      [,1] [,2] [,3]
#> [1,]    2    1    0
#> [2,]    1    2    0
#> [3,]    0    1    1

# Minors
for (i in 1:3) {
  mm <- matrix(A[1:i,1:i],ncol=i)
  print(mm)
  print(det(mm))
}
#>      [,1]
#> [1,]    2
#> [1] 2
#>      [,1] [,2]
#> [1,]    2    1
```

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

In any case, all the minors of A are positive and therefore A is positive definite. Being also symmetric, we can apply Cholesky decomposition and find the lower triangular matrix L and its transpose upper triangular, L^T. Using `chol` one has to remember that the function returns the upper triangular, rather than the lower triangular.

```
# Cholesky decomposition
L <- chol(A)
print(L) # Upper triangular
#>          [,1]      [,2] [,3]
#> [1,] 1.414214 0.7071068    0
#> [2,] 0.000000 1.2247449    0
#> [3,] 0.000000 0.0000000    1
print(t(L)) # Lower triangular
#>           [,1]     [,2] [,3]
#> [1,] 1.4142136 0.000000    0
#> [2,] 0.7071068 1.224745    0
#> [3,] 0.0000000 0.000000    1

# Their product yields A
print(t(L) %*% L)
#>      [,1] [,2] [,3]
#> [1,]    2    1    0
#> [2,]    1    2    0
#> [3,]    0    0    1
```

The determinant of A can be calculated easily using the decomposition because:

1. the determinant of the product of two matrices is the product of their determinants;
2. the determinant of a triangular matrix is the product of the elements of its diagonal;
3. the determinant of a matrix transpose is equal to the determinant of the matrix.

Therefore we can compute the determinant of A using the L derived from its Cholesky decomposition, multiply the elements of its diagonal and squaring the result.

```
# Product of the elements along the diagonal of L
detA <- prod(diag(L))

# Determinant of A
detA <- detA*detA
print(detA)
#> [1] 3

# Compare with the determinant computed using condet and det
print(condet(A))
#> [1] 3
print(det(A))
#> [1] 3
```

The three results coincide, as expected.

4.2.3 Exercise 08

Apply the Cholesky decomposition to the following symmetric matrix:

$$A = \begin{pmatrix} 2 & 1 & -3 \\ 1 & 2 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Does the `chol` function return a numeric result? Why?

SOLUTION

Let us work out, first of all, if the symmetric matrix is positive definite, using R.

```
# Matrix A
A <- matrix(c(2,1,0,1,2,1,-3,0,1),ncol=3)
print(A)
#>      [,1] [,2] [,3]
#> [1,]    2    1   -3
#> [2,]    1    2    0
#> [3,]    0    1    1

# Minors
for (i in 1:3) {
  mm <- matrix(A[1:i,1:i],ncol=i)
  print(mm)
  print(det(mm))
}
#>      [,1]
#> [1,]    2
#> [1] 2
#>      [,1] [,2]
#> [1,]    2    1
#> [2,]    1    2
#> [1] 3
#>      [,1] [,2] [,3]
#> [1,]    2    1   -3
#> [2,]    1    2    0
#> [3,]    0    1    1
#> [1] 0
```

Not all the leading principal minors are positive. We should expect something unusual happening when the function `chol` is applied to A.

```
# Is Cholesky valid?
L <- chol(A)
#> Error in chol.default(A): the leading minor of order 3 is not positive
```

It is indeed true that the leading principal minor of order 3 is not a positive number, as it is zero.

4.2.4 Exercise 09

The QR decomposition is at the foundation of most methods to find the eigenvalues of a matrix. The algorithm is a cyclical repetition of the following steps, provided that the matrix whose eigenvalues have to be found is the square matrix of size n, A:

1. Find the QR decomposition of A. This is cycle 1 of the algorithm, and A is called A_0. The QR decomposition yields,
$$A_0 = Q_0\, R_0$$

2. Build the new matrix, A_1, for cycle 1,
$$A_1 = R_0\, Q_0$$

3. Check that all the elements in the lower triangular part of A_1 are close to 0 (practically below a fixed threshold `zero_cut`). If they are all close to zero the algorithm has terminated successfully and the eigenvalues of A are the elements on the diagonal of A_1. Otherwise, go back to step 1, where in cycle 2 A_1 replaces A_0. In general, in cycle i, A_{i-1} replaces A_{i-2}.

4. If, after a pre-established number of cycles, `nmax`, not all elements of the lower triangular part of A_i, at cycle i, fall below the threshold `zero_cut`, the algorithm has not achieved convergence and it is not possible to find the eigenvalues.

Write a function, called `eigenQR`, that implements the steps above to find the eigenvalues of an input matrix, A. Besides the matrix, the function takes in the threshold, `zero_cut`, basically a small number (default is `1e-6`) and the maximum number of cycles, `nmax` (default is 1000). Apply the function to find the eigenvalues of,
$$A = \begin{pmatrix} -2 & -4 & 2 \\ -2 & 1 & 2 \\ 4 & 2 & 5 \end{pmatrix}$$

SOLUTION

A possible implementation of the function `eigenQR` is presented here.

```
eigenQR <- function(A,zero_cut=1e-6,nmax=1000) {
  # Size of A
  n <- dim(A)[1]

  # Loop
  izero <- 100
  icycle <- 1
  for (icycle in 1:nmax) {
    QR <- qr(A)
    Q <- qr.Q(QR)
    R <- qr.R(QR)
    A <- R %*% Q
    eigs <- diag(A)
    s <- 0
    for (i in 2:n) {
      s <- s+sum(abs(A[i,1:(i-1)]))
    }
    if (s < izero) izero <- s
    if (izero < zero_cut) {
      return(list(eigs=eigs,res=s))
    }
  }

  # Algorithm has not converged
  cat("The algorithm has not converged")

  return(list(eigs=diag(A),res=s))
}
```

The function can be written in any other way, as long as it delivers what asked. In fact, let us apply it to the matrix provided.

```
# Matrix provided
A <- matrix(c(-2,-2,4,-4,1,2,2,2,5),ncol=3)
print(A)
#>      [,1] [,2] [,3]
#> [1,]   -2   -4    2
#> [2,]   -2    1    2
#> [3,]    4    2    5

# Apply function to matrix
lEigens <- eigenQR(A)

# Eigenvalues
print(lEigens$eigs)
#> [1]  6 -5  3

# What is the sum of the absolute values of all elements
# in the lower triangular part of the final A(n) matrix?
print(lEigens$res)
#> [1] 8.423575e-07

# Check with built in function
lambdas <- eigen(A)
print(lambdas$values)
#> [1]  6 -5  3
```

In this case, the algorithm suggested works.

4.2.5 Exercise 10

What would be the result of applying `eigenQR` to the following matrix,

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}?$$

Try to understand what the issue is in this case, by tracing it back, through inspection of the function, to what is causing it. Verify that the eigenvalues are found with the default R function, `eigen`.

SOLUTION

Let us apply `eigenQR` to the matrix provided, without worrying to much of what the result can be.

```
# Matrix provided
A <- matrix(c(0,1,1,0),ncol=2)
print(A)
#>      [,1] [,2]
#> [1,]    0    1
#> [2,]    1    0

# Attempt at obtaining eigenvalues
lEigens <- eigenQR(A)
#> The algorithm has not converged
print(lEigens$eigs)
#> [1] 0 0
print(lEigens$res)
#> [1] 1
```

The algorithm has clearly not converged. The residual is relatively large (`res = 1`). The default number of cycles is 1000; was the residual larger, say at cycle 10, or 5, or 2? We can check that by changing the `nmax` parameter.

```
# eigenQR again, with nmax=10,5,2
lEigens <- eigenQR(A,nmax=10)
#> The algorithm has not converged
print(lEigens$eigs)
#> [1] 0 0
print(lEigens$res)
#> [1] 1
```

So, the residual is 1 already at the first few cycles. It seems that the algorithm gets *trapped* immediately in some sort of fixed matrix configuration. To explore this further, we will need to implement the algorithm one step at a time. It is simply a matter of carrying out the QR decompositionn manually and build the new matrix $R\,Q$, to see what that entails.

```
# First QR decomposition
QR <- qr(A)
Q <- qr.Q(QR)
print(Q)
#>      [,1] [,2]
#> [1,]    0   -1
#> [2,]   -1    0
R <- qr.R(QR)
print(R)
#>      [,1] [,2]
#> [1,]   -1    0
#> [2,]    0   -1

# New matrix
Anew <- R %*% Q
print(Anew)
#>      [,1] [,2]
#> [1,]    0    1
#> [2,]    1    0
```

With the above simple steps we have revealed what went wrong with the matrix A provided, because the new matrix is exactly equal to A. As the lower-triangular only element of the new matrix is 1, so is the residual which is, therefore, going to be always 1. Essentially, the algorithm gets trapped in a never-ending loop where the residual does not get smaller beyond a given value (1 in this case). This is the risk associated with using the algorithm above. The algorithm is, in fact, known as the *unshifted QR algorithm* because a modification of its steps that, among other things, shifts the elements of the matrix of a given quantity, either speeds up its convergence rate, or avoids situations in which convergence is never achieved. Improvements of the QR algorithm have been investigated and implemented and modern functions in general avoid the type of problems met with ths specific example. The eigenvalues are for instance readily found with `eigen`.

```
lEigens <- eigen(A)
print(lEigens)
#> eigen() decomposition
#> $values
#> [1]  1 -1
#>
#> $vectors
#>            [,1]       [,2]
#> [1,] 0.7071068 -0.7071068
```

```
#> [2,] 0.7071068  0.7071068
```

4.2.6 Exercise 11

The requirement for the algorithm of Exercise 09 did not included the eigenvectors as part of the output. In fact, obtaining the eigenvectors in general implies using additional algorithmical steps that use matrices in special forms. Rather than considering the general case, we will study the special case in which the starting matrix is symmetric. As explained in the theory of the QR algorithm, the similarity transformation between A and the A_k matrix obtained at cycle k of the algorithm is of the form,

$$A = (Q_0\, Q_1\, \cdots\, Q_k)\, A_k\, (Q_0\, Q_1\, \cdots\, Q_k)^T$$

The quantities in parenthesis can be indicated simply with the letter Q, as they are single matrices. The expression of the algorithm after k iteration is thus

$$A = QA_kQ^T$$

In general, A_k is an upper triangular matrix and the columns of Q are not the eigenvectors of A. The only thing that can be stated with accuracy is that the elements on the diagonal of A_k are good approximations of A's eigenvalues. If A is symmetric, though, A_k is a diagonal matrix and the columns of Q are the orthonormal eigenvectors of A. To see this let's consider that for a symmetric matrix $A^T = A$. Therefore, taking the transpose of the expression above we get,

$$A^T = QA_k^TQ^T = A = QA_kQ^T \quad \Rightarrow \quad A_k^T = A_k$$

But A_k is an upper triangular matrix and this can be at the same time symmetric only if the off-diagonal elements are all zero. In conclusion, A_k is a diagonal matrix containing the eigenvalues of A. And, accordingly, the columns of Q will be its ordered eigenvectors.

Modify the algorithm created for Exercise 09 so that the matrix of eigenvectors forms part of its output, and apply it to the symmetric matrix, $B = A + A^T$, where A was introduced in that exercise. Find an effective way to show that the eigenvectors found do correspond to the eigenvalues obtained.

SOLUTION

The modified algorithm just have to keep track of each matrix Q of the QR decomposition in order to form the product,

$$Q_0\, Q_1\, \cdots\, Q_k$$

Then an initial matrix that starts off the multiplication chain, $Q_0\, Q_1\, \cdots\, Q_k$, is needed. Subsequent matrices will build up in the loop through matrix multiplication. Therefore the initial matrix will have to be the identity matrix so that the first matrix multiplication will yield Q_0 (indeed it's $Q_0\, I = Q_0$). The algorithm is here illustrated in a modified `eigenQR` function called `eigenQR2`.

```
eigenQR2 <- function(A,zero_cut=1e-6,nmax=1000) {
  # Size of A
  n <- dim(A)[1]

  # Eventually, V will be the matrix with eigenvectors.
  # Initially, V is the identity matrix
  V <- diag(n)

  # Loop
  izero <- 100
  icycle <- 1
  for (icycle in 1:nmax) {
    QR <- qr(A)
```

```
    Q <- qr.Q(QR)
    R <- qr.R(QR)
    V <- V %*% Q
    A <- R %*% Q
    eigs <- diag(A)
    s <- 0
    for (i in 2:n) {
      s <- s+sum(abs(A[i,1:(i-1)]))
    }
    if (s < izero) izero <- s
    if (izero < zero_cut) {
      return(list(eigs=eigs,V=V,A=A,res=s))
    }
  }

  # Algorithm has not converged
  cat("The algorithm has not converged")

  return(list(eigs=diag(A),V=V,A=A,res=s))
}
```

Let us apply this new function to the suggested matrix.

```
# Suggested matrix
A <- matrix(c(-2,-2,4,-4,1,2,2,2,5),ncol=3) # Old matrix
B <- A + t(A) # Symmetric matrix

# Apply the new function and find eigenvalues and eigenvectors
lEigens <- eigenQR2(B)

# Eigenvalues
lbda <- lEigens$eigs

# Matrix whose columns are the eigenvectors
V <- lEigens$V

# Display values
print(lbda)
#> [1]  12.570240 -10.270905   5.700665
print(V)
#>            [,1]       [,2]        [,3]
#> [1,] -0.2667871  0.7987391  0.53929624
#> [2,] -0.2049283  0.4997661 -0.84156892
#> [3,] -0.9417160 -0.3350368  0.03035313
```

We can verify that the obtained scalars `lbda` and vectors `V` are eigenvalues and eigenvectors through the definition,

$$B\mathbf{v} = \lambda\mathbf{v},$$

where the eigenvector $\mathbf{v}$ corresponding to the eigenvalue λ is a 3×1 matrix. Considering that in R the division of two vectors is done in a pointwise fashion, we should have,

$$B\mathbf{v}/\mathbf{v} = \begin{pmatrix} \lambda \\ \lambda \\ \lambda \end{pmatrix}.$$

This can be applied to the three columns at the same time if the single eigenvector $\mathbf{v}$ is replaced by the matrix V. In this case we should have,

$$BV/V = \begin{pmatrix} \lambda_1 & \lambda_2 & \lambda_3 \\ \lambda_1 & \lambda_2 & \lambda_3 \\ \lambda_1 & \lambda_2 & \lambda_3 \end{pmatrix}$$

And indeed,

```
# Result from the application of QR
print(B %*% V / V)
#>          [,1]      [,2]     [,3]
#> [1,] 12.57024 -10.27091 5.700665
#> [2,] 12.57024 -10.27091 5.700665
#> [3,] 12.57024 -10.27090 5.700665

# Eigenvalues
print(lbda)
#> [1]  12.570240 -10.270905   5.700665

# Eigenvalues with a different method
print(eigen(B))
#> eigen() decomposition
#> $values
#> [1]  12.570240   5.700665 -10.270905
#>
#> $vectors
#>           [,1]        [,2]       [,3]
#> [1,] 0.2667871  0.53929624  0.7987391
#> [2,] 0.2049283 -0.84156892  0.4997661
#> [3,] 0.9417160  0.03035313 -0.3350368
```

It is also possible to verify that the three column vectors are orthonormal, using the inner product.

```
for (i in 1:3) {
  for (j in 1:3) {
    print(sum(V[,i]*V[,j]))
  }
}
#> [1] 1
#> [1] 5.967449e-16
#> [1] 6.245005e-17
#> [1] 5.967449e-16
#> [1] 1
#> [1] 1.752071e-16
#> [1] 6.245005e-17
#> [1] 1.752071e-16
#> [1] 1
```

Equivalently, the property of an orthogonal matrix can be exploited with a single matrix product.

```
# This should be a 3X3 identity matrix
print(V %*% t(V))
#>               [,1]          [,2]          [,3]
#> [1,]  1.000000e+00 -1.110223e-16 -3.504141e-16
#> [2,] -1.110223e-16  1.000000e+00 -9.714451e-17
#> [3,] -3.504141e-16 -9.714451e-17  1.000000e+00
```

4.2.7 Exercise 12

A square matrix A can be *diagonalised* if a diagonal matrix D and another, invertible, square matrix P can be found such that the following relation holds:

$$A = PDP^{-1} \tag{1}$$

When this happens, D has the eigenvalues of A along its diagonal, while the columns of P are, in order, the corresponding eigenvectors. This can be seen if the above equation is multiplied, on the left, by P,

$$AP = PD,$$

and if P is re-written in terms of its column vectors, $\mathbf{v}_i$,

$$A(\mathbf{v}_1 \ \dots \ \mathbf{v}_n) = (\mathbf{v}_1 \ \dots \ \mathbf{v}_n) \begin{pmatrix} \lambda_1 & \dots & 0 \\ \dots & \dots & \dots \\ 0 & \dots & \lambda_n \end{pmatrix}$$

$$\Downarrow$$

$$(A\mathbf{v}_1 \ \dots \ A\mathbf{v}_n) = (\lambda_1\mathbf{v}_1 \ \dots \ \lambda_n\mathbf{v}_n)$$

Clearly, the expression just derived is a set of n eigenvalue equations for n eigenvectors of A.

Using the function `eigen`, write the diagonal matrix D corresponding to the 4×4 matrix A generated by sampling randomly the integers $-1, 0, 1$ (use seed 1188). Find also the expression of P and, using matrix multiplication in R, show that $AP = PD = DP$.

SOLUTION

The matrix is generated using `sample`.

```
# Matrix A
set.seed(1188)
A <- matrix(sample(-1:1,size=16,replace=TRUE),ncol=4)
print(A)
#>      [,1] [,2] [,3] [,4]
#> [1,]    1   -1   -1    0
#> [2,]    0    1    0    0
#> [3,]    1    1   -1   -1
#> [4,]   -1    0    1    0
```

The eigenvalues and eigenvectors can be found using `eigen`. The printout of these objects can be difficult to read, as in general the result will include complex numbers (characterised by the presence of the letter `i`, representing the imaginary unit).

```
# Eigenvalues and eigenvectors in the named list lEigen
lEigen <- eigen(A)

# Eigenvalues
lbda <- lEigen$values
print(lbda)
#> [1]  2.220446e-16+1i  2.220446e-16-1i  1.000000e+00+0i -3.045239e-16+0i

# Corresponding eigenvectors
P <- lEigen$vectors
print(P)
#>                       [,1]                  [,2]          [,3]             [,4]
#> [1,]  0.3535534+0.3535534i  0.3535534-0.3535534i -0.7559289+0i  7.071068e-01+0i
#> [2,]  0.0000000+0.0000000i  0.0000000+0.0000000i  0.3779645+0i  0.000000e+00+0i
```

```
#> [3,]  0.7071068+0.0000000i  0.7071068+0.0000000i -0.3779645+0i  7.071068e-01+0i
#> [4,] -0.3535534-0.3535534i -0.3535534+0.3535534i  0.3779645+0i -2.220446e-16+0i
```

The four eigenvalues are,

$$\lambda_1 = \mathrm{i} \quad , \quad \lambda_2 = -\mathrm{i} \quad , \quad \lambda_3 = 1 \quad , \quad \lambda_4 = 0$$

To verify that thesecorrespond to the columns of P we should form both expressions,

$$A\mathbf{v}_i \quad \text{and} \quad \lambda_i \mathbf{v}_i$$

and verify that they coincide. The function `cbind` can be used for a better visual comparison.

```
# Verify eigen-equations for all eigenvalues
for (i in 1:4) {
  cat(paste("Eigenvalue",i,"\n"))
  print(cbind(A %*% P[,i],lbda[i]*P[,i]))
  cat("\n")
}
#> Eigenvalue 1
#>                            [,1]                       [,2]
#> [1,] -3.535534e-01+0.3535534i -3.535534e-01+0.3535534i
#> [2,]  0.000000e+00+0.0000000i  0.000000e+00+0.0000000i
#> [3,]  4.440892e-16+0.7071068i  1.570092e-16+0.7071068i
#> [4,]  3.535534e-01-0.3535534i  3.535534e-01-0.3535534i
#>
#> Eigenvalue 2
#>                            [,1]                       [,2]
#> [1,] -3.535534e-01-0.3535534i -3.535534e-01-0.3535534i
#> [2,]  0.000000e+00+0.0000000i  0.000000e+00+0.0000000i
#> [3,]  4.440892e-16-0.7071068i  1.570092e-16-0.7071068i
#> [4,]  3.535534e-01+0.3535534i  3.535534e-01+0.3535534i
#>
#> Eigenvalue 3
#>                [,1]          [,2]
#> [1,] -0.7559289+0i -0.7559289+0i
#> [2,]  0.3779645+0i  0.3779645+0i
#> [3,] -0.3779645+0i -0.3779645+0i
#> [4,]  0.3779645+0i  0.3779645+0i
#>
#> Eigenvalue 4
#>                   [,1]             [,2]
#> [1,] -1.110223e-16+0i -2.153309e-16+0i
#> [2,]  0.000000e+00+0i  0.000000e+00+0i
#> [3,]  1.110223e-16+0i -2.153309e-16+0i
#> [4,]  1.110223e-16+0i  6.761790e-32+0i

# In relation to the last eigenvalue, consider that
# the corresponding eigenvector is not a null vector
print(P[,4])
#> [1]  7.071068e-01+0i  0.000000e+00+0i  7.071068e-01+0i -2.220446e-16+0i
print(sum(Conj(P[,4])*P[,4]))
#> [1] 1+0i
```

The previous set of eigen-equations can be verified just with one matrix operation, once the diagonal matrix D of eigenvalues is formed.

```
# All eigen-equations true when AP=PD (=DP)
D <- diag(lbda,nrow=4,ncol=4)
print(D)
#>                  [,1]             [,2] [,3]             [,4]
#> [1,] 2.220446e-16+1i 0.000000e+00+0i 0+0i  0.000000e+00+0i
#> [2,] 0.000000e+00+0i 2.220446e-16-1i 0+0i  0.000000e+00+0i
#> [3,] 0.000000e+00+0i 0.000000e+00+0i 1+0i  0.000000e+00+0i
#> [4,] 0.000000e+00+0i 0.000000e+00+0i 0+0i -3.045239e-16+0i
print(cbind(A %*% P,P %*% D))
#>                           [,1]                        [,2]          [,3]
#> [1,] -3.535534e-01+0.3535534i -3.535534e-01-0.3535534i -0.7559289+0i
#> [2,]  0.000000e+00+0.0000000i  0.000000e+00+0.0000000i  0.3779645+0i
#> [3,]  4.440892e-16+0.7071068i  4.440892e-16-0.7071068i -0.3779645+0i
#> [4,]  3.535534e-01-0.3535534i  3.535534e-01+0.3535534i  0.3779645+0i
#>                   [,4]                        [,5]                        [,6]
#> [1,] -1.110223e-16+0i -3.535534e-01+0.3535534i -3.535534e-01-0.3535534i
#> [2,]  0.000000e+00+0i  0.000000e+00+0.0000000i  0.000000e+00+0.0000000i
#> [3,]  1.110223e-16+0i  1.570092e-16+0.7071068i  1.570092e-16-0.7071068i
#> [4,]  1.110223e-16+0i  3.535534e-01-0.3535534i  3.535534e-01+0.3535534i
#>                [,7]             [,8]
#> [1,] -0.7559289+0i -2.153309e-16+0i
#> [2,]  0.3779645+0i  0.000000e+00+0i
#> [3,] -0.3779645+0i -2.153309e-16+0i
#> [4,]  0.3779645+0i  6.761790e-32+0i
```

4.2.8 Exercise 13

Consider the full set of *Pauli matrices*,

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad , \quad \sigma_y = \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix} \quad , \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Using the function `eigen`, verify that the eigenvalues of each matrix are $+1$ and -1 and that the corresponding eigenvectors are,

$$\psi_{x+} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad , \quad \psi_{x-} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

$$\psi_{y+} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ \mathrm{i} \end{pmatrix} \quad , \quad \psi_{y-} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -\mathrm{i} \end{pmatrix}$$

$$\psi_{z+} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad , \quad \psi_{z-} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

SOLUTION

First we have to define the three Pauli matrices. The only difficulty with that could be the definition in R of a complex number. Using the syntax, $a + bi$, where a and b are numeric variables, any complex number can be defined.

```
# Pauli sigma_x
sigx <- matrix(c(0,1,1,0),ncol=2)

# Pauli sigma_y
sigy <- matrix(c(0,0+1i,0-1i,0),ncol=2)

# Pauli sigma_z
```

```
sigz <- matrix(c(1,0,0,-1),ncol=2)

print(sigx)
#>      [,1] [,2]
#> [1,]    0    1
#> [2,]    1    0
print(sigy)
#>      [,1] [,2]
#> [1,] 0+0i 0-1i
#> [2,] 0+1i 0+0i
print(sigz)
#>      [,1] [,2]
#> [1,]    1    0
#> [2,]    0   -1
```

Then the calculation of the eigenvectors and eigenvalues is readily done using `eigen`.

```
# Eigenvalues and eigenvectors of sigma_x
print(eigen(sigx))
#> eigen() decomposition
#> $values
#> [1]  1 -1
#>
#> $vectors
#>            [,1]       [,2]
#> [1,] 0.7071068 -0.7071068
#> [2,] 0.7071068  0.7071068

# Eigenvalues and eigenvectors of sigma_y
print(eigen(sigy))
#> eigen() decomposition
#> $values
#> [1]  1 -1
#>
#> $vectors
#>                         [,1]                   [,2]
#> [1,] -0.7071068+0.0000000i -0.7071068+0.0000000i
#> [2,]  0.0000000-0.7071068i  0.0000000+0.7071068i

# Eigenvalues and eigenvectors of sigma_z
print(eigen(sigz))
#> eigen() decomposition
#> $values
#> [1]  1 -1
#>
#> $vectors
#>      [,1] [,2]
#> [1,]   -1    0
#> [2,]    0   -1
```

The eigenvalues are ± 1, as expected, but the eigenvectors are not exactly what was defined in the problem. The signs of some components can differ. But this is coherent with the scaling arbitrariety in the definition of eigenvectors. The only important feature of the values found is that the eigenvectors of each matrix are

orthonormal. In terms of the diagonalising matrix P, this means that P is a unitary matrix:

$$P^\dagger P = PP^\dagger = I$$

```
# The matrices formed by the eigenvectors
# of each Pauli matrix, are unitary
ltmp <- eigen(sigx)
Px <- ltmp$vectors
print(t(Px) %*% Conj(Px))
#>      [,1] [,2]
#> [1,]    1    0
#> [2,]    0    1
ltmp <- eigen(sigy)
Py <- ltmp$vectors
print(t(Py) %*% Conj(Py))
#>      [,1] [,2]
#> [1,] 1+0i 0+0i
#> [2,] 0+0i 1+0i
ltmp <- eigen(sigz)
Pz <- ltmp$vectors
print(t(Pz) %*% Conj(Pz))
#>      [,1] [,2]
#> [1,]    1    0
#> [2,]    0    1
```

4.2.9 Exercise 14

After having generated a 12×10 random matrix M with the function `sample` and starting from the set $\{-2, -1, 0, 1, 2\}$ and with seed 2673, find its singular values without using the function `svd`. Verified then that the values found are correct by using the function `svd`.

SOLUTION

In Appendix B it is explained that the singular values can be found as square roots of the eigenvalues of the matrix $M_u = MM^\dagger$ or the matrix $M_v = M^\dagger M$. Mu is a 12×12 symmetric (Hermitian) matrix and has 12 eigenvalues, the two smallest being zero. Mv is a 10×10 symmetric matrix with 10 eigenvalues, exactly equal to the non-zero eigenvalues of Mu.

Let's first generate the random matrix M.

```
# Fixed starting seed
set.seed(2673)

# Pool for sampling
x <- -2:2

# Random 12 X 10 matrix
M <- matrix(sample(x,size=120,replace=TRUE),ncol=10)
print(dim(M))
#> [1] 12 10
```

Then let us find the eigenvalues of both Mu and Mv. The square root of the 10 largest ones are the requested singular values. Let us save them in a variable for later comparison.

```
# Mu: 12X12 symmetric matrix
Mu <- M %*% t(M)
```

```
# The singular values are square root of the
# first 10 (non-zero) eigenvalues (the last two
# are approximations to zero and, as such, can be also
# negative numbers)
lEigen <- eigen(Mu)
print(sqrt(lEigen$values[1:10]))
#>  [1] 8.098100 6.933402 5.906714 5.467179 5.177016 4.587080 3.964595 2.951754
#>  [9] 2.171782 1.256623

# Mv: 10X10 symmetric matrix
Mv <- t(M) %*% M

# The singular values are square root of the
# 10 eigenvalues. No zeros involved here
lEigen <- eigen(Mv)
sigmas <- sqrt(lEigen$values) # Store for later comparison
print(sigmas)
#>  [1] 8.098100 6.933402 5.906714 5.467179 5.177016 4.587080 3.964595 2.951754
#>  [9] 2.171782 1.256623
```

Finally, let us calculate the singular values using `svd`.

```
# normal SVD of the original matrix
lSVD <- svd(M,nu=12,nv=10)
print(lSVD$d)
#>  [1] 8.098100 6.933402 5.906714 5.467179 5.177016 4.587080 3.964595 2.951754
#>  [9] 2.171782 1.256623
print(sigmas) # Compare with previous result
#>  [1] 8.098100 6.933402 5.906714 5.467179 5.177016 4.587080 3.964595 2.951754
#>  [9] 2.171782 1.256623
```

The two sets of values, calculated with different functions, do coincide.

4.2.10 Exercise 15

Create a 20×20 real random matrix, A, using numbers generated from a normal distribution with mean 2 and standard deviation 0.5, and making sure to start the random generation with integer seed 3367 (fill the matrix column by column). Next, create a 20×1 real column vector, $\mathbf{b}$, filled with a random sample with repetitions from the set $\{-1, 1\}$, and making sure to start the sampling with integer seed 4189.

(a) Attempt to solve the linear system $A\mathbf{x} = \mathbf{b}$ using the functions `PJacobi` and `GSeidel` in the `comphy` package. Notice that the matrix A is not diagonally dominant.

(b) Turn A into a diagonally dominant matrix by adding a same positive integer to all the elements along its diagonal. What is the smallest positive integer that makes A diagonally dominant?

(c) With the diagonally dominant matrix found in part b, find the numerical solution of the system, using both `PJacobi` and `GSeidel`.

SOLUTION

The matrix and the column vector are generated with the following code.

```
# Fixed starting seed for A
set.seed(3367)
A <- matrix(rnorm(400,mean=2,sd=0.5),ncol=20)

# Fixed starting seed for b
```

```
set.seed(4189)
b <- matrix(sample(c(-1,1),size=20,replace=TRUE),ncol=1)
```

a As the matrix of the system is not diagonally dominant, convergence might not be obtained with either the Jacobi or the Gauss-Seidel method. In any case we will have to force both functions to attempt the solution.

```
# Initially both functions do not proceed
# as the matrix is not diagonally dominant
xP <- PJacobi(A,b)
#> The matrix of coefficients is not diagonally dominant.
#> Not attempting solution.
xG <- GSeidel(A,b)
#> The matrix of coefficients is not diagonally dominant.
#> Not attempting solution.

#
# Then the keyword 'ddominant' is switched to
# 'FALSE' so to make the functions to attempt
# and find a solution
xP <- PJacobi(A,b,ddominant=FALSE)
#> The matrix of coefficients is not diagonally dominant.
#> Attempting solution anyway...
#>
#> The increment is getting larger and larger.
#> Max value of increment at cycle 90: 1.177459338799921e+100
xG <- GSeidel(A,b,ddominant=FALSE)
#> The matrix of coefficients is not diagonally dominant.
#> Attempting solution anyway...
#>
#> The increment is getting larger and larger.
#> Max value of Delta_X at cycle 463: 1.487227920382l8e+100
```

b The matrix A is now changed with all elements on the diagonal being progressively increased by one unit, until A becomes diagonally dominant. A function to determine whether a matrix is diagonally dominant must be assembled first. A possibility to do this quickly is to copy, paste and modify the code used in `PJacobi`. The result is here presented in the following code.

```
check_dominant <- function(a) {
 # To check that A is diagonally dominant

 # Size of matrix
 tmp <- dim(A)
 n <- tmp[1]

 # Pivoting if not correct order
 # Swap rows to have largest values on diagonal
 for (i in 1:(n-1)) {
   idx <- which(abs(A[i:n,i]) == max(abs(A[i:n,i])))
   idx <- idx[length(idx)] # If more than one, pick the last
   idx <- i+idx-1
   N <- A[i,]
   A[i,] <- A[idx,]
   A[idx,] <- N
   N <- b[i]
   b[i] <- b[idx]
```

```
  b[idx] <- N
 }

 # Each a_{ii} greater than sum of rest
 ans <- TRUE
 for (i in 1:n) {
   ff <- abs(A[i,i])
   ss <- sum(abs(A[1,]))-ff
   if (ff <= ss) ans <- FALSE
 }

 return(ans)
}
```

The function returns `TRUE` or `FALSE` depending on whether A is diagonally dominant or not. We know, for instance, that A is not diagonally dominant. Let us check that.

```
ans <- check_dominant(A)
print(ans)
#> [1] FALSE
```

Next, a loop can be set up in which the elements on the diagonal are increased of one unit at each cycle. At each cycle the new matrix is checked for diagonal dominance. The iterations are stopped when the matrix is found diagonally dominant.

```
ans <- TRUE
i <- 0
while (ans) {
  i <- i+1
  diag(A) <- diag(A)+i
  ans <- !(check_dominant(A))
}
print(i)
#> [1] 9
```

So the answer to this question is that the positive integer making A dominant is 9.

c

It is then easy to find the solution using the two methods, as the current A is diagonally dominant.

```
# A is diagonally dominant
ans <- check_dominant(A)
print(ans)
#> [1] TRUE

# Solution using Jacobi
xP <- PJacobi(A,b)
#> Number of cycles needed to converge: 36
#> Last relative increment: 9.39080491257904e-07

# Solution using Gauss-Seidel
xG <- GSeidel(A,b)
#> Number of cycles needed to converge: 10
#> Last relative increment: 2.38275426389656e-07

# The two solutions are equal
```

```
absdif <- sum(abs(xP-xG))
print(absdif)
#> [1] 1.611361e-07
```

4.2.11 Exercise 16

To study the convergence of both the Jacobi and the Gauss-Seidel algorithm, the norm of the approximate solutions, $\mathbf{x}^{(i)}$, can be measured and plotted with respect to the iteration number. This is one of the many ways to study convergence, initially at a qualitative level.

In this exercise you should try and modify `PJacobi` to observe convergence qualitatively. Call the function you have modified, `MPJacobi`. Its input is the same as the one for `PJacobi`, but its output, besides the solution `x`, is the norm of all approximations from cycle 1 to the last cycle. Then plot the norms against the iteration number and verify that the values converge to a finite value. Try calculating the solution using different starting solutions and comment qualitatively on the character of the convergence.

Use the following non diagonally dominant matrix A and column vector $\mathbf{b}$

$$A = \begin{pmatrix} 3 & 4 & -1 & 1 & 5 \\ -1 & 1 & 0 & 4 & -1 \\ 1 & 2 & 3 & 5 & -1 \\ 1 & 4 & 0 & -1 & 0 \\ -1 & 2 & 0 & 3 & 4 \end{pmatrix} \quad , \quad \mathbf{b} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

for all calculations.

SOLUTION

The parts of `PJacobi` that are modified are shown in the following extract from the code of the functions `MPJacobi`.

```
.........
.........
.........
# Iterations
  x <- x0

  # New vector containing the norm of x, cycle by cycle
  normx <- norm(as.matrix(x),type="2")

  for (icyc in 1:nmax) {
.........
.........
.........
    # Update values
    x <- x + Deltax

    # Norm of x
    normx <- c(normx,norm(as.matrix(x),type="2"))
.........
.........
.........
  }
.........
.........
.........
  # Modify output
```

```
  #return(x)
  ltmp <- list(x=x,norms=normx)
  return(ltmp)
}
```

The modified function, written in the file *modified_PJacobi.R*, is then loaded in memory with `source` (make sure to have this file, containing `MPJacobi`, available for the command `source`).

```
source("./modified_PJacobi.R")
```

To investigate the convergence to solution, we can apply `MPJacobi` to the linear system, starting from three different values of $\mathbf{x}^{(0)}$, chosen arbitrarily:

$$\mathbf{x}_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad , \quad \mathbf{x}_2 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \quad , \quad \mathbf{x}_3 = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{pmatrix}$$

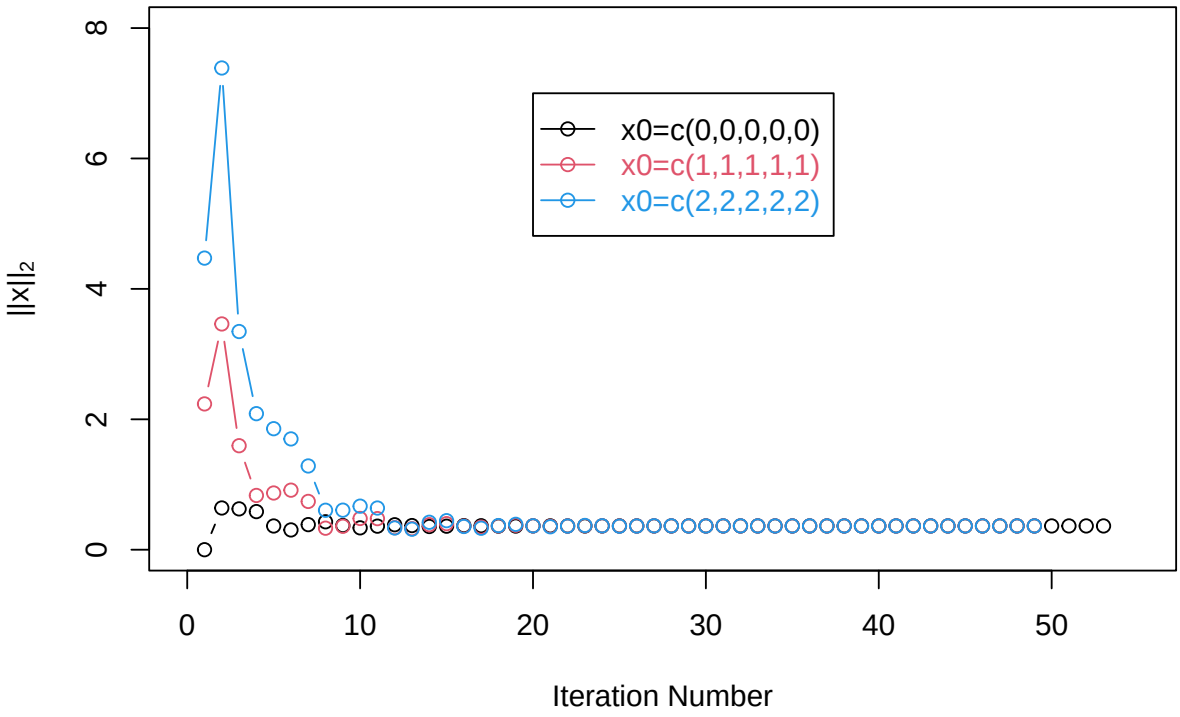
Other values can be chosen and even a systematic set of different starting solutions can be adopted. Here we will be happy just to show how a qualitative study of convergence via the norm of the successive solutions, can be implemented. The creation of all norms needed is obtained in the following passage.

```
# Create the A and b suggested
A <- matrix(c( 3, 4,-1, 1, 5,
              -1, 1, 0, 4,-1,
               1, 2, 3, 5,-1,
               1, 4, 0,-1, 0,
              -1, 2, 0, 3, 4),ncol=5,byrow=TRUE)
b <- matrix(c(1,1,1,1,1),ncol=1)

# Solution and sequence of norms when starting from (0,0,0,0,0)
lP1 <- MPJacobi(A,b,ddominant=FALSE)
#> The matrix of coefficients is not diagonally dominant.
#> Attempting solution anyway...
#>
#> Number of cycles needed to converge: 52
#> Last relative increment: 7.37413211826521e-07
print(lP1$x)
#> [1] -0.09745764  0.30932196 -0.08474564  0.13983061 -0.03389816

# Solution and sequence of norms when starting from (1,1,1,1,1)
lP2 <- MPJacobi(A,b,ddominant=FALSE,x0=c(1,1,1,1,1))
#> The matrix of coefficients is not diagonally dominant.
#> Attempting solution anyway...
#>
#> Number of cycles needed to converge: 48
#> Last relative increment: 6.99668518677754e-07
print(lP2$x)
#> [1] -0.09745850  0.30932323 -0.08474777  0.13982963 -0.03390047

# Solution and sequence of norms when starting from (2,2,2,2,2)
lP3 <- MPJacobi(A,b,ddominant=FALSE,x0=c(2,2,2,2,2))
#> The matrix of coefficients is not diagonally dominant.
#> Attempting solution anyway...
#>
```

```
#> Number of cycles needed to converge: 48
#> Last relative increment: 7.26060911282467e-07
print(lP3$x)
#> [1] -0.09745917  0.30932460 -0.08475005  0.13982839 -0.03390300
```

We can then plot the norms with respect to the iteration number, and see qualitatively how convergence to the solution is obtained by the algorithm. In the plots one has to consider that the range for each series varies and therefore a common range will have to be worked out using the function `range`.

```
# Work out range for the three convergence plots
print(range(lP1$norms,lP2$norms,lP3$norms))
#> [1] 0.000000 7.387583

# The range ylim=c(0,8) will then be chosen
# Also, xlim=c(0,55), as this includes the total
# number of cycles in the three cases (52,48,48)

# First plot (x0=c(0,0,0,0,0))
plot(lP1$norms,type="b",xlim=c(0,55),ylim=c(0,8),
     xlab="Iteration Number",
     ylab=expression(paste("||",x,"||"[2])))

# Second plot (x0=c(1,1,1,1,1))
points(lP2$norms,type="b",col=2)

# Third plot (x0=c(2,2,2,2,2))
points(lP3$norms,type="b",col=4)

# Add legend
legend(x=20,y=7,text.col=c(1,2,4),legend=c("x0=c(0,0,0,0,0)",
                                           "x0=c(1,1,1,1,1)",
                                           "x0=c(2,2,2,2,2)"),
       pch=c(1,1,1),lty=c(1,1,1),col=c(1,2,4))
```



From a quick observation of the plot, one can form the idea that convergence acts in two stages. First, the approximate solution is rapidly pushed in a close region around the final solution (here withinn the first 10-15 cycles). Then the algorithm takes longer to reach the required precision, corresponding to a more accurate convergence to the correct solution.

Several and varied ways to study convergence with norms are possible and in this exercise we have just touched lightly on one way to do that. When the code for the algorithm is available (most of the code is available in R), then it is important to know how to interact with it in order to extract the information needed to study convergence or other features of the algorithm.

4.2.12 Exercise 17

Carry out the calculations presented in Chapter 4 to introduce ill-conditioning. In the text we have two matrices. The first,

$$A_1 = \begin{pmatrix} 2 & 3 \\ 1 & 1 \end{pmatrix},$$

is well-conditioned, the second,

$$A_2 = \begin{pmatrix} 2 & 1.99 \\ 1 & 1.00 \end{pmatrix},$$

is ill-conditioned. They have been used to solve the two systems,

$$A_1\mathbf{x} = \mathbf{b}_1 \quad , \quad A_2\mathbf{x} = \mathbf{b}_2,$$

where

$$\mathbf{b}_1 = \begin{pmatrix} 7 \\ 3 \end{pmatrix} \quad , \quad \mathbf{b}_2 = \begin{pmatrix} 5.99 \\ 3 \end{pmatrix}.$$

Conditioning of the two systems is manifested when $\mathbf{b}_1$ and $\mathbf{b}_2$ are *perturbed*, i.e. slightly changed into the two vectors,

$$\mathbf{b}_1^{'} = \begin{pmatrix} 6.99 \\ 3.01 \end{pmatrix}, \qquad \mathbf{b}_2^{'} = \begin{pmatrix} 6.00 \\ 2.99 \end{pmatrix}.$$

These can be re-written as

$$\mathbf{b}_1^{'} = \mathbf{b}_1 - \Delta\mathbf{b}_1, \qquad \mathbf{b}_2^{'} = \mathbf{b}_2 - \Delta\mathbf{b}_2,$$

with

$$\Delta\mathbf{b}_1 = \begin{pmatrix} -0.01 \\ 0.01 \end{pmatrix}, \qquad \Delta\mathbf{b}_2 = \begin{pmatrix} 0.01 \\ -0.01 \end{pmatrix}.$$

Using the `norm` function in R, calculate the following quantities,

$$\|A_1\|,\ \|A_2\|,\ \|A_1^{-1}\|,\ \|A_2^{-1}\|,$$

$$\|\mathbf{b}_1\|,\ \|\mathbf{b}_2\|,$$

$$\|\Delta\mathbf{b}_1\|,\ \|\Delta\mathbf{b}_2\|.$$

Calculate also the solutions of the linear systems with the original and perturbed right hand sides, using the `solve` R function. You should verify that the relative error, $\|\Delta\mathbf{x}\|$, satisfies the inequality provided in the text:

$$\frac{1}{\|A^{-1}\|\|A\|}\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} \leq \frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A\|\|A^{-1}\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}.$$

For all calculations use both the infinity-norm (yielding the numbers in the text) and the Frobenius norm.

SOLUTION

Let us first create in memory all matrices and vectors needed.

```
# Well-conditioned matrix
A1 <- matrix(c(2,1,3,1),ncol=2)

# Its inverse
A1inv <- solve(A1)

# Ill-conditioned matrix
A2 <- matrix(c(2,1,1.99,1),ncol=2)

# Its inverse
A2inv <- solve(A2)

# b1 and b1'
b1 <- matrix(c(7,3),ncol=1)
bb1 <- matrix(c(6.99,3.01),ncol=1)

# b2 and b2'
b2 <- matrix(c(5.99,3.00),ncol=1)
bb2 <- matrix(c(6.00,2.99),ncol=1)
```

Next, the solutions of the four systems involved are calculated with `solve`.

```
# Solution A1*x=b1
x1 <- solve(A1,b1)

# Solution A1*x=bb1
xx1 <- solve(A1,bb1)

# The two solutions are not very different
```

```
print(x1)
#>      [,1]
#> [1,]    2
#> [2,]    1
print(xx1)
#>      [,1]
#> [1,] 2.04
#> [2,] 0.97

# Solution A2*x=b2
x2 <- solve(A2,b2)

# Solution A2*x=bb2
xx2 <- solve(A2,bb2)

# The two solutions are very different
print(x2)
#>      [,1]
#> [1,]    2
#> [2,]    1
print(xx2)
#>       [,1]
#> [1,]  4.99
#> [2,] -2.00
```

Now we can calculate all the norms involved, using the infinity-norm version. The condition numbers are also given.

```
# Norm for A1 and A1inv (infinity-norm)
nA1 <- norm(A1,"I")
nA1inv <- norm(A1inv,"I")
print(c(nA1,nA1inv))
#> [1] 5 4

# Condition number for A1
cn1 <- nA1*nA1inv
print(cn1)
#> [1] 20

# Norm for A2 and A2inv (infinity-norm)
nA2 <- norm(A2,"I")
nA2inv <- norm(A2inv,"I")
print(c(nA2,nA2inv))
#> [1]   3.99 300.00

# Condition number for A2
cn2 <- nA2*nA2inv
print(cn2)
#> [1] 1197

# Norm of b1 and Delta b1 (infinity-norm)
nb1 <- norm(b1,"I")
print(nb1)
#> [1] 7
```

```
nDb1 <- norm(b1-bb1,"I")
print(nDb1)
#> [1] 0.01

# Ratio Delta b1 / b1
rb1 <- nDb1/nb1
print(rb1)
#> [1] 0.001428571

# Norm of b2 and Delta b2 (infinity-norm)
nb2 <- norm(b2,"I")
print(nb2)
#> [1] 5.99
nDb2 <- norm(b2-bb2,"I")
print(nDb2)
#> [1] 0.01

# Ratio Delta b2 / b2
rb2 <- nDb2/nb2
print(rb2)
#> [1] 0.001669449
```

With the above quantities calculated we can now calculate the interval for the relative errors.

```
# Interval for the relative error of x1
lower1 <- rb1/cn1
upper1 <- cn1*rb1
print(c(lower1,upper1))
#> [1] 7.142857e-05 2.857143e-02

# Interval for the relative error of x2
lower2 <- rb2/cn2
upper2 <- cn2*rb2
print(c(lower2,upper2))
#> [1] 1.394694e-06 1.998331e+00
```

We should verify that the relative errors do fall inside the respective intervals. We will need to calculate the norm of the solutions and their difference.

```
# Norm of Delta x1, x1 and their ratio (infinity-norm)
nx1 <- norm(x1,"I")
nDx1 <- norm(x1-xx1,"I")
rx1 <- nDx1/nx1
print(c(nDx1,nx1,rx1))
#> [1] 0.04 2.00 0.02

# Norm of Delta x2, x2 and their ratio (infinity-norm)
nx2 <- norm(x2,"I")
nDx2 <- norm(x2-xx2,"I")
rx2 <- nDx2/nx2
print(c(nDx2,nx2,rx2))
#> [1] 3.0 2.0 1.5

# The values are within the estimated range
```

```r
# Dx1/x1
print(c(lower1,rx1,upper1))
#> [1] 7.142857e-05 2.000000e-02 2.857143e-02

# Dx2/x2
print(c(lower2,rx2,upper2))
#> [1] 1.394694e-06 1.500000e+00 1.998331e+00
```

In both cases the relative errors are inside the predicted range. All the calculations involving the norms can be carried out, obviously with different numerical values, using other types of norm. But the predicted interval should still contain the relative error. For the Frobenius norm we have, in the ill-conditioned case:

```r
# Norm of A2, A2inv and condition number
cn2 <- norm(A2,"F")*norm(A2inv,"F")
print(cn2)
#> [1] 996.01

# Norm of Delta b2, b2 and ratio
rb2 <- norm(b2-bb2,"F")/norm(b2,"F")
print(rb2)
#> [1] 0.002110999

# Norm of Delta x2, x2 and ratio
rx2 <- norm(x2-xx2,"F")/norm(x2,"F")
print(rx2)
#> [1] 1.894207

# Interval
lower2 <- rb2/cn2
upper2 <- cn2*rb2
print(c(lower2,rx2,upper2))
#> [1] 2.119456e-06 1.894207e+00 2.102576e+00
```

The relative error is still within the new interval.

4.2.13 Exercise 18

Most of the time, values of $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$, where $\mathbf{x}$ is the solution of a linear system, are far from their upper limit. More specifically, when in the linear system,

$$A\mathbf{x} = \mathbf{b}$$

the right hand side is perturbed, the solution is changed so that the relative error acquires values in the following interval,

$$\frac{1}{\|A^{-1}\|\|A\|}\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} \leq \frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A\|\|A^{-1}\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|},$$

where $\|\Delta\mathbf{b}\|/\|\mathbf{b}\|$ measures the relative change in the right hand side of the linear system. Very often, $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$ is far from the upper limit of the inequality, $\|A\|\|A^{-1}\|\|\Delta\mathbf{b}\|/\|\mathbf{b}\|$, so that the system behaves as well-conditioned, even if the matrix condition number is high. In fact, both the right hand side of the system and its change, $\Delta\mathbf{b}$, are important quantities when the stability of the solution is contemplated.

It is possible to investigate the range of values of $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$, once A, $\mathbf{b}$ and $\Delta\mathbf{b}$ are given. In this exercise you are required to prove, using the Frobenius norm, that $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$ never reaches its upper limit, if A is a 2×2 matrix.

SOLUTION

The inequalities involving the upper limit are,

$$\|\mathbf{x}\| \geq \frac{1}{\|A\|}\|\mathbf{b}\| \qquad \text{and} \qquad \|\Delta\mathbf{x}\| \leq \|A^{-1}\|\|\Delta\mathbf{b}\|,$$

where the norms are all Frobenius norms. For example, given the dimensions of the matrix A and the column vector $\mathbf{x}$,

$$\|A\| = \sqrt{a_{11}^2 + a_{12}^2 + a_{21}^2 + a_{22}^2} \qquad , \qquad \|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2}$$

The maximum value of $\|\Delta\mathbf{x}\|/\|\mathbf{x}\|$ is reached if and when the following two equations hold:

$$\|\mathbf{x}\| = \frac{1}{\|A\|}\|\mathbf{b}\| \quad \text{and} \quad \|\Delta\mathbf{x}\| = \|A^{-1}\|\|\Delta\mathbf{b}\|$$

Let us concentrate on the first of the two, as the demonstration for the second follows a similar argument.

The equation written using the Frobenius norm is:

$$\sqrt{x_1^2 + x_2^2} = \frac{1}{\|A\|}\sqrt{b_1^2 + b_2^2}$$

or, squaring both sides of the equation:

$$x_1^2 + x_2^2 = \frac{1}{\|A\|^2}(b_1^2 + b_2^2)$$

The components of $\mathbf{b}$, b_1, b_2 , are connected to the components of $\mathbf{x}$, x_1, x_2, via the linear system,

$$A\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad \left\{ \begin{array}{rcl} a_{11}x_1 + a_{12}x_2 & = & b_1 \\ a_{21}x_1 + a_{22}x_2 & = & b_2 \end{array} \right.$$

Therefore the relation between norms becomes:

$$x_1^2 + x_2^2 = \frac{1}{\|A\|^2}\left((a_{11}x_1 + a_{12}x_2)^2 + (a_{21}x_1 + a_{22}x_2)^2\right)$$

$$\Downarrow$$

$$\left(1 - \frac{1}{\|A\|^2}(a_{11}^2 + a_{21}^2)\right)x_1^2 + \left(1 - \frac{1}{\|A\|^2}(a_{12}^2 + a_{22}^2)\right)x_2^2 - \frac{2}{\|A\|^2}(a_{11}a_{12} + a_{21}a_{22})x_1x_2 = 0$$

The above is one equations with two unknowns. The solution will therefore include a free parameter. Here we will simply fix the second component as the free parameter k:

$$\left(1 - \frac{1}{\|A\|^2}(a_{11}^2 + a_{21}^2)\right)x_1^2 - \frac{2k}{\|A\|^2}(a_{11}a_{12} + a_{21}a_{22})x_1 + k^2\left(1 - \frac{1}{\|A\|^2}(a_{12}^2 + a_{22}^2)\right) = 0$$

The equation in the unknown x_1 is the second-degree equation,

$$ax_1^2 - kbx_1 + k^2c = 0,$$

where,

$$a = 1 - \frac{1}{\|A\|^2}(a_{11}^2 + a_{21}^2) \quad , \quad b = \frac{2k}{\|A\|^2}(a_{11}a_{12} + a_{21}a_{22}) \quad , \quad c = 1 - \frac{1}{\|A\|^2}(a_{12}^2 + a_{22}^2)$$

The equation has one or two solutions if the discriminant, Δ is greater or equal than zero:

$$\Delta = k^2b^2 - 4k^2ac = k^2(b^2 - 4ac) \geq 0$$

k^2 is always non-negative. Therefore let us calculate $b^2 - 4ac$.

$$b^2 - 4ac = \frac{4}{\|A\|^4}(a_{11}a_{12} + a_{21}a_{22})^2 - 4\left(1 - \frac{1}{\|A\|^2}(a_{11}^2 + a_{21}^2)\right)\left(1 - \frac{1}{\|A\|^2}(a_{12}^2 + a_{22}^2)\right)$$

$$\Downarrow$$

$$\begin{aligned} b^2 - 4ac &= \tfrac{4}{\|A\|^4}(a_{11}^2 a_{12}^2 + 2a_{11}a_{12}a_{21}a_{22} + a_{21}^2 a_{22}^2) \\ &\quad -4\left(1 - \tfrac{a_{11}^2+a_{21}^2+a_{12}^2+a_{22}^2}{\|A\|^2} + \tfrac{a_{11}^2a_{12}^2+a_{11}^2a_{22}^2+a_{21}^2a_{12}^2+a_{21}^2a_{22}^2}{\|A\|^4}\right) \\ &= \tfrac{4}{\|A\|^4}(a_{11}^2 a_{12}^2 + 2a_{11}a_{12}a_{21}a_{22} + a_{21}^2 a_{22}^2) \\ &\quad -4\left(1 - \tfrac{\|A\|^2}{\|A\|^2} + \tfrac{a_{11}^2a_{12}^2+a_{11}^2a_{22}^2+a_{21}^2a_{12}^2+a_{21}^2a_{22}^2}{\|A\|^4}\right) \\ &= \tfrac{4}{\|A\|^4}(a_{11}^2 a_{12}^2 + 2a_{11}a_{12}a_{21}a_{22} + a_{21}^2 a_{22}^2) \\ &\quad -\tfrac{4}{\|A\|^4}(a_{11}^2a_{12}^2 + a_{11}^2a_{22}^2 + a_{21}^2a_{12}^2 + a_{21}^2a_{22}^2) \end{aligned}$$

$$\Downarrow$$

$$b^2 - 4ac = -\frac{4}{\|A\|^4}(a_{11}^2a_{22}^2 + a_{21}^2a_{12}^2 - 2a_{11}a_{12}a_{21}a_{22})$$

$$\Downarrow$$

$$b^2 - 4ac = -\frac{4}{\|A\|^4}(a_{11}a_{22} + a_{12}a_{21})^2 < 0$$

Therefore, the discriminant is always negative, unless the matrix of coefficients is the trivial null matrix. This means that there are no solutions, $\mathbf{x}$, of the linear system that can cause the relative solution error to be equal to its upper limit.

4.2.14 Exercise 19

The function `illcond_sample` in `comphy` is a *toy* function that creates examples of linear systems that manifest ill-conditioning in a dramatic way. The highest theoretical limit for the solution relative error is often unreachable (see Exercise 18), but evident effects of ill-conditioning can be seen also without the relative error to reach such a limit. It is possible to find high values of the relative error, given the matrix A, if sampling of $\mathbf{b}$ and $\Delta\mathbf{b}$ is carried out in a statistically-meaningful way, i.e. with a high number of randomly generated values. This is what is achieved by `illcond_sample`.

1. Study the documentation and code of `illcond_sample` and try to understand how the function works.
2. The matrix,
$$A = \begin{pmatrix} 2 & 1.99 \\ 1 & 1.00 \end{pmatrix}$$
was used in the text to demonstrate the effects of ill-conditioning. With,
$$\mathbf{b} = \begin{pmatrix} 5.99 \\ 3.00 \end{pmatrix},\ \mathbf{b}' = \begin{pmatrix} 6.00 \\ 2.99 \end{pmatrix}$$
$$\Downarrow$$
$$\Delta\mathbf{b} \equiv \mathbf{b} - \mathbf{b}' = \begin{pmatrix} -0.01 \\ 0.01 \end{pmatrix}$$
the relative error using the Frobenius norm turns out to be, for the above values of $\mathbf{b}$ and $\Delta\mathbf{b}$,
$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} = 1.89421$$
The upper limit for this case is,
$$\|A^{-1}\|\|A\|\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} = 2.10258$$
Use `illcond_sample` to try and find a solution relative error greater than the one presented here. You will observe that it is relatively easy to reach a case with a relative solution error higher than the 1.89421 presented here, but this in general corresponds to an upper limit much higher than 2.10258. We can empirically observe that for the specific matrix presented, the ratio between the relative solution error and its upper limit struggles to reach the value $1.89421/2.10258 = 0.90090$ of the case treated here.

Note, some of the numbers here are different from those in the main text because there the norm were infinity-norms, while here they are Frobenius norms.

SOLUTION

The function takes other input besides the matrix. One possibility to increase sampling is to generate more random $\mathbf{b}$'s, where the components of each random $\mathbf{b}$ are fished from the uniform distribution. By default the number of random generations is `ncyc=100000`. We can increase `ncyc` to 500000 and perhaps re-execute the function a few times with a different starting seed.

```
# Matrix
A <- matrix(c(2,1,1.99,1),ncol=2)

# First attempt (iseed=1786)
ltmp1 <- illcond_sample(A,ncyc=500000,iseed=1786)
#> Relative error: 413.685246696998 < 580.149547812422: upper limit.
#> Ratio: 0.713066567502959

# Second attempt (iseed=1787)
ltmp2 <- illcond_sample(A,ncyc=500000,iseed=1787)
#> Relative error: 648.925626820576 < 747.84595698469: upper limit.
#> Ratio: 0.867726328877995
```

Both scenarions don't reach the 0.90090 ratio previously found, but have relative errors much larger than the related 1.89421 value. This is reflected in a change in solution, $\Delta\mathbf{x}$, more dramatic than the change,

$$\Delta\mathbf{x} = \begin{pmatrix} -2.99 \\ 3.00 \end{pmatrix}$$

manifested in the scenario previously found.

```
# Try just the first smpled value
b <- ltmp1$b
Db <- ltmp1$Db
b2 <- b-Db
x <- solve(A,b)
print(x)
#>             [,1]
#> [1,] 0.67441924
#> [2,] 0.09327894
x2 <- solve(A,b2)
print(x2)
#>            [,1]
#> [1,]  199.3330
#> [2,] -199.5647
Dx <- x-x2
print(Dx)
#>            [,1]
#> [1,] -198.6586
#> [2,]  199.6580
```

5 Chapter 05

5.1 Exercises on least squares

5.1.1 Exercise 01

Study the main function for linear regression, `solveLS`, in `comphy` and apply it to estimate the parameters, a_1, a_2, a_3, a_4, of the linear model

$$y = a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4$$

The data points are listed in the following table.

x_1	x_2	x_3	y
−0.959	0.829	0.419	4.327
−0.781	−0.218	0.752	6.513
−0.957	0.549	0.499	4.847
0.413	−0.184	0.919	9.838
−0.989	−0.683	0.819	6.931
0.460	−0.274	−0.362	4.606
−0.479	0.739	−0.156	2.738
−0.436	−0.229	−0.664	2.191
−0.329	0.289	0.663	6.657
−0.866	0.662	−0.252	1.630
0.988	−0.635	0.957	11.556
0.885	−0.422	−0.736	4.372
−0.760	−0.231	0.596	6.010
−0.338	0.705	0.502	5.397
−0.734	0.551	−0.991	−1.245

SOLUTION

The function `solveLS` can be studied if its source code is dumped on the screen. This is readily done simply typing `solveLS` without parentheses.

```
solveLS
#> function(x,intercept=TRUE,tol=NULL) {
#>   # Check input
#>   ans <- (is.matrix(x) | is.data.frame(x))
#>   if (!ans) {
#>     msg <- "Input has to be either a matrix or a data frame\n"
#>     cat(msg)
#>     return(NULL)
#>   }
#>
#>   # Number of data points (n) and parameters (m+1)
#>   tmp <- dim(x)
#>   n <- tmp[1]
#>   m <- tmp[2]-1
#>
#>   # Build A and y matrices
#>   ones <- matrix(rep(1,times=n),ncol=1)
#>   A <- as.matrix(x[,1:m])
#>   colnames(A) <- NULL
#>   rownames(A) <- NULL
#>   if (intercept) A <- cbind(A,ones)
#>   y <- matrix(x[,m+1],ncol=1)
#>   colnames(y) <- NULL
```

```
#>   rownames(y) <- NULL
#>
#>   # Build F and g matrices to get solution through solve
#>   F <- t(A) %*% A
#>   g <- t(A) %*% y
#>
#>   # Check whether F is singular
#>   d <- det(F)
#>   if (abs(d) < 1e-6) {
#>     msg <- paste0("There are infinite solutions to ",
#>                   "this least squares fitting.\n")
#>     cat(msg)
#>     return(NULL)
#>   }
#>
#>   # Change tolerance, if required
#>   if (is.null(tol)) tol <- 1e-17
#>
#>   # Solution
#>   a <- solve(F,g,tol=tol)
#>
#>   # Print out the sum of squared residuals:
#>   eps <- y-A %*% a
#>   d <- sum(eps^2)
#>   msg <- sprintf("Sum of squared residuals: %f\n",d)
#>   cat(msg)
#>
#>   # Reshape for output
#>   a <- as.vector(a)
#>
#>   return(a)
#> }
#> <bytecode: 0x00000289cb8214d0>
#> <environment: namespace:comphy>
```

The code is essentially an implementation of the matrix form of the least squares' normal equations, where a prominent role is played by the matrix A in which the last column is filled with 1's. For example, in the case of the model studied in this exercise, we have

$$A = (\mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3 \quad \mathbf{1})$$

An exception, cleverly coded with the `intercept` parameter, is implemented for the case in which the constant term (a_{m+1}) is not part of the model (or $a_{m+1} = 0$). In this case the column filled with 1's is not included in A (check the one-liner in which this is taken care of).

Another important aspect of the code is that when the matrix F of coefficients of the normal equations is ill-conditioned, then the `solve` function can fail, unless the `tol` keyword is appropriately small. The default is `1e-17`, but the user can input smaller values if the default makes execution fail.

Let us now try and solve the regression problem. First we have to insert the tabulated data into an array structure, say a data frame.

```
# Data are saved into a data frame
x1 <- c(-0.959,-0.781,-0.957,0.413,-0.989,0.460,-0.479,-0.436,
        -0.329,-0.866,0.988,0.885,-0.760,-0.338,-0.734)
x2 <- c(0.829,-0.218,0.549,-0.184,-0.683,-0.274,0.739,-0.229,
```

```
        0.289,0.662,-0.635,-0.422,-0.231,0.705,0.551)
x3 <- c(0.419,0.752,0.499,0.919,0.819,-0.362,-0.156,-0.664,
        0.663,-0.252,0.957,-0.736,0.596,0.502,-0.991)
y <- c(4.327,6.513,4.847,9.838,6.931,4.606,2.738,2.191,
       6.657,1.630,11.556,4.372,6.010,5.397,-1.245)
x <- data.frame(x1=x1,x2=x2,x3=x3,y=y)
print(x)
#>         x1     x2     x3      y
#> 1  -0.959  0.829  0.419  4.327
#> 2  -0.781 -0.218  0.752  6.513
#> 3  -0.957  0.549  0.499  4.847
#> 4   0.413 -0.184  0.919  9.838
#> 5  -0.989 -0.683  0.819  6.931
#> 6   0.460 -0.274 -0.362  4.606
#> 7  -0.479  0.739 -0.156  2.738
#> 8  -0.436 -0.229 -0.664  2.191
#> 9  -0.329  0.289  0.663  6.657
#> 10 -0.866  0.662 -0.252  1.630
#> 11  0.988 -0.635  0.957 11.556
#> 12  0.885 -0.422 -0.736  4.372
#> 13 -0.760 -0.231  0.596  6.010
#> 14 -0.338  0.705  0.502  5.397
#> 15 -0.734  0.551 -0.991 -1.245
```

Then we can proceed with the regression. The function prints out also the sum of squared residuals (SSE).

```
a <- solveLS(x)
#> Sum of squared residuals: 0.694310

# Present result with same precision as the data
a <- round(a,digits=3)
print(a)
#> [1]  2.009 -0.987  3.996  5.050
```

The SSE is relatively small. So it's expected that the data variability is well covered by the linear model found.

5.1.2 Exercise 02

Simulate a set of 15 data points coming from measuring x of the following physical law,

$$x(t) = 5t + \sin(2\pi t) + \cos(2\pi t) \quad , \quad 0 \leq t \leq 3$$

for the values of t equal to

$$t = 0.0\ ,\ 0.2\ ,\ 0.6\ ,\ 1.8\ ,\ 1.9\ ,\ 2.5\ ,\ 2.6$$

and where data are affected by random errors distributed according to the normal distribution with mean 0 and standard deviation 0.5. Use the seed 5522 for the simulation. Organise the data thus simulated into a data frame and use `solveLS`to estimate the physical law, assuming the following model,

$$x = a_1 t + a_2 \sin(2\pi t) + a_3 \cos(2\pi t)$$

You should find that the estimate of a_1, a_2, a_3 is close to their theoretical value, $a_1 = 5, a_2 = 1, a_3 = 1$. What is the SSE if the model does not have the cosine components? On a same graph, plot the data points simulated and the two regression curves corresponding to the two models.

SOLUTION

The simulation is done simply by creating an array for the 7 values of t, calculating the x array based on the formula suggested, and adding 7 randomly generated values from a normal distribution.

```
# Sampled values of t (regular grid)
t <- c(0,0.2,0.6,1.8,1.9,2.5,2.6)

# Random errors (with seed suggested)
set.seed(5522)
eps <- rnorm(7,mean=0,sd=0.5)

# Simulated data
x <- 5*t+sin(2*pi*t)+cos(2*pi*t)+eps
```

To find the estimate for the 3 parameters of the model suggested, a multilinear regression is needed with the three variables,

$$x_1 = t \quad , \quad x_2 = \sin(2\pi t) \quad , \quad x_3 = \cos(2\pi t)$$

Thus the linear model is:

$$y = a_1 x_1 + a_2 x_2 + a_3 x_3$$

It is important for the use of the `solveLS` function, to consider that the intercept term is not present in the model. The data are turned into a data frame, prior to using them in `solveLS`.

```
# Variables for the linear model
x1 <- t
x2 <- sin(2*pi*t)
x3 <- cos(2*pi*t)

# Data frame for use in solveLS
ddd <- data.frame(x1=x1,x2=x2,x3=x3,x=x)
print(round(ddd,digits=3))
#>    x1     x2     x3      x
#> 1 0.0  0.000  1.000  1.590
#> 2 0.2  0.951  0.309  1.651
#> 3 0.6 -0.588 -0.809  1.628
#> 4 1.8 -0.951  0.309  8.390
#> 5 1.9 -0.588  0.809  9.172
#> 6 2.5  0.000 -1.000 10.558
#> 7 2.6 -0.588 -0.809 11.419

# Parameter estimation via regression
a <- solveLS(ddd,intercept=FALSE)
#> Sum of squared residuals: 0.524553
print(a)
#> [1] 4.6927321 0.4277631 1.0984756
```

The estimated coefficients are relatively close to the $5, 1, 1$ values of the simulated data. The difference is obviously due to the effect of the random differences. If the model is

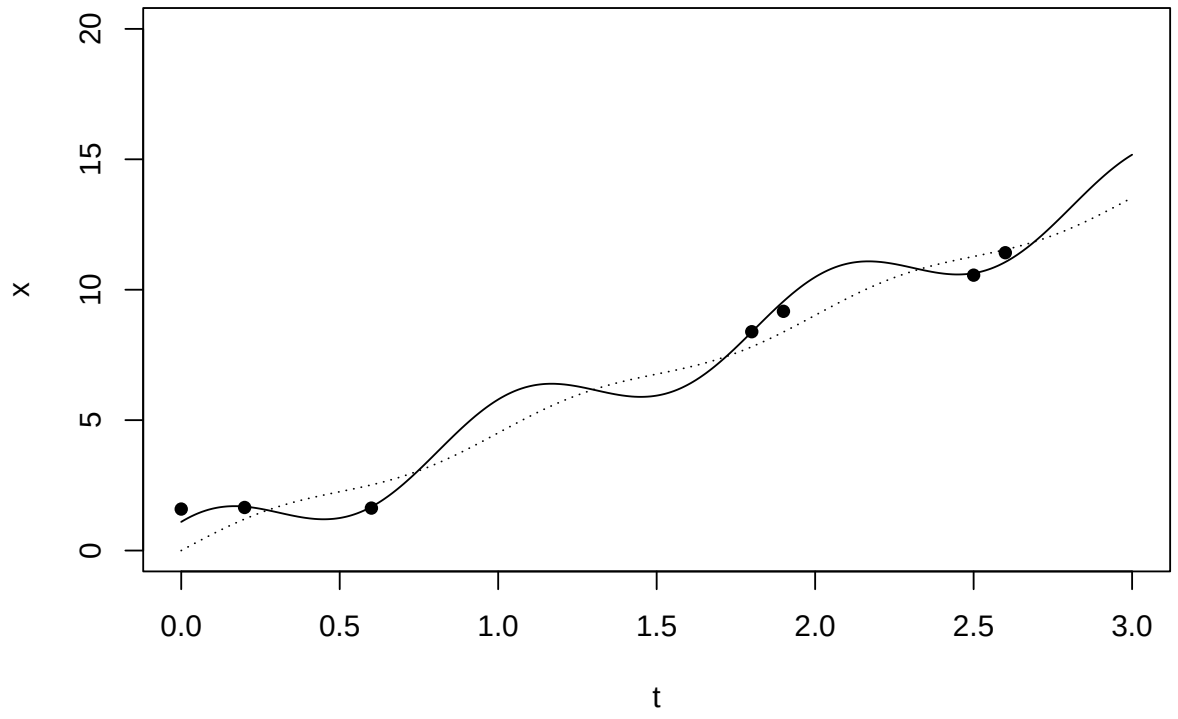
$$y = a_1 x_1 + a_2 x_2,$$

i.e. if the cosine term is dropped, the regression yields:

```
# Regression without cosine term
aa <- solveLS(ddd[c(1:2,4)],intercept=FALSE)
#> Sum of squared residuals: 5.003979
print(aa)
#> [1] 4.5098132 0.3208128
```

The SSE, 5.004, is much higher than the value 0.525 found with the previous model. This indicates that the second model provides a worst fitting, compared to the first. Visually, the fit quality can be appreciated by looking at the plot of the data points and the fitting curves.

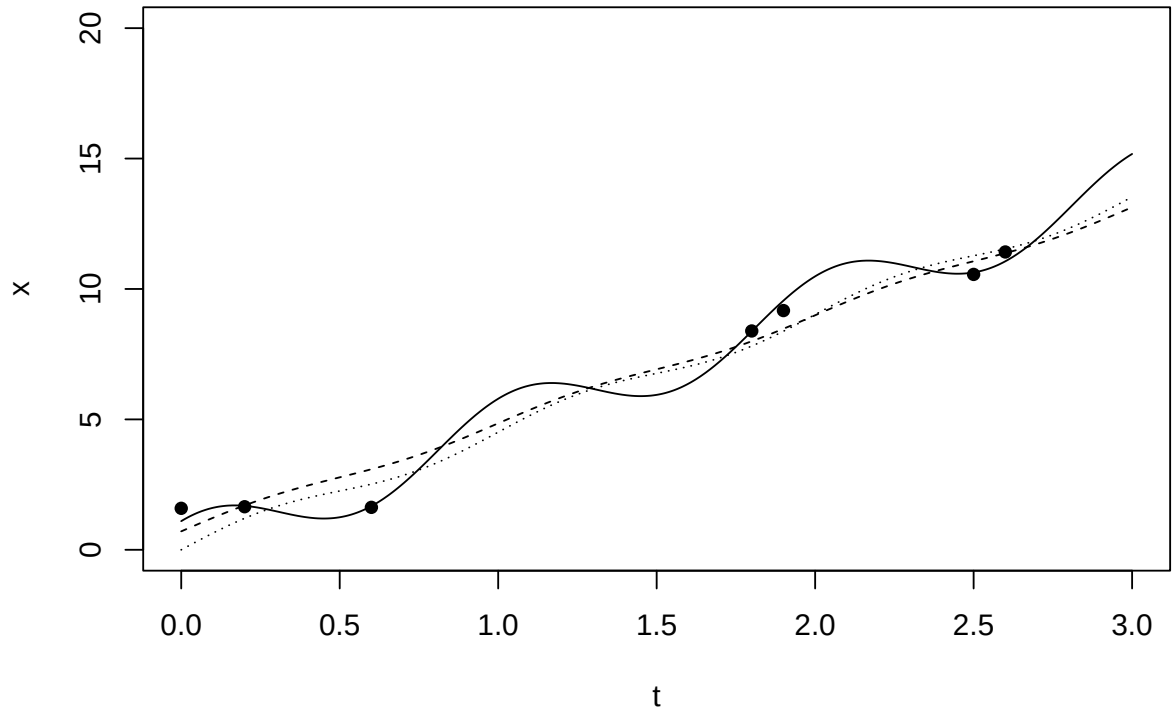
```
# Plot simulated points
plot(t,x,pch=16,xlab="t",ylab="x",
     xlim=c(0,3),ylim=c(0,20))

# Finer grid for plot
tt <- seq(0,3,length=1000)

# First curve
xx1 <- a[1]*tt+a[2]*sin(2*pi*tt)+a[3]*cos(2*pi*tt)

# Second curve
xx2 <- aa[1]*tt+aa[2]*sin(2*pi*tt)

# Plot both curves (dashed one is the second curve)
points(tt,xx1,type="l")
points(tt,xx2,type="l",lty=3)
```



It might be asked whether the intercept term made things better, in the second case.

```
# Regression without cosine term, but with intercept term
aaa <- solveLS(ddd[c(1:2,4)])
#> Sum of squared residuals: 3.816207
print(aaa)
```

```
#> [1] 4.1421249 0.1752267 0.7074379

# New curve
xx3 <- aaa[1]*tt+aaa[2]*sin(2*pi*tt)+aaa[3]

# Plot all curves
# (dashed one is the second curve, dotted one the new curve)
# Plot simulated points
plot(t,x,pch=16,xlab="t",ylab="x",
     xlim=c(0,3),ylim=c(0,20))
points(tt,xx1,type="l")
points(tt,xx2,type="l",lty=3)
points(tt,xx3,type="l",lty=2)
```



The SSE is definitely better (3.816), but the curve is still struggling do account for the data properly. In situations like this, it is clear that the model has one or more terms missing, or that it is entirely wrong.

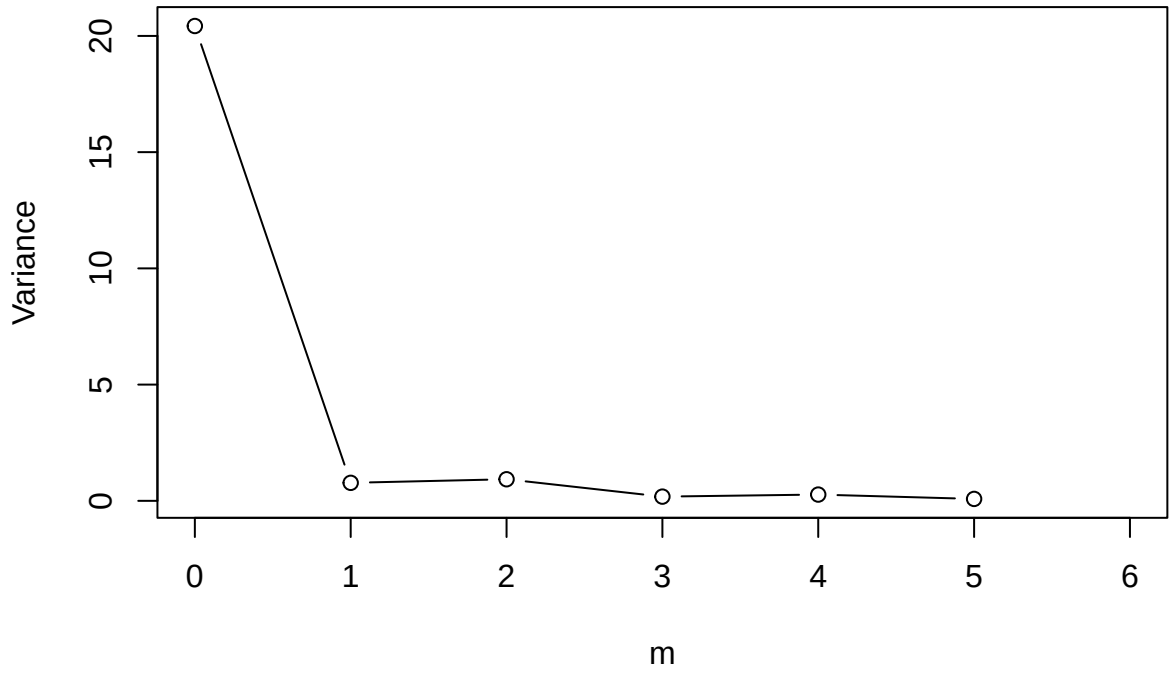
5.1.3 Exercise 03

Using the simulated data of Exercise 02, find out which polynomial fits them best, using the variance σ_e^2 as criterion.

SOLUTION

The criterion makes use of the plot m versus σ_e^2, where m is the polynomial's degree. We can use the `comphy` function `which_poly` to do that.

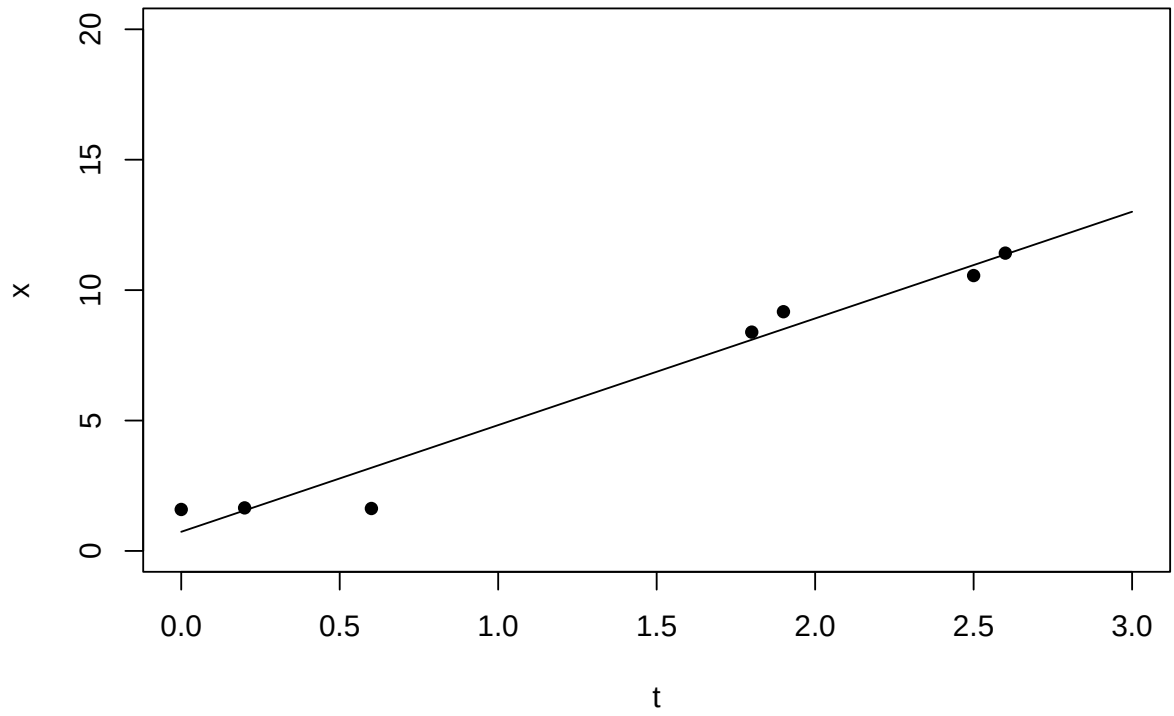
```
# Reproduce simulated data of Exercise 02
t <- c(0,0.2,0.6,1.8,1.9,2.5,2.6)
set.seed(5522)
eps <- rnorm(7,mean=0,sd=0.5)
x <- 5*t+sin(2*pi*t)+cos(2*pi*t)+eps

# Data frame for input
pts <- data.frame(t=t,x=x)

# Test for best polynomial (plot provided by default)
# The output is a data frame
dtmp <- which_poly(pts)
```



From the plot is very clear that a polynomial of degree one can account for most data variability, degrees higher than one would most likely model data noise. The fit is, subsequently, carried out with the function `polysolveLS`.

```
# Linear regression with a straight line
ltmp <- polysolveLS(pts,1)
a <- ltmp$a
print(a)
#> [1] 4.0902289 0.7344686
print(ltmp$SSE)
#> [1] 3.87083

# Finer grid for plotting the regression curve
tt <- seq(0,3,length=1000)
```

```r
# Regression curve
xx <- a[1]*tt+a[2]

# Plot
plot(t,x,pch=16,xlab="t",ylab="x",
     xlim=c(0,3),ylim=c(0,20))
points(tt,xx,type="l")
```



The sum of squared residuals is SSE= 3.871, even better than the values found in the last models of Exercise 02. This means that, unless an additional trigonometric contribution is known to be part of the model, it is better to avoid it as an unnecessary complication. One should also notice that the estimate of a_1 is close to the value 5 used to simulate the data, which makes sense because $5t$ is the trend dominating the data in this case.

5.1.4 Exercise 04

Consider the dampened oscillations of a body of mass $m = 5$ kg, attached to a spring of unknown constant k, and oscillating horizontally, with angular frequency $\omega = 1.998$ (rad/sec), on a smoothed plane which progressively decreases the amplitude of the oscillation. The motion of the body is described by the equation

$$x = x(t) = Ae^{-bt/(2m)}\cos(\omega t),$$

which describes the position of the body around its equilibrium position. In the above equation,

$$\omega = \sqrt{\frac{k}{m} - \left(\frac{b}{2m}\right)^2}$$

In order to calculate the spring constant, k, and the coefficient b that causes the dampening, a series of $n = 11$ regular measurements of the body's position is taken at regular time intervals between 0 seconds and 20 seconds. The values are summarised in the following table.

t (sec)	x (m)
0	16.218
2	−11.947
4	−1.979
6	5.046
8	−4.620
10	2.821
12	2.026
14	−3.459
16	2.597
18	0.959
20	−0.982

Using linear regression, estimate the value of both k (N/m) and b (N sec/m).

SOLUTION

When the natural logarithm of both sides of the formula describing the oscillation is taken, the result is:

$$\log(x) = \log(A) - \frac{b}{2m}t + \log(\cos(\omega t))$$

As both x and ωt are known at the 11 sampled points, the above relation can be re-written as,

$$\log(x) - \log(\cos(\omega t)) = -\frac{b}{2m}t + \log(A)$$

This is equivalent to the linear model,

$$y = a_1 t + a_2,$$

where,

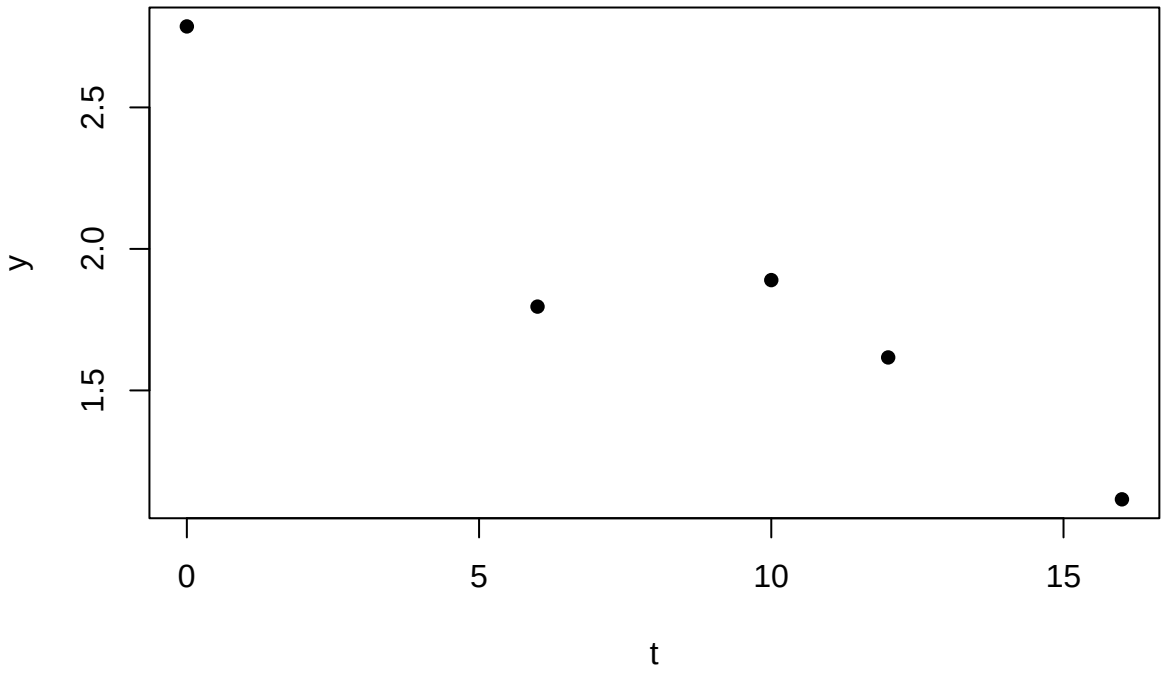
$$y \equiv \log(x) - \log(\cos(\omega t)) \quad , \quad a_1 = -\frac{b}{2m} \quad , \quad a_2 = \log(A)$$

Let us load the data in memory so that all calculation can be carried out using R.

```
t <- c(0,2,4,6,8,10,12,14,16,18,20)
x <- c(16.218,-11.947,-1.979,5.046,-4.620,
        2.821,  2.026,-3.459,2.597, 0.959,-0.982)

# Omega is 1.99750
#om <- 1.99750
om <- 1.998

# Other constants
m <- 5

# New variable y
y <- log(x)-log(cos(om*t))
#> Warning in log(x): NaNs produced
#> Warning in log(cos(om * t)): NaNs produced
```

The warning printed out is reminding us that some of the arguments of the logarithm are non positive, which is not allowed. We will have, therefore to make a selection of the data, if we want to persevere with the original plan of performing linear regression. The data are not many, but this is our only option. This is how only the positive values can be selected, remembering that also $\cos(\omega t)$ must be positive.

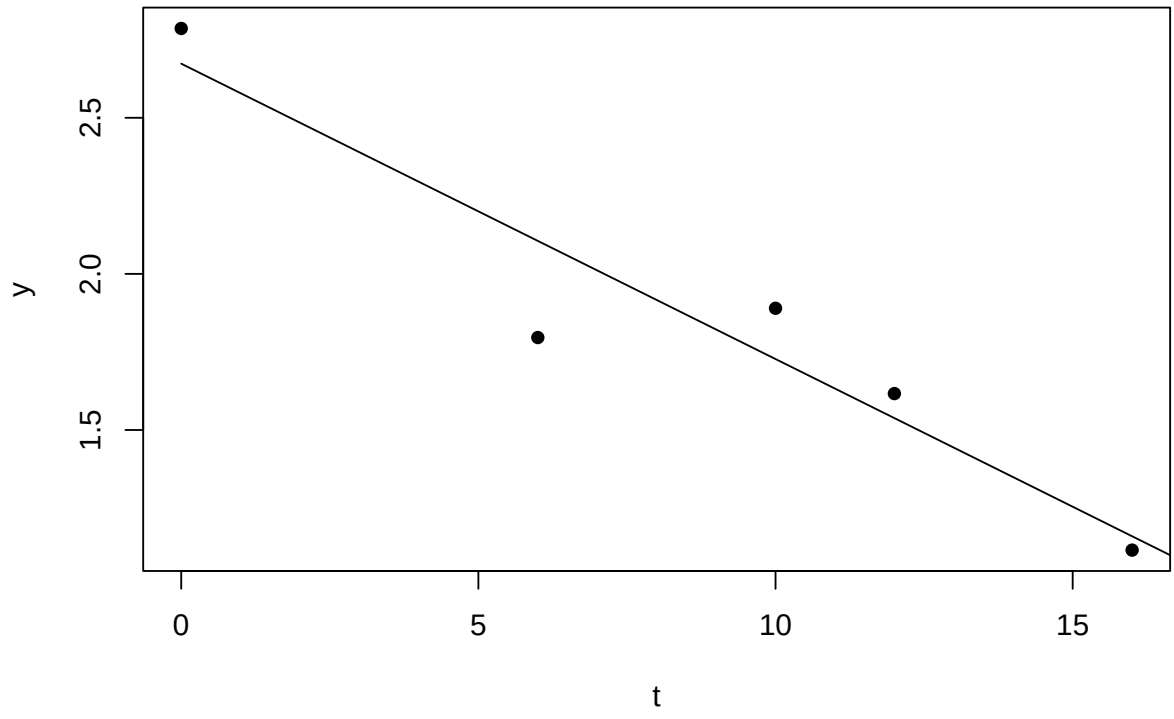
```r
idx <- which(x > 0 & cos(om*t) > 0)

# New data
newt <- t[idx]
newx <- x[idx]

# Data for regression
newy <- log(newx)-log(cos(om*newt))
pts <- data.frame(t=newt,y=newy)

# The data plot should give the impression of a
# negative gradient
plot(pts,pch=16,xlab="t",ylab="y")
```



The regression can be carried out using `solveLS`. The regression curve is shown here, overlapped with the few data points available.

```r
# Regression
a <- solveLS(pts)
#> Sum of squared residuals: 0.143123
print(a)
#> [1] -0.09460272  2.67325789

# Regression curve
tt <- seq(0,20,length=1000)
yy <- a[1]*tt+a[2]
```

```
plot(pts,pch=16,xlab="t",ylab="y")
points(tt,yy,type="l")
```




```
# Value of b
b <- -2*m*a[1]
print(b)
#> [1] 0.9460272

# Value of A
A <- exp(a[2])
print(A)
#> [1] 14.48709
```

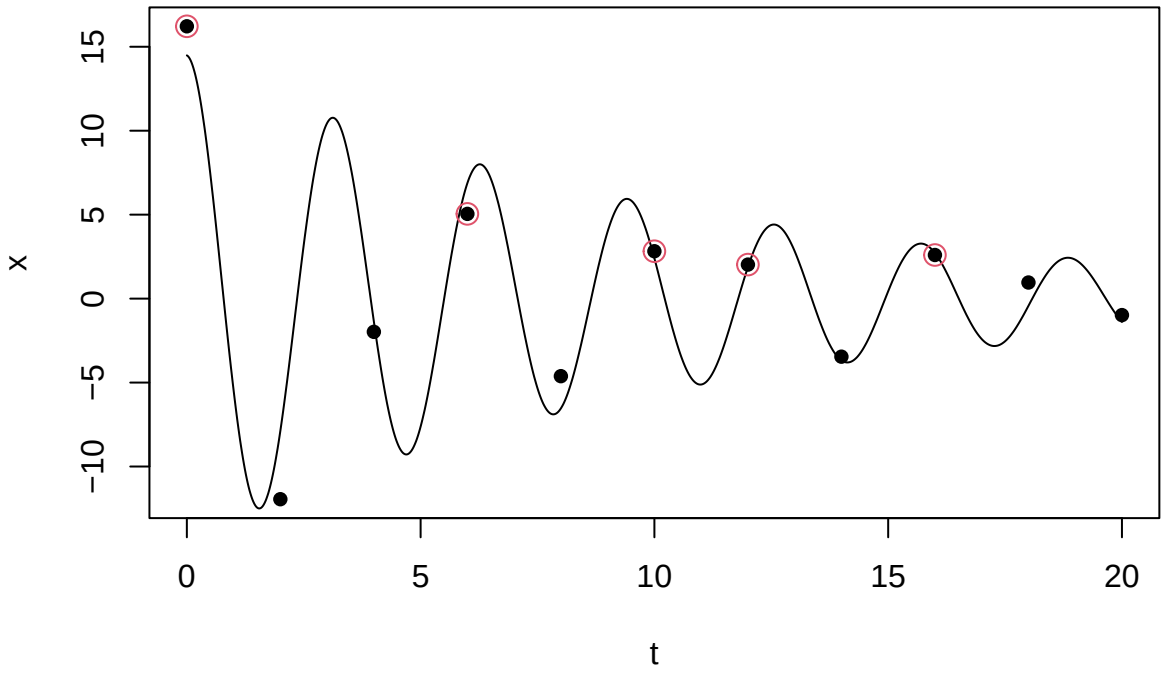
The fit appears to be reasonable and the SSE is also relatively small (0.141). The advantage of having carried out the regression is that now we have an estimate of A and b. Knowledge of b implies knowledge of k:

$$\omega = \sqrt{\frac{k}{m} - \left(\frac{b}{2m}\right)^2} \quad \rightarrow \quad k = m\omega^2 + \frac{b^2}{4m}$$

All parameters needed to determine the model are thus available and can be used to check how the curve determined fits all data.

```
# Value of k
k <- m*om^2+b^2/(4*m)
print(k)
#> [1] 20.00477
```

```r
# Final estimated curve overlapped on data
xx <- A*exp(-b*tt/(2*m))*cos(om*tt)

# Plot
# (open red circles are the data used for initial estimate)
plot(t,x,pch=16,xlab="t",ylab="x")
points(tt,xx,type="l")
points(t[idx],x[idx],cex=1.5,col=2)
```



The important feature of the plot above is that the points that have not been used for the regression are relatively close to the estimated curve. This is a good indication that the fitting is relatively accurate. The unused data kind of *validate* the goodness of fit, a feature known as *cross validation* in data analysis.

5.1.5 Exercise 05

Water is discharged from a container, according to the following dynamical formula:

$$h(t) = a + \frac{b}{1+\sqrt{t}},$$

where h is the height of the discharging water surface above the ground, and a and b two positive constants related to the water's height before discharging starts (time $t = 0$) and after it has finished (time $t = \infty$).

The level has been measured at random, specified times and the values are reported in the following table.

t (sec)	h (cm)
3.5	117.6
7.1	113.7
14.1	110.5
28.2	107.9
31.8	107.5
49.4	106.2
52.9	106.0
56.5	105.9

Using least squares regression, find the two constants a, b and the water's height before discharging starts and after it has finished.

SOLUTION

The levels of water before discharging has started, and after it has finished are:

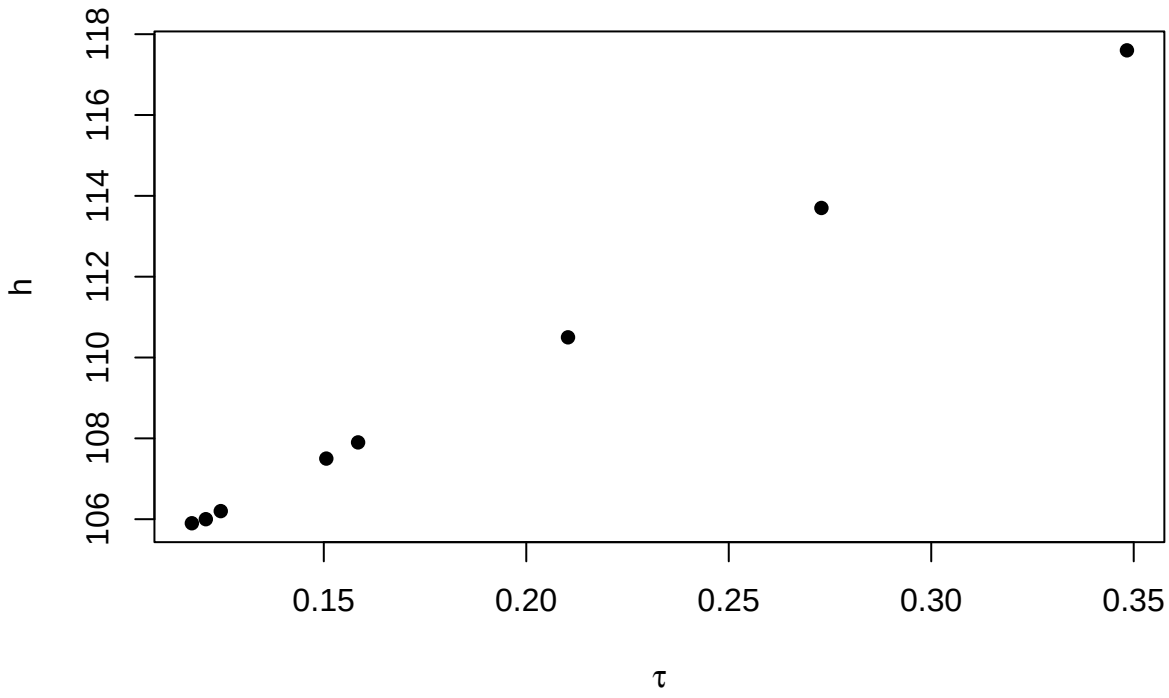
$$h_0 = h(0) = a + b \qquad , \qquad h_f = \lim_{t\to\infty} h(t) = a$$

These levels can be calculated once the parameters a and b have been estimated. This can be done *via* linear regression because the main relation between t and h can be transformed and made linear. It will suffice to take $1/(1+\sqrt{t})$ as the new independent regression variable, while leaving h as the dependent variable.

$$h = a + \frac{b}{1+t} \quad \rightarrow \quad h = a + b\tau \quad \text{, with} \quad \tau = \frac{1}{1+t}$$

Let us load the data in memory and calculate how they are transformed.

```
# Experimental data
t <- c(3.5,7.1,14.1,28.2,31.8,49.4,52.9,56.5)
h <- c(117.6,113.7,110.5,107.9,107.5,106.2,106.0,105.9)

# Transformed data
newt <- 1/(1+sqrt(t))
newh <- h

# Save them in a data frame
pts <- data.frame(newt,newh)
print(pts)
#>        newt  newh
#> 1 0.3483315 117.6
#> 2 0.2728824 113.7
#> 3 0.2103051 110.5
#> 4 0.1584694 107.9
#> 5 0.1506217 107.5
#> 6 0.1245561 106.2
#> 7 0.1208716 106.0
#> 8 0.1174171 105.9

# Plot data to verify visually they are roughly
# located around a straight line
plot(pts,pch=16,xlab=expression(tau),ylab=expression(h))
```



The alignment of the data points along a seemingly straight line is a strong indication that the relation used is correct. Next, the regression coefficients can be estimated via least squares.

```
# Linear regression
coeffs <- solveLS(pts)
#> Sum of squared residuals: 0.010183
print(coeffs)
#> [1] 50.77876 99.86955
```

The SSE is very small, as expected (SSE=0.010). The estimated coefficients are,

$$a = 99.9 \qquad , \qquad b = 50.8$$

Therefore in the full container the water has level 150.7 cm, while once it is fully discharged (at $t = \infty$, meaning after a sufficiently long time) the level is 99.9 cm.

5.2 Exercises on statistical linear regression

5.2.1 Exercise 06

Using the `lm` function, find the estimated parameters for the following models:

1. $y = 2x + 1$
2. $y = 6x_1 - 2x_2 + 3x_3 + 1$

The independent variables, x, x_1, x_2, x_3 have the following values:

$$\begin{aligned} x &= \{0.23, 0.31, 0.34, 0.64, 0.65, 1.42, 2.07, 3.03, 3.17, 3.27, 4.96\} \\ x_1 &= \{0.09, 0.34, 1.00, 1.04, 2.05, 2.05, 2.59, 3.44, 4.44, 4.59, 4.99\} \\ x_2 &= \{1.39, 4.51, 4.63, 5.12, 5.50, 7.89, 8.03, 9.48, 9.64, 9.64, 9.94\} \\ x_3 &= \{5.13, 5.41, 5.45, 5.49, 5.62, 5.67, 5.93, 6.41, 6.79, 6.97, 7.77\} \end{aligned}$$

The errors (residuals) for the first simulation are generated by a random distribution with mean 0 and variance $\sigma^2 = 0.01$ (use seed 1132), while for the second simulation the mean is still 0 and the variance $\sigma^2 = 0.04$ (use seed 2311).

SOLUTION

Let us start with the first simulation.

```
# Simulating y=2x+1
x <- c(0.23,0.31,0.34,0.64,0.65,1.42,2.07,3.03,3.17,3.27,4.96)
n <- length(x)
s <- sqrt(0.01)
set.seed(1132)
y <- 2*x+1+rnorm(n,mean=0,sd=s)

# Regression
md1 <- lm(y ~ x)
slm1 <- summary(md1)
print(slm1)
#>
#> Call:
#> lm(formula = y ~ x)
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -0.16455 -0.05798  0.02546  0.05808  0.12603
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  1.06220    0.04629   22.94 2.70e-09 ***
#> x            1.97191    0.01951  101.06 4.61e-15 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.09801 on 9 degrees of freedom
#> Multiple R-squared:  0.9991, Adjusted R-squared:  0.999
#> F-statistic: 1.021e+04 on 1 and 9 DF,  p-value: 4.614e-15

# Estimated variance
print(slm1$sigma^2)
#> [1] 0.009605766
```

So the estimates for the first model, $y = a_1 x + a_2$ are 1.97191 for a_1 (close to its correct value, 2) and 1.06220 for a_2 (close to its correct value, 1). The estimated variance is 0.00961, relatively close to its correct value, 0.01.

For the second simulation we have.

```
# Simulating y=6x1-2x2+3x3+1
x1 <- c(0.09,0.34,1.00,1.04,2.05,2.05,2.59,3.44,4.44,4.59,4.99)
n <- length(x1)
```

```r
x2 <- c(1.39,4.51,4.63,5.12,5.50,7.89,8.03,9.48,9.64,9.64,9.94)
x3 <- c(5.13,5.41,5.45,5.49,5.62,5.67,5.93,6.41,6.79,6.97,7.77)
s <- sqrt(0.04)
set.seed(2311)
y <- 6*x1-2*x2+3*x3+1+rnorm(n,mean=0,sd=s)

# Regression
md2 <- lm(y ~ x1+x2+x3)
slm2 <- summary(md2)
print(slm2)
#>
#> Call:
#> lm(formula = y ~ x1 + x2 + x3)
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -0.40968 -0.11751  0.03982  0.10186  0.28137
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  2.95958    1.68074   1.761    0.122
#> x1           6.12695    0.21405  28.624 1.63e-08 ***
#> x2          -1.97420    0.07677 -25.714 3.44e-08 ***
#> x3           2.59066    0.31827   8.140 8.16e-05 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.2387 on 7 degrees of freedom
#> Multiple R-squared:  0.9994, Adjusted R-squared:  0.9991
#> F-statistic:  3637 on 3 and 7 DF,  p-value: 1.553e-11

# variance
print(slm2$sigma^2)
#> [1] 0.05696041
```

If the model is written as $y = a_1x_1 + a_2x_2 + a_3x_3 + a_4$, the estimate for a_1, a_2, a_3 are relatively good (by looking at the number of stars at the end of the corresponding lines, the chances of those values being accidental are really low). But the estimate of a_4 is not very reliable and, in fact, the estimate is not close to the correct value, 1. The estimate of the variance σ^2 is 0.05696, which is slightly higher than the correct value, 0.04.

5.2.2 Exercise 07

The estimation of parameters in multilinear regression can be problematic when one or more independent variables are correlated (*multicollinearity*). This exercise is about perfect collinearity between two independent variables in a linear model of the type,

$$y = a_1x_1 + a_2x_2 + a_3$$

The exercise (and its solution) provides insights on the problem of multicollinearity and on how this can be explained and managed.

The data points are listed in the following table:

x_1	x_2	y
0.0	−5.0	−7.16
0.5	−4.5	−3.06
1.0	−4.0	0.96
1.5	−3.5	5.19
2.0	−3.0	9.02
2.5	−2.5	13.02
3.0	−2.0	17.10
3.5	−1.5	21.17
4.0	−1.0	24.86
4.5	−0.5	29.27
5.0	0.0	32.82
5.5	0.5	37.00
6.0	1.0	40.90
6.5	1.5	44.95
7.0	2.0	48.91
7.5	2.5	53.08
8.0	3.0	56.95
8.5	3.5	60.81
9.0	4.0	65.18
9.5	4.5	69.22
10.0	5.0	73.04

a. Try the least squares regression on these data, using the function `lm`. You will not be able to proceed. Can you explain why?

b. Calculate, using the function `cor`, the correlation between x_1 and x_2.

c. Using the result from part b, attempt a way out the problem met in part a, and finally find an estimate for a_1, a_2, a_3.

SOLUTION

Part a

First, we should try and load the data in memory.

```
# Data points
x1 <- c(0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,
        5.5,6.0,6.5,7.0,7.5,8.0,8.5,9.0,9.5,10.0)
x2 <- c(-5.0,-4.5,-4.0,-3.5,-3.0,-2.5,-2.0,-1.5,-1.0,-0.5,0.0,
        0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0)
y <- c(-7.16,-3.06,0.96,5.19,9.02,13.02,17.10,21.17,24.86,29.27,
       32.82,37.00,40.90,44.95,48.91,53.08,56.95,60.81,65.18,69.22,73.04)
```

Then we use `lm` with the appropriate linear model.

```
# Regression
# summary returns an informative message
model <- lm(y ~ x1+x2)
summary(model)
#>
#> Call:
#> lm(formula = y ~ x1 + x2)
#>
#> Residuals:
```

```
#>      Min       1Q   Median       3Q      Max
#> -0.21813 -0.11077 -0.01341  0.09887  0.26150
#>
#> Coefficients: (1 not defined because of singularities)
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -7.01359    0.05898  -118.9   <2e-16 ***
#> x1           8.00491    0.01009   793.4   <2e-16 ***
#> x2                NA         NA      NA       NA
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.14 on 19 degrees of freedom
#> Multiple R-squared:      1,  Adjusted R-squared:      1
#> F-statistic: 6.294e+05 on 1 and 19 DF,  p-value: < 2.2e-16
```

One of the parameters has not been determined and a message is printed, warning us that one of them is not defined because of singularities. This is due to the independent variables not being really independent, but being, in fact, collinear.

Part b

In order to check how severe the multicollinearity is, we can calculate the correlation between x_1 and x_2.

```
c
#> function (...)  .Primitive("c")
# Correlation between x1 and x2
print(cor(x1,x2))
#> [1] 1
```

The correlation is exactly 1. This can only happen when one of the two variables is a linear function of the other, i.e.

$$x_2 = \alpha x_1 + \beta$$

Part c Given what found in part c, we can eliminate x_2 from the linear model and carry out regression analysis with two, rather than three, parameters. To find the coefficients α and β, we can use two of the data points (x_1, x_2) provided. So, using,

$$(x_1, x_2) = (0, -5) \qquad \text{and} \qquad (x_1, x_2) = (1, -4)$$

we have:

$$\begin{cases} -5 = 0\alpha + \beta \\ -4 = 1\alpha + \beta \end{cases} \quad \Rightarrow \quad \begin{cases} \alpha = 1 \\ \beta = -5 \end{cases}$$

The transformation is, therefore,

$$x_2 = x_1 - 5$$

What we have to do, next, is to ignore x_2 and carry out the regression only using x_1 and y. The parameters found can then be used to recover the parameters of the original regression. In fact,

$$y = a_1 x_1 + a_2 x_2 + a_3$$

$$\Downarrow$$

$$y = a_1 x_1 + a_2 (x_1 - 5) + a_3$$

$$\Downarrow$$

$$y = (a_1 + a_2) x_1 + (-5a_2 + a_3) \equiv a_1^{'} x_1 + a_2^{'},$$

where $a_1^{'}$ and $a_2^{'}$ are the parameters of the new regression. These are only two, not enough to recover the three parameters of the original model. Inverting the transformation, we have thus

$$\begin{array}{rcl} a_1^{'} &=& a_1 + a_2 \\ a_2^{'} &=& -5a_2 + a_3 \end{array} \quad \Rightarrow \quad \begin{array}{rcl} a_1 &=& a_1^{'} - \gamma \\ a_2 &=& \gamma \\ a_3 &=& a_2^{'} + 5\gamma, \end{array}$$

where γ is an arbitrary parameter.

```
# New regression (y=a'_1x_1+a'_2)
model2 <- lm(y ~ x1)
summary(model2)
#>
#> Call:
#> lm(formula = y ~ x1)
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -0.21813 -0.11077 -0.01341  0.09887  0.26150
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -7.01359    0.05898  -118.9   <2e-16 ***
#> x1           8.00491    0.01009   793.4   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.14 on 19 degrees of freedom
#> Multiple R-squared:      1,  Adjusted R-squared:      1
#> F-statistic: 6.294e+05 on 1 and 19 DF,  p-value: < 2.2e-16
```

The regression returns clear values for both $a_1^{'}$ and $a_2^{'}$. We have, therefore:

$$\begin{array}{rcll} a_1 &=& 8.005 - \gamma & \\ a_2 &=& \gamma & \text{with} \qquad \gamma \in \mathbb{R} \\ a_3 &=& -7.014 + 5\gamma & \end{array}$$

The set of parameters satisfying the above relations are the coefficients we were looking for. The equation,

$$y = a_1 x_1 + a_2 x_2 + a_3$$

represents a plane in the Cartesian system in which the coordinates are represented on the x_1 axis, the x_2 axis and the y axis. Due to the collinearity of x_1 and x_2, the plane is not unique. It is, in fact, represented by parametric equations, where the free parameter is the γ introduced earlier. All the planes intersect at a straight line described by the parametric equations:

$$x_1 = \delta, \; x_2 = \delta - 5, \; y = 8.005\delta - 7.014$$

These parametric equations can be obtained simply as intersection of any two planes of the parametric family above (say the one with $\gamma = 1.005$ and the one with $\gamma = 0$), where $x_1 = \delta$ is the parameter used.

6 Chapter 06

6.1 Exercises on the roots of nonlinear equations

6.1.1 Exercise 01

The only root of $f(x) = \cos(x)$ in the interval $[0, \pi]$ is $x = \pi/2$. Find the numerical value of this root with a precision of 12 digits, using `roots_bisec`, and compare it with the correct value $\pi/2$.

SOLUTION

For the specific function $f(x) = \cos(x)$ there is no need to define it because it is part of functions already implemented in R by default. Then we can use the suggested interval, $[0, \pi]$, as the search interval. The tolerance, `tol`, will have to be changed to 10^{-12}. To check the numerical result coincides with the analytic one to 12 decimal figures, we need to switch to a higher precision, using `options`.

```
# Increase display precision (one more than the suggested 12)
backup_options <- options()
options(digits=13)

# Use roots_bisec (no need to define a function cos
# because it already exists).
xx <- roots_bisec(cos,lB=0,rB=pi,tol=1e-12)
#> Searching interval: [0.000000,3.141593].
#> The root is 1.570796. The error is less than 1.000000e-12.

# Comparison
print(pi/2)
#> [1] 1.570796326795
print(xx)
#> [1] 1.570796326796

# Back to default accuracy
options(backup_options)
```

6.1.2 Exercise 02

When the search interval for `roots_bisec` is symmetric with respect to the two roots of a function, only one of the roots will be found, due to the way the algorithm works. This exercise is devoted to understanding such a mechanism, using the function $f(x) = x^2 - 1$, which has the two roots -1 and $+1$. Any interval symmetric with respect to 0, and including both -1 and $+1$, will trigger the output only of $x = -1$.

1 Try the claim using the search intervals $[-2, 2]$ and $[-3, 3]$.

2 Explore the code of `roots_bisec` and consider the following specific chunk, inside the main `while` loop:

```
if (f(a,...)*f(c,...) == 0 | f(b,...)*f(c,...) == 0) {
      a <- c
      b <- c
    } else if (f(a,...)*f(c,...) < 0) {
      b <- c
    } else if (f(b,...)*f(c,...) < 0) {
      a <- c
    }
```

a and b are the left extreme and right extreme of the search interval, respectively. c is their mid point. When the two extremes, $f(a), f(c)$, have different signs, a stays the same, but b becomes c; the line coming next in the `if` sequence is ignored as the current line satisfies it. Therefore, the second search interval will always be, in this case, on the left. This is, essentially, the reason why when we consider an interval symmetric around 0, for the function considered, the root found is always the one closest to the left of the interval.

3 Consider the function $f(x) = x^3 - 4x^2 + 2x$. It has three zeros at $x = 0, 1, 2$. Choose any two roots and try to find one of them with `roots_bisec`, using a search interval symmetric with respect to the mid point of the roots chosen. Do you observe the same problematic highlighted earlier? What happens if you choose a search interval symmetric with respect to $x = 1$ and including both 0 and 2? Can you explain why?

SOLUTION

1 The first part of the exercise is easy to carry out, once $x^2 - 1$ is implemented as function `f`.

```
# Function f(x)=x^2-1
f <- function(x) {return(x^2-1)}

# Root search with [-2,2]
xx <- roots_bisec(f,lB=-2,rB=2)
#> Searching interval: [-2.000000,2.000000].
#> The root is -1.000000. The error is less than 1.000000e-09.

# Root search with [-3,3]
xx <- roots_bisec(f,lB=-3,rB=3)
#> Searching interval: [-3.000000,3.000000].
#> The root is -1.000000. The error is less than 1.000000e-09.
```

2 Until the left extreme of the search interval includes the leftmost root, $x = -1$, this will be the root returned by the function. Therefore, if we use a number at the right of $x = -1$ as the left extreme, the root returned should be $x = 1$. This is tried with a couple of intervals in the following code.

```
# The left extreme is -1.1
# The root found is -1
xx <- roots_bisec(f,lB=-1.1,rB=2)
#> Searching interval: [-1.100000,2.000000].
#> The root is -1.000000. The error is less than 1.000000e-09.

# The left extreme is -0.9
# The root found is +1
xx <- roots_bisec(f,lB=-0.9,rB=2)
#> Searching interval: [-0.900000,2.000000].
#> The root is 1.000000. The error is less than 1.000000e-09.
```

3 We have to change the function for this part. Then we can consider symmetric search intervals around the roots $x = 0, 1$ and/or the roots $x = 1, 2$. The result will always be the leftmost root, i.e. 0 in the first case and 1 in the second case.

```
# New function
f <- function(x) {return(x^3-3*x^2+2*x)}

# Search interval including 0 and 1
xx <- roots_bisec(f,lB=-0.5,rB=1.5)
#> Searching interval: [-0.500000,1.500000].
#> The root is 0.000000. The error is less than 1.000000e-09.

# Search interval including 1 and 2
xx <- roots_bisec(f,lB=0.5,rB=2.5)
#> Searching interval: [0.500000,2.500000].
#> The root is 1.000000. The error is less than 1.000000e-09.
```

When a symmetric search interval around 1 is selected, the root found is 1 because that coincides with the mid point of the seach interval.

```
# Symmetric interval around 1 and including all roots.
# The root returned is 1
xx <- roots_bisec(f,lB=-0.5,rB=2.5)
#> Searching interval: [-0.500000,2.500000].
#> The root is 1.000000. The error is less than 1.000000e-09.
```

The goal of this exercise was to show that situations like those just depicted can happen, due to the particular

symmetry of the problem under study. It is thus important to make sure that the presence of multiple roots is acknowledged and fully explored. A plot of the function can often help to select the appropriate search interval.

6.1.3 Exercise 03

Finding the zeros of a function is also related to finding its optimal points. These can be found as zeros of the function's first derivative. Consider the fractional function

$$f(x) = \frac{x^3 + 6x^2 - x - 30}{x - 2}.$$

As this function is the ratio of a third degree and first degree polynomials, it has the same behaviour of a second degree polynomial. It has therefore only one optimal point. Find its optimal point using `roots_bisec`. Then plot the function between -20 and 20, and highlight its optimal point.

SOLUTION

The first derivative of the given function is

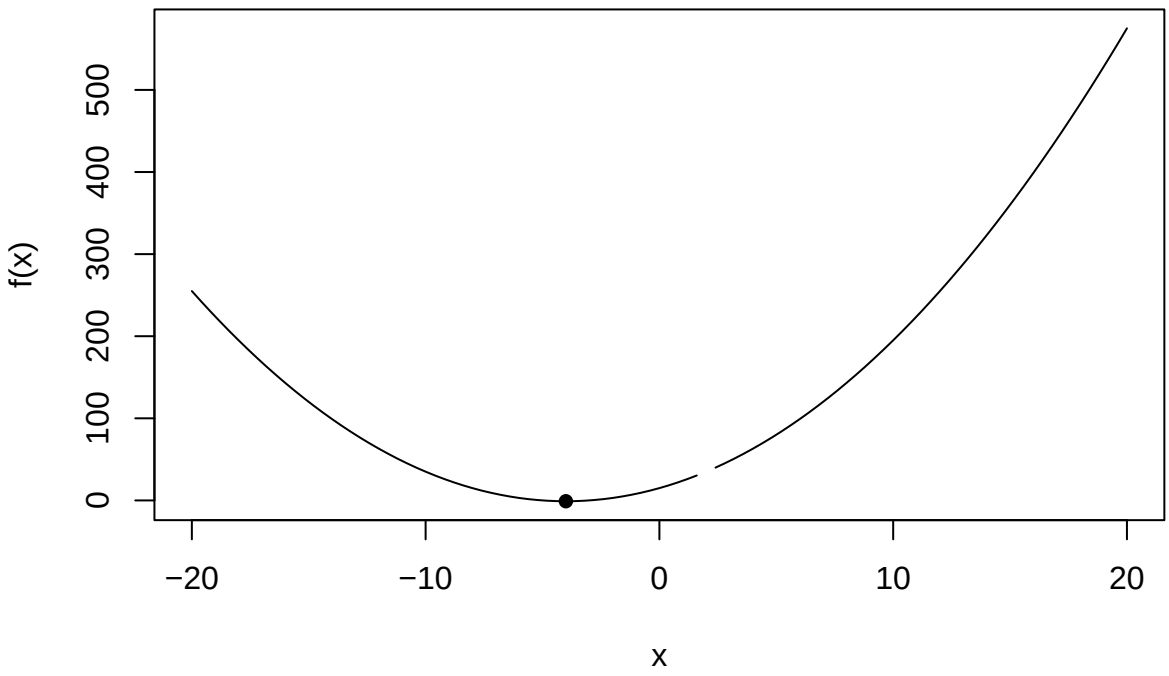
$$\frac{(3x^2 + 12x - 1)(x - 2) - (x^3 + 6x^2 - x - 30)}{(x - 2)^2}.$$

We do not need to simplify this function because it would make little difference when this is implemented in the code. To find the optimal point we might want to avoid involving $x = 2$ in the search because the function is not define there. In fact, the function becomes $0/0$ when $x = 2$ is replaced in its analytic form. We could try and simplify the function factorising $x - 2$, but will not bother doing this and will try a straight search with a large interval, say between -100 and 100.

```
# Function
f0 <- function(x) {return((x^3+6*x^2-x-30)/(x-2))}

# First derivative
f1 <- function(x) {return(((3*x^2+12*x-1)*(x-2)-(x^3+6*x^2-x-30))/(x-2)^2)}

# Optimal point
xx <- roots_bisec(f1,lB=-100,rB=100)
#> Searching interval: [-100.000000,100.000000].
#> The root is -4.000000. The error is less than 1.000000e-09.
yy <- f0(xx)

# Coordinates of the optimal point
print(c(xx,yy))
#> [1] -4 -1
```

We can check the nature of the optimal point graphically.

```
# Plot of function
curve(f0(x),from=-20,to=20,ylab="f(x)")

# Optimal point
points(xx,yy,pch=16)
```



The optimal point is clearly a minimum. Note that the curve is broken around $x = 2$ because the function is not defined there when its original expression is used. The R function `curve` thus avoids all points in a small neighbourhood of $x = 2$.

6.1.4 Exercise 04

Find the intersections between the curves C_1, given by the equation $y = 2\sin(6\pi x)$, and the curve C_2, given by the equation $y = e^{-x}$, in the interval $x \in [0, 1]$. Use Newton-Raphson for the numerical solutions.

SOLUTION

The x coordinate of the intersections are those values satisfying the equation

$$2\sin(6\pi x) = e^{-x}$$

The roots of this equation can be found as zeros of the function

$$f(x) = 2\sin(6\pi x) - e^{-x}$$

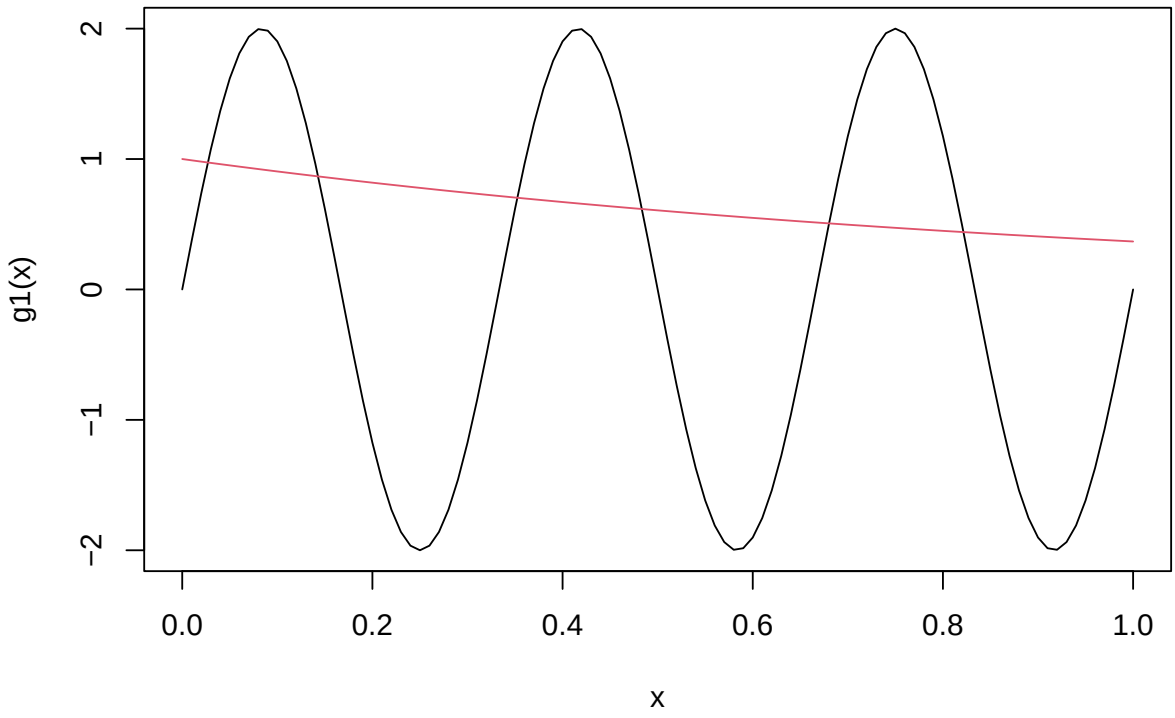
To use Newton-Raphson, we need the first derivative of this function, which is

$$f'(x) = 12\pi\cos(6\pi x) + e^{-x}$$

Before applying the function `roots_newton` to the problem, we need to have a rough idea of how many zeros the function has in the given interval, and where such zeros are. A plot can provide this information.

```
# Define the two functions
g1 <- function(x) return(2*sin(6*pi*x))
g2 <- function(x) return(exp(-x))
```

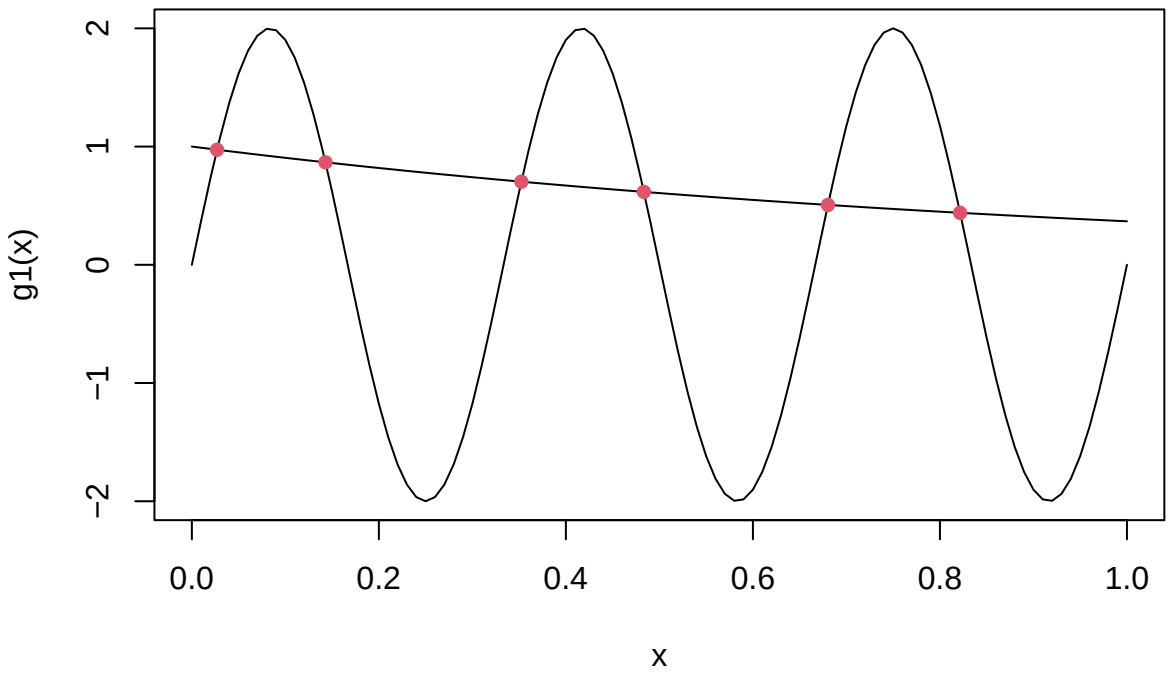
```r
# Plot functions between 0 and 1
curve(g1(x),from=0,to=1)
curve(g2(x),from=0,to=1,col=2,add=TRUE)
```



From the plot it is fairly clear that the intersections are six and that they happen roughly close to 0, 0.2, 0.3, 0.5, 0.7 and 0.9; these are the points we are going to use, in turn, when applying Newton-Raphson. The two functions $f(x)$ and $f'(x)$ will have to be define ahead of applying the method.

```r
# Function
f0 <- function(x) return(2*sin(6*pi*x)-exp(-x))

# First derivative
f1 <- function(x) return(12*pi*cos(6*pi*x)+exp(-x))

# Vector containing all intersections
vint <- c()

# Calculate intersections

# First
x0 <- 0
vint <- c(vint,roots_newton(f0,f1,x0))
#> The root is 0.026966. The error is less than 1.000000e-09.

# Second
```

```r
x0 <- 0.2
vint <- c(vint,roots_newton(f0,f1,x0))
#> The root is 0.142884. The error is less than 1.000000e-09.

# Third
x0 <- 0.3
vint <- c(vint,roots_newton(f0,f1,x0))
#> The root is 0.352388. The error is less than 1.000000e-09.

# Fourth
x0 <- 0.5
vint <- c(vint,roots_newton(f0,f1,x0))
#> The root is 0.483370. The error is less than 1.000000e-09.

# Fifth
x0 <- 0.7
vint <- c(vint,roots_newton(f0,f1,x0))
#> The root is 0.680250. The error is less than 1.000000e-09.

# Sixth
x0 <- 0.8
vint <- c(vint,roots_newton(f0,f1,x0))
#> The root is 0.821573. The error is less than 1.000000e-09.

# Plot for a visual check
yint <- g1(vint)
curve(g1(x),from=0,to=1)
curve(g2(x),from=0,to=1,add=TRUE)
points(vint,yint,pch=16,col=2)
```



The visual check confirms that the numerical solutions found are very close to their correct value.

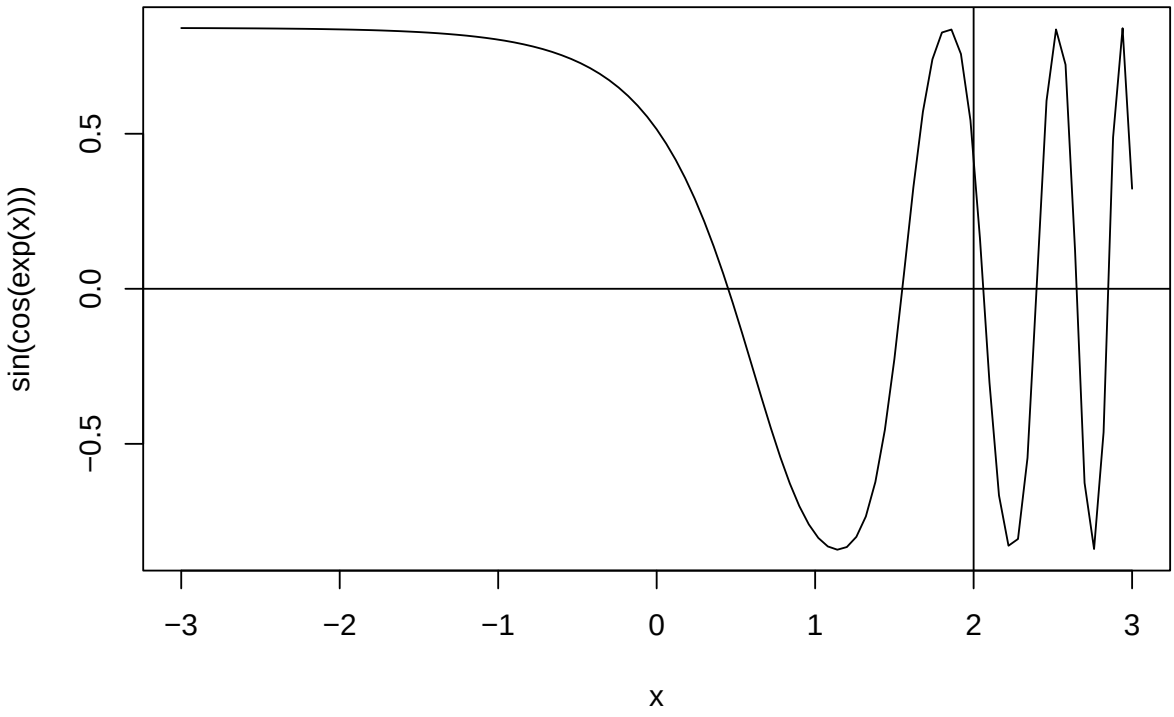
6.1.5 Exercise 05

Find the zeros of the function $f(x) = \sin(\cos(e^x))$ between 0 and 2, using both Newton-Raphson and the secant method. Is there any difference between the sets of numerical solutions found? With what method was convergence reached first? How can it be demonstrated? What is the drawback in using Newton-Raphson, rather than the secant method?

SOLUTION

Let us first draw the function to see how many intersections fall between 0 and 2.

```
curve(sin(cos(exp(x))),from=-3,to=3)

# An horizontal and vertical line to find out
# whether there's one zero before x=2
abline(h=0)
abline(v=2)
```



From the above plot we can see there are only two zeros between 0 and 1. They are roughly close to 0.5 and 1.5. To find the zeros using Newton-Raphson, we need the first derivative too. This is,

$$-e^x \sin(e^x) \cos(\cos(e^x))$$

```
# Function and its derivative
f0 <- function(x) return(sin(cos(exp(x))))
f1 <- function(x) return(-exp(x)*sin(exp(x))*cos(cos(exp(x))))

# First zero
x0 <- 0.5
xN1 <- roots_newton(f0,f1,x0)
#> The root is 0.451583. The error is less than 1.000000e-09.

# Second zero
x0 <- 1.5
xN2 <- roots_newton(f0,f1,x0)
#> The root is 1.550195. The error is less than 1.000000e-09.
```

These zeros can also be found using the secant method. In this case we need two starting points that contains the zeros. They could be, for instance, 0 and 1 for the first zero and 1 and 2 for the second zero.

```
# The function was defined in the previous code chunk

# First zero
x0L <- 0
x0R <- 1
```

```
xS1 <- roots_secant(f0,x0L,x0R)
#> The root is 0.451583. The error is less than 1.000000e-09.

# Second zero
x0L <- 1
x0R <- 2
xS2 <- roots_secant(f0,x0L,x0R)
#> The root is 9.379428. The error is less than 1.000000e-09.
```

This time, while the first value is correct, the second belongs to a different zero, not the one in the interval $[1, 2]$.

```
# xS2 is still a zero of f0 (within accuracy)
print(f0(xS2))
#> [1] -1.165234e-10
```

To find the zero wanted, the search interval must be restricted.

```
# Second zero
x0L <- 1.3
x0R <- 1.7
xS2 <- roots_secant(f0,x0L,x0R)
#> The root is 1.550195. The error is less than 1.000000e-09.
```

The second zero has now been found. To explore convergence we can, with both methods, use the `logg=TRUE` option in the respective R functions. We will restrict here the demonstration only to the first zero.

```
# Starting value/values
x0 <- 0.5
x0L <- 0
x0R <- 1

# Newton-Raphson method
xN1 <- roots_newton(f0,f1,x0,logg=TRUE)
#> The root is 0.451583. The error is less than 1.000000e-09.
#>        Root        Shift
#> 1 0.5000000           NA
#> 2 0.4525443 4.745567e-02
#> 3 0.4515832 9.611672e-04
#> 4 0.4515827 4.607524e-07

# Secant method
xS1 <- roots_secant(f0,x0L,x0R,logg=TRUE)
#> The root is 0.451583. The error is less than 1.000000e-09.
#>          x0        x1
#> 1 1.0000000 0.0000000
#> 2 0.0000000 0.3941841
#> 3 0.3941841 0.4748670
#> 4 0.4748670 0.4508753
#> 5 0.4508753 0.4515748
#> 6 0.4515748 0.4515827
#> 7 0.4515827 0.4515827
```

We can see that convergence is best achieved with Newton-Raphson, as expected, because the value within the accuracy desired is reached in 4, rather than 7, cycles. The drawback in using Newton-Raphson is, though, the need to provide (and therefore to calculate analytically) the first derivative of the function.

6.1.6 Exercise 06

When the zero of a function is known, one can track the behaviour of the errors as n becomes bigger and bigger. It is then possible to plot the natural logarithm of $|\epsilon_{n+1}|$ versus the natural logarithm of $|\epsilon_n|$. The resulting graph should produce points with regression lines passing through them, having slopes equal to the specific convergence order.

Using a specific nonlinear function, say

$$f(x) = x^3 + (1 - \sqrt{3})x^2 + (1 - \sqrt{3})x - \sqrt{3},$$

create the plots suggested and compare the regression straight lines with lines passing through the origin,

$$y = px,$$

where p is 1, $(1 + \sqrt{5})/2$, 2 for the bisection, secant and Newton-Raphson methods, respectively. It can be of help to know that the only real zero of this function is $\sqrt{3}$.

SOLUTION

Let us start with the bisection method. The procedure will be similar for the other methods. The `logg=TRUE` option will, in any case, provide the only feasible way to access subsequent approximations of the zeros. Different ways would involve changing parts of the function's code.

We need to define function and searching interval. Consider that the only real root of the cubic function used is $\sqrt{3} \approx 1.7$.

```
# Function
f0 <- function(x) {
  return(x^3+(1-sqrt(3))*x^2+(1-sqrt(3))*x-sqrt(3))
}

# Extremes of searching interval (root is sqrt(3))
x0L <- 1.5
x0R <- 2.0
```

Next, the function is executed with `logg=TRUE` and `message=FALSE`, because we do not need reading any specific result. As more decimals might be needed for precision, the option to increase the digits will be also used.

```
backup_options <- options()
options(digits=15)
roots_bisec(fn=f0,lB=x0L,rB=x0R,message=FALSE,logg=TRUE)
#>                Left             Right              Root            Difference
#> 1  1.50000000000000 2.00000000000000 1.75000000000000 5.00000000000000e-01
#> 2  1.50000000000000 1.75000000000000 1.62500000000000 2.50000000000000e-01
#> 3  1.62500000000000 1.75000000000000 1.68750000000000 1.25000000000000e-01
#> 4  1.68750000000000 1.75000000000000 1.71875000000000 6.25000000000000e-02
#> 5  1.71875000000000 1.75000000000000 1.73437500000000 3.12500000000000e-02
#> 6  1.71875000000000 1.73437500000000 1.72656250000000 1.56250000000000e-02
#> 7  1.72656250000000 1.73437500000000 1.73046875000000 7.81250000000000e-03
#> 8  1.73046875000000 1.73437500000000 1.73242187500000 3.90625000000000e-03
#> 9  1.73046875000000 1.73242187500000 1.73144531250000 1.95312500000000e-03
#> 10 1.73144531250000 1.73242187500000 1.73193359375000 9.76562500000000e-04
#> 11 1.73193359375000 1.73242187500000 1.73217773437500 4.88281250000000e-04
#> 12 1.73193359375000 1.73217773437500 1.73205566406250 2.44140625000000e-04
#> 13 1.73193359375000 1.73205566406250 1.73199462890625 1.22070312500000e-04
#> 14 1.73199462890625 1.73205566406250 1.73202514648438 6.10351562500000e-05
#> 15 1.73202514648438 1.73205566406250 1.73204040527344 3.05175781250000e-05
```

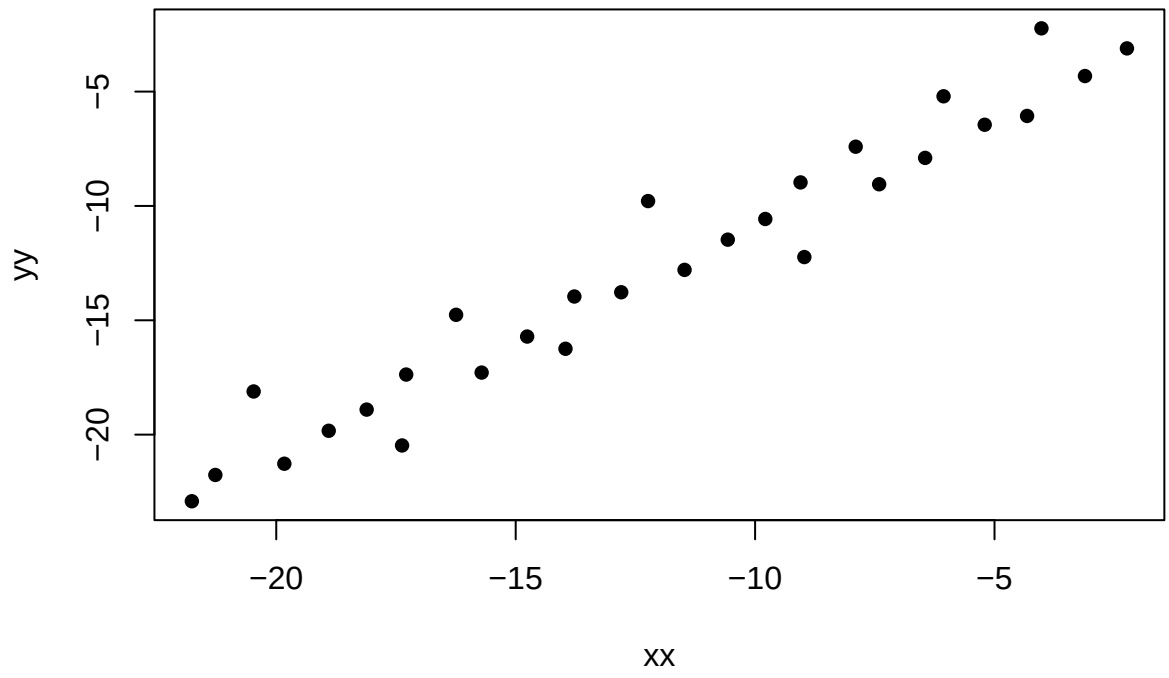
```
#> 16 1.73204040527344 1.73205566406250 1.73204803466797 1.52587890625000e-05
#> 17 1.73204803466797 1.73205566406250 1.73205184936523 7.62939453125000e-06
#> 18 1.73204803466797 1.73205184936523 1.73204994201660 3.81469726562500e-06
#> 19 1.73204994201660 1.73205184936523 1.73205089569092 1.90734863281250e-06
#> 20 1.73204994201660 1.73205089569092 1.73205041885376 9.53674316406250e-07
#> 21 1.73205041885376 1.73205089569092 1.73205065727234 4.76837158203125e-07
#> 22 1.73205065727234 1.73205089569092 1.73205077648163 2.38418579101562e-07
#> 23 1.73205077648163 1.73205089569092 1.73205083608627 1.19209289550781e-07
#> 24 1.73205077648163 1.73205083608627 1.73205080628395 5.96046447753906e-08
#> 25 1.73205080628395 1.73205083608627 1.73205082118511 2.98023223876953e-08
#> 26 1.73205080628395 1.73205082118511 1.73205081373453 1.4901161193847e-08
#> 27 1.73205080628395 1.73205081373453 1.73205081000924 7.45058059692383e-09
#> 28 1.73205080628395 1.73205081000924 1.73205080814660 3.72529029846191e-09
#> 29 1.73205080628395 1.73205080814660 1.73205080721527 1.86264514923096e-09
#> 30 1.73205080721527 1.73205080814660 1.73205080768093 9.31322574615479e-10
#> [1] 1.73205080721527
```

The third column of the table displayed is, next, manually input in a vector to be later used for the plot. From this, the vector of errors is readily computed when $\sqrt{3}$ is subtracted.

```
# Vector of approximated zeros
xr <- c(1.75000000000000,1.62500000000000,1.68750000000000,
        1.71875000000000,1.73437500000000,1.72656250000000,
        1.73046875000000,1.73242187500000,1.73144531250000,
        1.73193359375000,1.73217773437500,1.73205566406250,
        1.73199462890625,1.73202514648438,1.73204040527344,
        1.73204803466797,1.73205184936523,1.73204994201660,
        1.73205089569092,1.73205041885376,1.73205065727234,
        1.73205077648163,1.73205083608627,1.73205080628395,
        1.73205082118511,1.73205081373453,1.73205081000924,
        1.73205080814660,1.73205080721527,1.73205080768093)

# Errors
ers <- xr-sqrt(3)
```

There are 30 elements in ers, therefore we create a plot with 29 points. Remember that the plot is between $\log(|\epsilon_{n+1}|)$ and $\log(|\epsilon_n|)$.

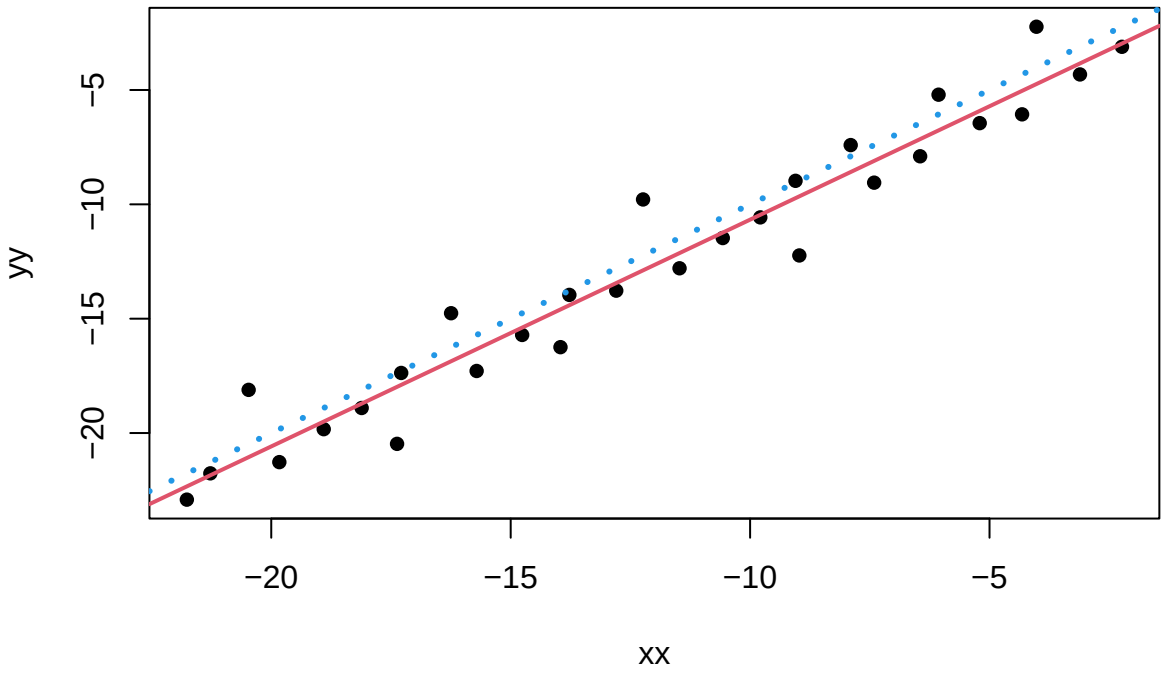
```
# x coordinates
xx <- log(abs(ers[1:29]))

# y coordinates
yy <- log(abs(ers[2:30]))

# Plot
plot(xx,yy,pch=16)
```



The slope of the regression line should be close to 1. We could check this using `lm` and printing `summary` of the output.

```
# Linear regression
model <- lm(yy ~ xx)
summary(model)
#>
#> Call:
#> lm(formula = yy ~ xx)
#>
#> Residuals:
#>              Min               1Q           Median               3Q              Max
#> -2.585629619011 -0.751323065980 -0.321008270102  0.521807708722  3.098578616426
#>
#> Coefficients:
#>                    Estimate       Std. Error  t value Pr(>|t|)
#> (Intercept) -0.7527376288887  0.6154285812035 -1.22311  0.23186
#> xx           0.9916323902036  0.0458492548389 21.62810  < 2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 1.43180123555 on 27 degrees of freedom
#> Multiple R-squared:  0.945429721419, Adjusted R-squared:  0.94340859999
#> F-statistic: 467.774824353 on 1 and 27 DF,  p-value: < 2.220446049e-16

# Plot with regression line and line y=x
plot(xx,yy,pch=16)
```

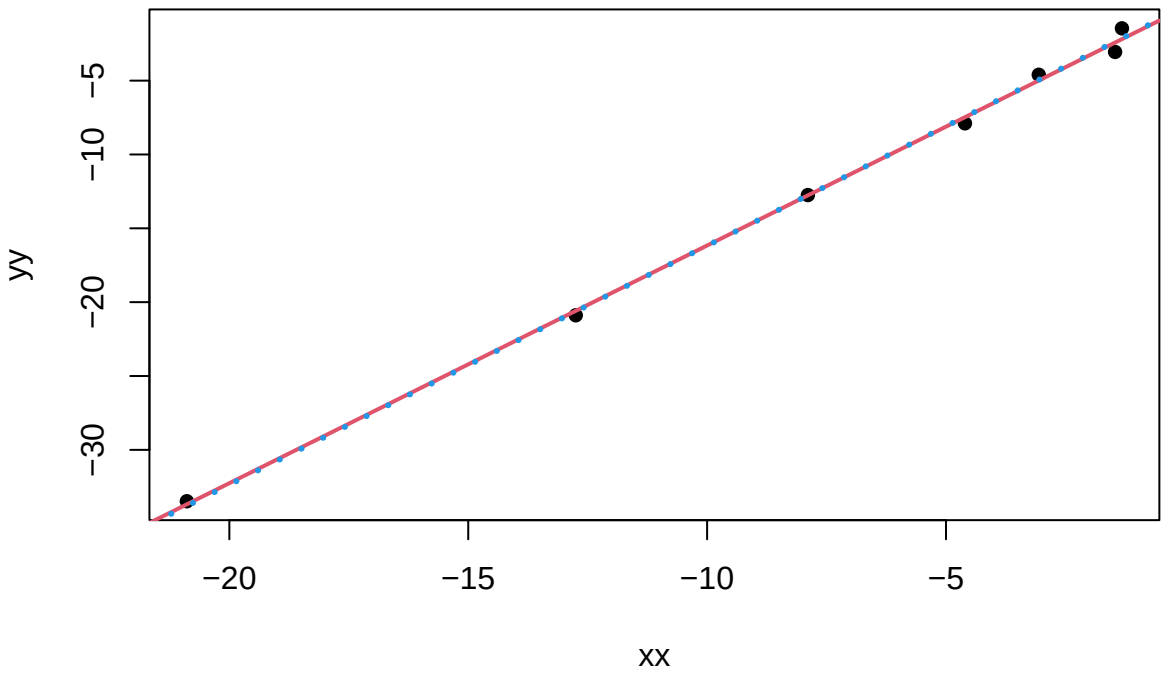
```
abline(model,lwd=2,col=2)
abline(a=0,b=1,lwd=3,lty=3,col=4)
```



The slope of the regression is close to 1 and the $y = x$ line passing through the origin is, indeed, parallel to the regression line. Let's now reproduce the same results for the secant method. Here convergence is reached only after 8 steps.

```
# Secant method
roots_secant(fn=f0,x0=x0L,x1=x0R,message=FALSE,logg=TRUE)
#>                  x0                x1
#> 1 2.00000000000000 1.50000000000000
#> 2 1.50000000000000 1.68507112987172
#> 3 1.68507112987172 1.74207066230820
#> 4 1.74207066230820 1.73167658855650
#> 5 1.73167658855650 1.73204790293642
#> 6 1.73204790293642 1.73205080841558
#> 7 1.73205080841558 1.73205080756888
#> [1] 1.73205080756888

# Prepare vector of approximating zeros
xr <- c(2.00000000000000,1.50000000000000,1.68507112987172,
        1.74207066230820,1.73167658855650,1.73204790293642,
        1.73205080841558,1.73205080756888)

# Errors
ers <- xr-sqrt(3)
```

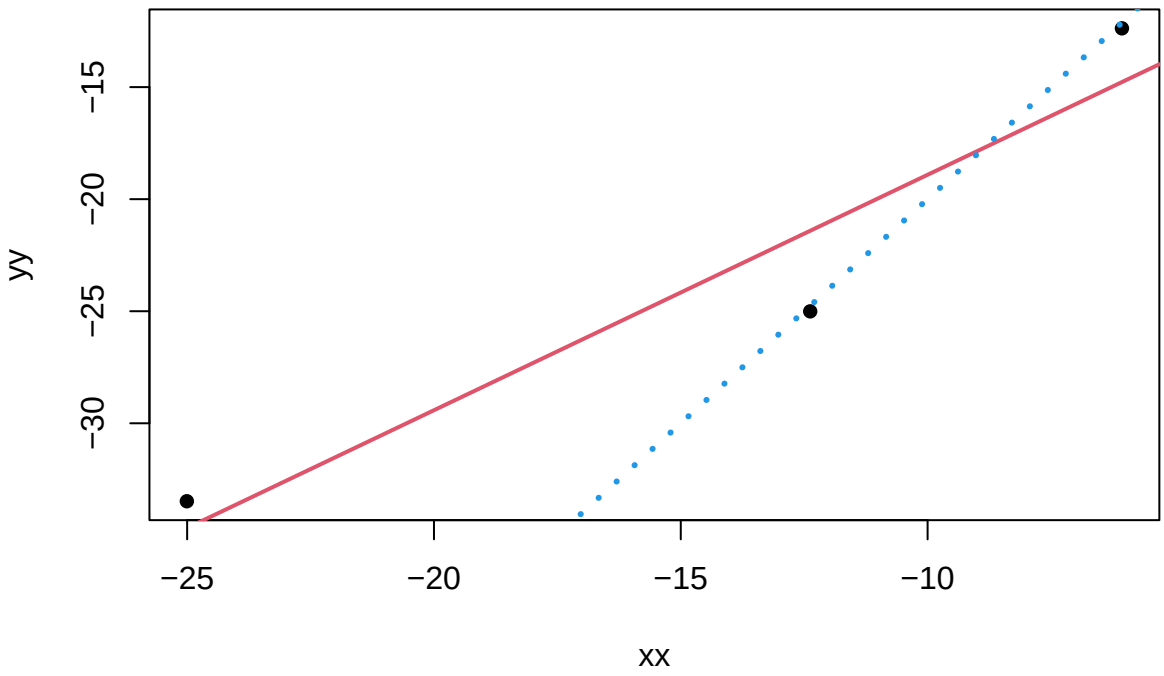
```r
# x coordinates
xx <- log(abs(ers[1:7]))

# y coordinates
yy <- log(abs(ers[2:8]))

# Plot
plot(xx,yy,pch=16)

# Linear regression
model <- lm(yy ~ xx)
summary(model)
#>
#> Call:
#> lm(formula = yy ~ xx)
#>
#> Residuals:
#>                1                2                3                4
#>  0.7302904498978 -0.6354830144256  0.3896373589342 -0.4114087598810
#>                5                6                7
#>  0.0202457975967 -0.3019101317502  0.2086282996281
#>
#> Coefficients:
#>                    Estimate       Std. Error  t value   Pr(>|t|)
#> (Intercept) -0.0718520221002  0.2989830969201 -0.24032    0.81962
#> xx           1.6091914296553  0.0299928832322 53.65244 4.2534e-08 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.527886849504 on 5 degrees of freedom
#> Multiple R-squared:  0.998266047021, Adjusted R-squared:  0.997919256425
#> F-statistic: 2878.58453839 on 1 and 5 DF,  p-value: 4.25344944473e-08

# Plot with regression line and line y=px
# p = (1+sqrt(5))/2
abline(model,lwd=2,col=2)
p <- (1+sqrt(5))/2
abline(a=0,b=p,lwd=3,lty=3,col=4)
```



In this case, too, the slope is close to what expected. For Newton-Raphson is worth decreasing the tolerance to avoid having too few points for the plot. We will use 10^{-18} as tolerance here. We also need the first derivative. And the starting point will be `x0L`. As seen from the chunk below, there are only siz points, which are hardly a trend of $n \to \infty$. It's better to at least eliminate the first two points as they introduce a strong bias.

```
# First derivative
f1 <- function(x) {
  return(3*x^2+2*(1-sqrt(3))*x+1-sqrt(3))
}

# Starting point
x0 <- x0L

# Secant method
roots_newton(f0=f0,f1=f1,x0=x0,tol=1e-18,message=FALSE,
             logg=TRUE)
#>               Root                Shift
#> 1 1.50000000000000                   NA
#> 2 1.78840919660718 2.88409196607183e-01
#> 3 1.73437871204450 5.40304845626869e-02
#> 4 1.73205501710228 2.32369494221452e-03
#> 5 1.73205080758268 4.20951960378169e-06
#> 6 1.73205080756888 1.38002942406956e-11
#> [1] 1.73205080756888

# Prepare vector of approximating zeros (only 4 values)
```

```r
xr <- c(1.73437871204450,1.73205501710228,1.73205080758268,
        1.73205080756888)

# Errors
ers <- xr-sqrt(3)

# x coordinates
xx <- log(abs(ers[1:3]))

# y coordinates
yy <- log(abs(ers[2:4]))

# Plot
plot(xx,yy,pch=16)

# Linear regression
model <- lm(yy ~ xx)
summary(model)
#>
#> Call:
#> lm(formula = yy ~ xx)
#>
#> Residuals:
#>              1              2              3
#>  2.39707288000 -3.59587017795  1.19879729795
#>
#> Coefficients:
#>                    Estimate      Std. Error  t value Pr(>|t|)
#> (Intercept) -8.405550099365  5.420024704886 -1.55083  0.36461
#> xx           1.050619419086  0.328781557087  3.19549  0.19308
#>
#> Residual standard error: 4.48479159942 on 1 degrees of freedom
#> Multiple R-squared:  0.910803326832, Adjusted R-squared:  0.821606653664
#> F-statistic: 10.2111804676 on 1 and 1 DF,  p-value: 0.193078050825

# Plot with regression line and line y=px
# p = 2
abline(model,lwd=2,col=2)
p <- 2
abline(a=0,b=p,lwd=3,lty=3,col=4)
```



This time the regression does not give a value close to the expected slope, $p = 2$. This is certainly due to too few points before convergence is reached. It is, though, a good sign to observe that the correct $y = 2x$ line passes through the last two points, showing that the later trend for convergence seems to be the theoretical one.

6.2 Exercises on the roots of systems of nonlinear equations

6.2.1 Exercise 07

Consider the following system of nonlinear equations:

$$\begin{cases} x^2 + y^2 &= 4 \\ e^x + y &= 1. \end{cases}$$

Use a plot of the two curves represented by the above system to select starting points to find the solutions to the system. Use both `nleqslv` and `multiroot` for the task.

SOLUTION

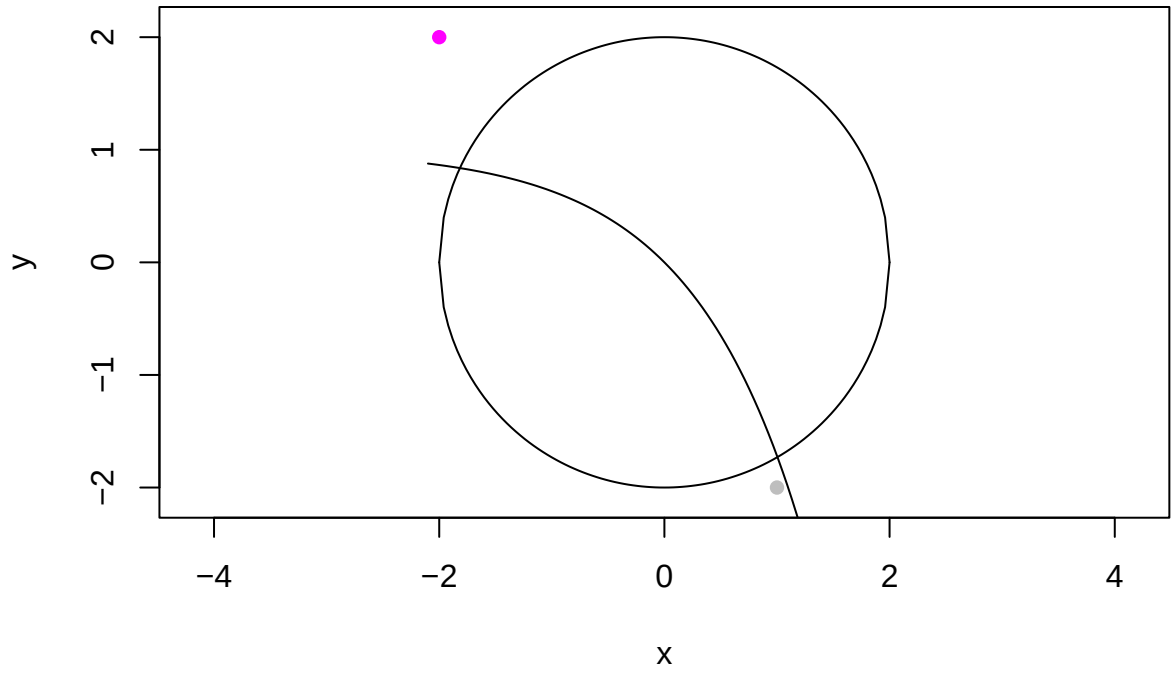
A plot of the two curves

$$x^2 + y^2 = 4, \qquad y = 1 - e^x,$$

should display the rough location of all the intersections.

```
# Empty plot
plot(NA,NA,xlim=c(-2.2,2.2),ylim=c(-2.1,2.1),
     xlab=expression(x),ylab=expression(y),asp=1)

# First curve (two halves)
```

```r
curve(sqrt(4-x^2),from=-2,to=2,ylim=c(-2.1,2.1),add=TRUE)
curve(-sqrt(4-x^2),from=-2,to=2,add=TRUE)

# Second curve
curve(1-exp(x),from=-2.1,to=2.1,add=TRUE)

# Points close to intersections
points(-2,2,pch=16,col="magenta")
points(1,-2,pch=16,col="grey")
```



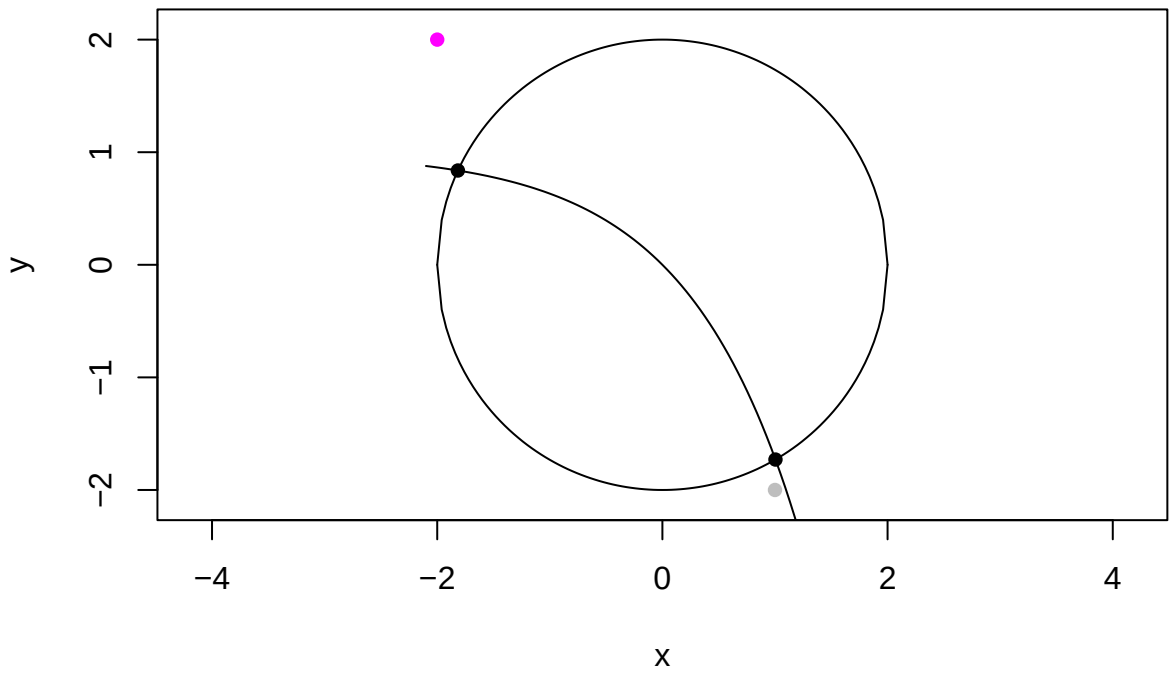
Two possible points to start the search for the intersections are $(-2, 1)$, coloured in magenta, and $(1, -2)$, coloured in grey. The search with both R functions is done here.

```r
# Define system:x[1]=x, x[2]=y
f <- function(x) {
  f1 <- x[1]^2+x[2]^2-4
  f2 <- exp(x[1])+x[2]-1

  return(c(f1,f2))
}

# Top starting point
xs1 <- c(-2,1)

# Search (load library)
require(nleqslv)
```

```
#> Loading required package: nleqslv
res1 <- nleqslv(x=xs1,fn=f)
print(res1$message)
#> [1] "Function criterion near zero"
print(res1$x)
#> [1] -1.816264069388648  0.837367799757065

# Bottom starting point
xs2 <- c(1,-2)

# Search
res2 <- nleqslv(x=xs2,fn=f)
print(res2$message)
#> [1] "Function criterion near zero"
print(res2$x)
#> [1]  1.00416873845203 -1.72963728698423
```

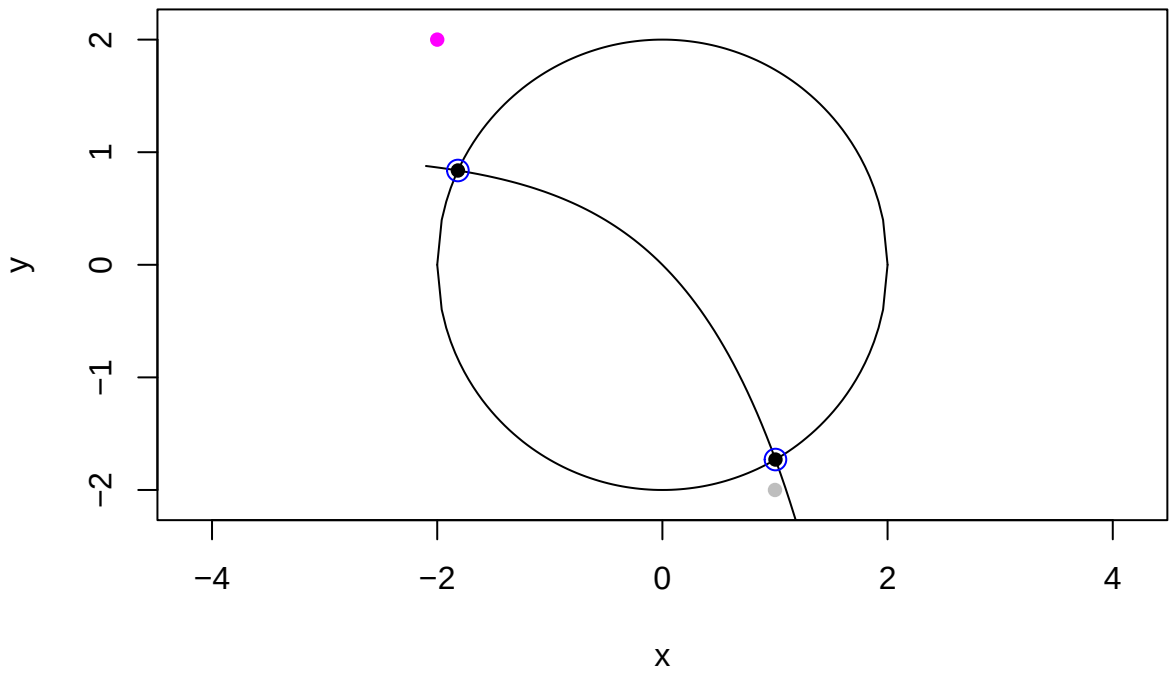
We can also plot the points found on the curves' plot.

```
# Empty plot
plot(NA,NA,xlim=c(-2.2,2.2),ylim=c(-2.1,2.1),
     xlab=expression(x),ylab=expression(y),asp=1)

# First curve (two halves)
curve(sqrt(4-x^2),from=-2,to=2,ylim=c(-2.1,2.1),add=TRUE)
curve(-sqrt(4-x^2),from=-2,to=2,add=TRUE)

# Second curve
curve(1-exp(x),from=-2.1,to=2.1,add=TRUE)

# Points close to intersections
points(-2,2,pch=16,col="magenta")
points(1,-2,pch=16,col="grey")

# Points corresponding to solutions
points(res1$x[1],res1$x[2],pch=16,col="black")
points(res2$x[1],res2$x[2],pch=16,col="black")
```



We can carry out a similar task with `multiroot`.

```
# Search (load library)
require(rootSolve)
#> Loading required package: rootSolve
resA <- multiroot(f=f,start=xs1)
resB <- multiroot(f=f,start=xs2)
print(resA$root)
#> [1] -1.816264069308840  0.837367800109253
print(resB$root)
#> [1]  1.00416873942263 -1.729637287778070

# Plot
plot(NA,NA,xlim=c(-2.2,2.2),ylim=c(-2.1,2.1),
     xlab=expression(x),ylab=expression(y),asp=1)
curve(sqrt(4-x^2),from=-2,to=2,ylim=c(-2.1,2.1),add=TRUE)
curve(-sqrt(4-x^2),from=-2,to=2,add=TRUE)
curve(1-exp(x),from=-2.1,to=2.1,add=TRUE)
points(-2,2,pch=16,col="magenta")
points(1,-2,pch=16,col="grey")
points(resA$root[1],resA$root[2],pch=1,cex=1.5,col="blue")
points(resB$root[1],resB$root[2],pch=1,cex=1.5,col="blue")

# Comparison with nleqslv
points(res1$x[1],res1$x[2],pch=16,col="black")
points(res2$x[1],res2$x[2],pch=16,col="black")
```



So, the two methods give solutions which are very close to each other.

6.2.2 Exercise 08

Solve the system

$$\begin{cases} x^2+y^2+z^2 &= 9 \\ xyz &= 1 \\ x+y-z^2 &= 0 \end{cases}$$

using Newton's method to obtain the solution near $(2.4, 0.2, 1.7)$.

SOLUTION

The *pure* Newton method can be applied using `nleqslv`, selecting the method and imposing no global strategy.

```
# Define the system x[1]=x, x[2]=y, x[3]=z
f <- function(x) {
  f1 <- x[1]^2+x[2]^2+x[3]^2-9
  f2 <- x[1]*x[2]*x[3]-1
  f3 <- x[1]+x[2]-x[3]^2

  return(c(f1,f2,f3))
}

# Starting point
xstart <- c(2.4,0.2,1.7)

# Search
```

```
res <- nleqslv(x=xstart,fn=f,method="Newton",global="none")

# Results
print(res$message)
#> [1] "Function criterion near zero"
print(res$x)
#> [1] 2.491375696830764 0.2427458787757069 1.653517939300366
print(res$fvec)
#> [1]  6.46593889541691e-13 -1.92956761679852e-13 -2.94875235340442e-13
```

`nleqslv` seems to have had no problem whatsoever finding the solution with *pure* Newton, without global strategy. Would that be the case starting from further apart?

```
# Distant starting point
xstart <- c(-1,0,1)

# Search
res <- nleqslv(x=xstart,fn=f,method="Newton",global="none")

# Results
print(res$message)
#> [1] "Function criterion near zero"
print(res$x)
#> [1] 0.2427458787757383 2.491375696830732 1.653517939300436
print(res$fvec)
#> [1]  8.70414851306123e-13  1.12954090525363e-12 -2.44693154627385e-13
```

Yes, it seems that the nonlinear system is not pathological. It is in general worth trying out a few different starting solutions. It can, in fact, happen that for some starting solutions the Jacobian is not well defined, as in the following case.

```
# Different starting point
xstart <- c(0,0,0)

# Search
res <- nleqslv(x=xstart,fn=f,method="Newton",global="none")

# Results (Jacobian not good)
print(res$message)
#> [1] "Jacobian is singular (1/condition=0.0e+00) (see allowSingular option)"
print(res$termcd)
#> [1] 6
print(res$x)
#> [1] 0 0 0
print(res$fvec)
#> [1] -9 -1  0
```

So, even when a numeric solution is return by the algorithm, this does not mean that it is an actual solution of the system. In code triggered by the result of `nleqslv`, it can be useful to use the returned integer `termcd` to decide what to do next, in case it is not equal to 1, the only number signalling correct convergence.

6.2.3 Exercise 09

Given the system

$$\begin{cases} xyz - x^2 + y^2 &= 1.33 \\ xy - z^3 &= 0.1 \\ e^x - e^y + z &= 0.41, \end{cases}$$

find as many of its solutions as possible, in the range

$$-0.5 \leq x \leq 0.5, \quad -1 \leq y \leq 1, \quad -1 \leq z \leq 1.$$

SOLUTION

A possible way of finding the solutions is to carry out a coarse search prior to use `nleqslv` or `multiroot`. We know, indeed, that these functions have better chances of success when the starting point for the iterations is close to the actual zero of the function. The idea is then to create a coars grid of values in the range provided and to measure the distance between the left and right hand sides of the three nonlinear equations composing the system. A good step for the grid is 0.1.

```
# Search grid
G <- expand.grid(x=seq(-0.5,0.5,by=0.1),
                 y=seq(-1,1,by=0.1),
                 z=seq(-1,1,by=0.1))
G <- as.matrix(G)  # For speed
print(G[1:10,])
#>           x  y  z
#>  [1,] -0.5 -1 -1
#>  [2,] -0.4 -1 -1
#>  [3,] -0.3 -1 -1
#>  [4,] -0.2 -1 -1
#>  [5,] -0.1 -1 -1
#>  [6,]  0.0 -1 -1
#>  [7,]  0.1 -1 -1
#>  [8,]  0.2 -1 -1
#>  [9,]  0.3 -1 -1
#> [10,]  0.4 -1 -1
```

Then we define the three $f_i(x, y, z)$, starting from the system:

```
# Define the functions from which roots are extracted
f <- function(x) {
  f1 <- x[1]*x[2]*x[3]-x[1]^2+x[2]^2-1.33
  f2 <- x[1]*x[2]-x[3]^3-0.1
  f3 <- exp(x[1])-exp(x[2])+x[3]-0.41

  return(c(f1,f2,f3))
}
```

Finally, we calculate the Euclidean norm of $\mathbf{f}(x, y, z)$ at all points of the grid. Candidate points for the search are those with small values of the Euclidean norm.

```
# Fast way to calculate norms
norms <- apply(G,1,function(p) {sqrt(sum(f(p)^2))})

# Add column to matrix G
G <- cbind(G,norms)
print(G[1:10,])
#>           x  y  z             norms
```

```
#>  [1,] -0.5 -1 -1 2.120956851947968
#>  [2,] -0.4 -1 -1 1.925821334847442
#>  [3,] -0.3 -1 -1 1.741808248643816
#>  [4,] -0.2 -1 -1 1.566801265595419
#>  [5,] -0.1 -1 -1 1.398500044390595
#>  [6,]  0.0 -1 -1 1.234502501008886
#>  [7,]  0.1 -1 -1 1.072444290882153
#>  [8,]  0.2 -1 -1 0.910256172039074
#>  [9,]  0.3 -1 -1 0.746727301485248
#> [10,]  0.4 -1 -1 0.583032860391344

# List smaller values of norm
idx <- order(G[,4])
print(G[idx[1:10],])
#>         x  y    z             norms
#>  [1,] 0.5 -1 -0.9 0.185448642230765
#>  [2,] 0.4 -1 -0.8 0.190917309021586
#>  [3,] 0.5 -1 -0.8 0.212514857859330
#>  [4,] 0.3 -1 -0.7 0.252464418534925
#>  [5,] 0.4 -1 -0.7 0.262570886006064
#>  [6,] 0.3 -1 -0.6 0.303712291333574
#>  [7,] 0.2 -1 -0.6 0.306660972942776
#>  [8,] 0.3 -1 -0.8 0.311347730592766
#>  [9,] 0.4 -1 -0.9 0.322424204411018
#> [10,] 0.2 -1 -0.5 0.326672030825958
```

The nonlinear solver `nleqslv` can now be tried on just a few grid points at the top of the sorted list.

```
# Try solutions oly on top 20 grid points (lowest norm)
Fres <- c()  # Final set of solutions
for (i in idx[1:20]) {
  xstart <- G[i,1:3]
  res <- nleqslv(x=xstart,fn=f)
  print(res$message)
  print(res$x)
  Fres <- rbind(Fres,res$x)
}
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#>  0.468799765885295 -1.062955245235229 -0.842641514772478
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#>  0.468799766339860 -1.062955246223213 -0.842641520142203
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#>  0.468799767834220 -1.062955244454144 -0.842641520233479
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#>  0.468799767696128 -1.062955246488206 -0.842641523834438
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#>  0.468799766059173 -1.062955246572065 -0.842641519340518
#> [1] "Function criterion near zero"
#>                  x                  y                  z
```

```
#>  0.4687997688818373 -1.062955243618921 -0.842641522168084
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#>  0.468799767442123 -1.062955245381050 -0.842641520696309
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#>  0.468799767547277 -1.062955245089863 -0.842641520404916
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#>  0.468799767169866 -1.062955245558578 -0.842641519835576
#> [1] "Function criterion near zero"
#>                   x                   y                   z
#> -0.0788410491163221 -1.1639357322566961 -0.2019320064380295
#> [1] "Function criterion near zero"
#>                   x                   y                   z
#> -0.0788410482839457 -1.1639357308336011 -0.2019320055113270
#> [1] "Function criterion near zero"
#>                   x                   y                   z
#> -0.0788410493212581 -1.1639357291848549 -0.2019320029896711
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#> -0.078841050899976 -1.163935735627784 -0.201932008032926
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#> -0.078841047743583 -1.163935730865419 -0.201932005943954
#> [1] "Function criterion near zero"
#>                  x                  y                  z
#>  0.468799767327967 -1.062955245217759 -0.842641520029501
#> [1] "Function criterion near zero"
#>                   x                   y                   z
#> -0.0788410475965267 -1.1639357321341743 -0.2019320074093170
#> [1] "Function criterion near zero"
#>                   x                   y                   z
#> -0.0788410488982134 -1.1639357309737624 -0.2019320048426080
#> [1] "Function criterion near zero"
#>                   x                   y                   z
#> -0.0788410485823711 -1.1639357304296920 -0.2019320049265862
#> [1] "Function criterion near zero"
#>                   x                   y                   z
#> -0.0788410475320107 -1.1639357265397210 -0.2019320025401986
#> [1] "Function criterion near zero"
#>                   x                   y                   z
#> -0.0788410460759864 -1.1639357298619257 -0.2019320066537222
```

There seem to be two solutions. To single them out automatically, we could carry out first a rounding (say to the 6th decimal place), and then use function `duplicated` in a clever way.

```
# Rounding to eliminate "rounding noise"
rounded_x <- round(Fres,digits=6)

# Selects the first of a series of identical sets
unique_rows <- !duplicated(rounded_x)

# Only the first (unique) element is kept.
```

```
# The 'drop=FALSE' means do not transform the matrix into
# a non-matrix, even if only one element is kept
filtered <- rounded_x[unique_rows, ,drop=FALSE]
print(filtered)
#>              x         y         z
#> [1,]  0.468800 -1.062955 -0.842642
#> [2,] -0.078841 -1.163936 -0.201932
```

So, there seem to be only two solutions. In fact, this procedure can be extended to all the points in the grid, without selecting those with lowest norm. The output can be selected using the parameter `termcd`, which is 1 when convergence is achieved.

```
# Apply the algorithm to all points of the grid
Fres <- c()
for (i in 1:length(G[,1])) {
  xstart <- G[i,1:3]
  res <- nleqslv(x=xstart,fn=f)
  if (res$termcd == 1) {
    Fres <- rbind(Fres,res$x)
  }
}

# Unique solutions
rounded_x <- round(Fres,digits=6)
unique_rows <- !duplicated(rounded_x)
filtered <- rounded_x[unique_rows, ,drop=FALSE]
print(filtered)
#>              x         y         z
#> [1,] -0.078841 -1.163936 -0.201932
#> [2,]  0.468800 -1.062955 -0.842642
#> [3,]  0.890274  1.092378  0.955560
```

So, in the assigned region, there exist three solutions to the given system. We can check these are solutions by calculating $\mathbf{f}(\mathbf{x})$ at these three points.

```
print(f(filtered[1,]))
#>                         x                         x                         x
#>  6.41503163922863e-07 -3.50381704344871e-08  1.34103133209162e-07
print(f(filtered[2,]))
#>                         x                         x                         x
#> -3.88865831757457e-07  8.90054873275981e-07 -1.92831431433671e-07
print(f(filtered[3,]))
#>                         x                         x                         x
#> -9.67731059509092e-07 -1.24406761584095e-06  1.65566755677693e-06
```

6.2.4 Exercise 10

Solve the nonlinear system

$$\begin{cases} f_1(x,y) = x^2 + y^2 - 1 \\ f_2(x,y) = \exp(x) - y \end{cases}$$

using the `nleqslv` function in R. This system has a solution near $(x, y) \approx (0.0, 1.0)$, where the unit circle intersects the exponential curve. Use the starting point $(x_0, y_0) = (0.5, 0.5)$ and the default method and global strategy. Then:

1. Fix `ftol` but vary `xtol` in the following way:

- Set `ftol = 1e-8`.
- Try `xtol = 1e-1, 1e-4, 1e-8, 1e-12`.
- For each run, record: the final result `res\$x`, the number of iterations `res\$iter`, the termination code `res\$termcd`, the norm of the residual vector $\|\mathbf{f}(\mathbf{x})\|$.

2. Fix `xtol` but vary `ftol` in the following way:

- Set `xtol = 1e-8`.
- Try `ftol = 1e-1, 1e-4, 1e-8, 1e-12`.
- Record the same outputs as in Step 1.

For large `xtol`, does the solver stop before the function values are small? For large `ftol`, does the solver stop even if the update step is still large? When both `xtol` and `ftol` are tight, does the result improve? Do more iterations occur? Based on the termination codes, which stopping condition triggered the exit in each case?

This exercise shows how `xtol` controls the size of the update step, while `ftol` governs how close the function values must be to zero. Either stopping condition can dominate, depending on how the tolerances are set. The exercise also illustrates the trade-off between accuracy and computational effort.

SOLUTION

The first part of the exercise is straightforward.

```
# Define system
f <- function(x) {
  f1 <- x[1]^2+x[2]^2-1
  f2 <- exp(x[1])-x[2]

  return(c(f1,f2))
}

# Define norm for use after each run
Normf <- function(ff) {
  nrm <- sqrt(ff[1]^2+ff[2]^2)

  return(nrm)
}

# Starting point
xstart <- c(0.5,0.5)

# Solve with default values
res <- nleqslv(x=xstart,fn=f)

# Prints
dtmp <- data.frame(x=res$x[1],y=res$x[2],
                   iter=res$iter,code=res$termcd,
                   norm=Normf(res$x))
print(dtmp)
#>                         x                y iter code              norm
#> 1 -1.23609422642487e-11 0.999999999993082    8    1 0.999999999993082
```

Next, we keep `ftol` fixed and vary `xtol`.

```
# Run with fixed ftol
dtmp <- data.frame()
```

```
for (xt in c(1e-1,1e-4,1e-8,1e-12)) {
  res <- nleqslv(x=xstart,fn=f,control=list(ftol=1e-8,xtol=xt))
  dtmp <- rbind(dtmp,data.frame(xtol=xt,x=res$x[1],y=res$x[2],
                   iter=res$iter,code=res$termcd,
                   norm=Normf(res$x)))
}
print(dtmp)
#>    xtol                       x                 y iter code              norm
#> 1 1e-01  3.54378449196690e-02 1.014230494269306    3    2 1.014849415607218
#> 2 1e-04  2.99971503714161e-08 1.000000016808381    7    2 1.000000016808382
#> 3 1e-08 -1.23609422642487e-11 0.999999999993082    8    1 0.999999999993082
#> 4 1e-12 -1.23609422642487e-11 0.999999999993082    8    1 0.999999999993082
```

Now, we keep `xtol` fixed and vary `ftol`.

```
# Run with fixed xtol
dtmp <- data.frame()
for (ft in c(1e-1,1e-4,1e-8,1e-12)) {
  res <- nleqslv(x=xstart,fn=f,control=list(ftol=ft,xtol=1e-8))
  dtmp <- rbind(dtmp,data.frame(ftol=ft,x=res$x[1],y=res$x[2],
                   iter=res$iter,code=res$termcd,
                   norm=Normf(res$x)))
}
print(dtmp)
#>    ftol                       x                 y iter code              norm
#> 1 1e-01  6.94581536831068e-02 0.972855348655671    2    1 0.975331720247533
#> 2 1e-04  7.93872051176861e-06 1.000004484096160    6    1 1.000004484127671
#> 3 1e-08 -1.23609422642487e-11 0.999999999993082    8    1 0.999999999993082
#> 4 1e-12  7.31574168681706e-16 1.000000000000000    9    1 1.000000000000000
```

The above results can be framed properly while answering the questions posed in the exercise.

For large `xtol`, does the solver stop before the function values are small?

This could be re-phrased: *Does large `xtol` cause early termination?* The answer is **yes**. For example, with `xtol=1e-1`, the solver stopped after 3 iterations and a residual norm of 1.01. This is well above an acceptable tolerance, indicating the step-size criterion was met before the function values became small. The termination code 2 confirms this.

For large `ftol`, does the solver stop even if the update step is still large?

This could be re-phrased: *Does large `ftol` cause acceptance of inaccurate solutions?* The answer is **yes**. With `ftol=1e-1` and `xtol=1e-8`, the solver stopped after just 2 iterations with a norm around 0.975, even though the step size could still be improved. This demonstrates that `ftol` governs how small the function values must be.

When both `xtol` and `ftol` are tight, does the result improve? Do more iterations occur?

This could be re-phrased: *Do tighter tolerances improve results?* The answer is **yes**. When both `xtol` and `ftol` are small (e.g., `1e-8` or `1e-12`), the solver performs more iterations but returns a more accurate solution. The residual norm improves to within machine precision.

Based on the termination codes, which stopping condition triggered the exit in each case?

This could be re-phrased: *Which stopping condition dominated?* For large `xtol`, termination was by step size (code = 2). For large `ftol`, termination was by function value (code = 1). When both are tight, the function value convergence (code = 1) is the dominant stopping criterion.

Ultimately, this analysis demonstrates the importance of setting both `xtol` and `ftol` appropriately, depending on the required accuracy of the solution.

7 Chapter 07

7.1 Exercises on differentiation

7.1.1 Exercise 01

Consider the simple function $f(x) = x + e^x + sin(x)$. Calculate the first derivative at $x = 0$ using forward, backward and centred differences. Use $h = 0.1$ and $h = 0.01$. Verify that the numerical errors are $O(h)$ and $O(h^2)$.

SOLUTION

The first derivative can ba calculated analytically and it is given by

$$f'(x) = 1 + e^x + \cos(x).$$

At $x = 0$ we have $f'(0) = 3$, exactly.

The `comphy` function to calculate fotward, backward, and centred difference derivatives is `deriv_reg()`. The necessary input are values at a regular grid and the corresponding values tabulated for the function. So, we will need to generate the regular grid first, using the R function `seq`. We can do that via the parameter `by`. As the derivative must be calculated at $x = 0$, we should create the grid so to contain that value. Also, as only one value is required, we don't need to use a large grid (a grid with many values); in this case, a 5-values grid will suffice. The grid will contain points

$$-0.2, -0.1, 0, 0.1, 0.2$$

when $h = 0.1$, and

$$-0.02, -0.01, 0, 0.01, 0.02$$

when $h = 0.01$. We can also use the parameter `length.out=5` in `seq`, to decide the grid size.

```
# Create the grid around x=0 in two ways, adding the final point or without it
# Adding the final point
x1 <- seq(-0.2,0.2,by=0.1) # Grid when h=0.1
x2 <- seq(-0.02,0.02,by=0.01) # Grid when h=0.01
print(x1)
#> [1] -0.2 -0.1  0.0  0.1  0.2
print(x2)
#> [1] -0.02 -0.01  0.00  0.01  0.02
# Without final point
x1 <- seq(-0.2,by=0.1,length.out=5)
x2 <- seq(-0.02,by=0.01,length.out=5)
print(x1)
#> [1] -0.2 -0.1  0.0  0.1  0.2
print(x2)
#> [1] -0.02 -0.01  0.00  0.01  0.02
```

Once the grid is created, the function's tabulated points are also easily created, using the grid's variable as the x in the analytic expression of the function.

```
# Create tabulated points
f1 <- x1 + exp(x1) + sin(x1)
f2 <- x2 + exp(x2) + sin(x2)
```

For this specific exercise, the derivative is required only at $x = 0$. The variable `x0` needed for the `comphy` function to work must include one or more exact grid points of `x`. Here the only point needed is $x = 0$, which happens to be the third grid point of both grid `x1` and `x2`. The derivatives are then easily calculated with `deriv_reg()`, and subsequently printed for display, using formatting to appreciate the accuracy of the numbers calculated.

```
# Only one point is needed for the derivative.
# This must be, exactly, one of the grid points, or the function stops
# h=0.1
x0 <- x1[3]
f1derC <- deriv_reg(x0,x1,f1) # Default scheme is with centred difference
f1derF <- deriv_reg(x0,x1,f1,scheme="f") # Forward difference
f1derB <- deriv_reg(x0,x1,f1,scheme="b") # Backward difference
sprintf("%10.7f, %10.7f, %10.7f",f1derC,f1derF,f1derB)
#> [1] " 3.0000017,  3.0500433,  2.9499600"
# h=0.01
x0 <- x2[3]
f1derC <- deriv_reg(x0,x2,f2) # Default scheme is with centred difference
f1derF <- deriv_reg(x0,x2,f2,scheme="f") # Forward difference
f1derB <- deriv_reg(x0,x2,f2,scheme="b") # Backward difference
sprintf("%10.7f, %10.7f, %10.7f",f1derC,f1derF,f1derB)
#> [1] " 3.0000000,  3.0050000,  2.9950000"
```

The errors are indeed $\mathrm{O}(h)$ or better for $h = 0.1$ and $h = 0.01$ when forward and backward differences are used. When $h = 0.1$ we have

$$f'_{\text{true}}(0) - f'_F(0) = 3 - 3.0500433 = -0.0500433, \qquad f'_{\text{true}}(0) - f'_B(0) = 3 - 2.9499600 = 0.0500400$$

and, when $h = 0.01$, we have

$$f'_{\text{true}}(0) - f'_F(0) = 3 - 3.0050000 = -0.0050000, \qquad f'_{\text{true}}(0) - f'_B(0) = 3 - 2.9950000 = 0.0050000.$$

The accuracy is higher when centred differences are used. For $h = 0.1$ we should have $\mathrm{O}(h^2) = \mathrm{O}(0.01)$. Indeed:

$$f'_{\text{true}}(0) - f'_C(0) = 3 - 3.0000017 = -0.000017,$$

which means that the accuracy is better in this case. And for $h = 0.01$ we should have an accuracy equal to $\mathrm{O}(0.0001)$ which, to the significant figured shown, is certainly true.

7.1.2 Exercise 02

Use the function $f(x) = e^x$ around $x = 1$, and values of h between 0.001 and 0.1 to show that the error goes like $\mathrm{O}(h^2)$.

SOLUTION

The grid can, as in the previous exercise, consist of just 5 grid points centred at $x = 1$. Then we apply `deriv_reg()` for as many values as h as it is necessary. In the exercise it is not explained how many values of h to use, but we can very simply create a regular grid of h between 0.001 and 0.1, with step 0.001.

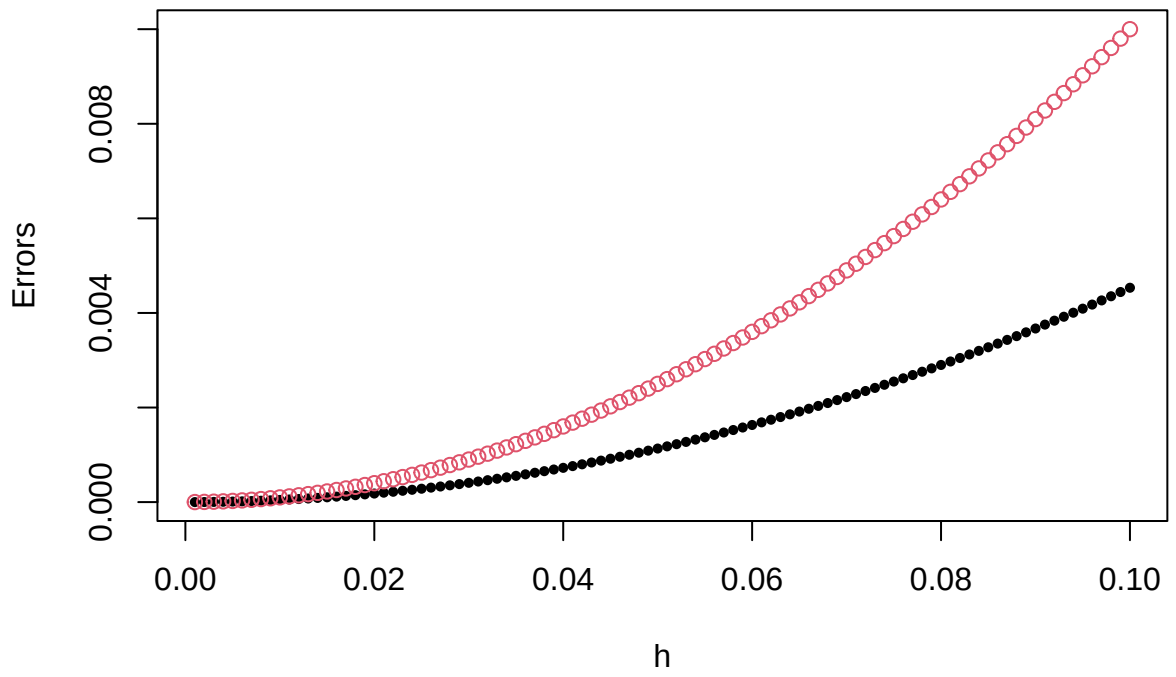
In the code here presented, a loop over the values of h is carried out and values of the centred difference derivative are stored.

```
## Exercise on the derivative of f(x)=exp(x) at x=1

# Grid of values for h
hh <- seq(0.001,0.1,by=0.001)

# Loop over all values of h.
```

```r
# At each loop we need to redefine the grid and tabulate f(x)
DfT <- exp(1) # True derivative
DfC <- c()
for (h in hh) {
  x <- seq(1-2*h,by=h,length.out=5)
  f <- exp(x)
  tmp <- deriv_reg(x[3],x,f)
  DfC <- c(DfC,tmp)
}
```

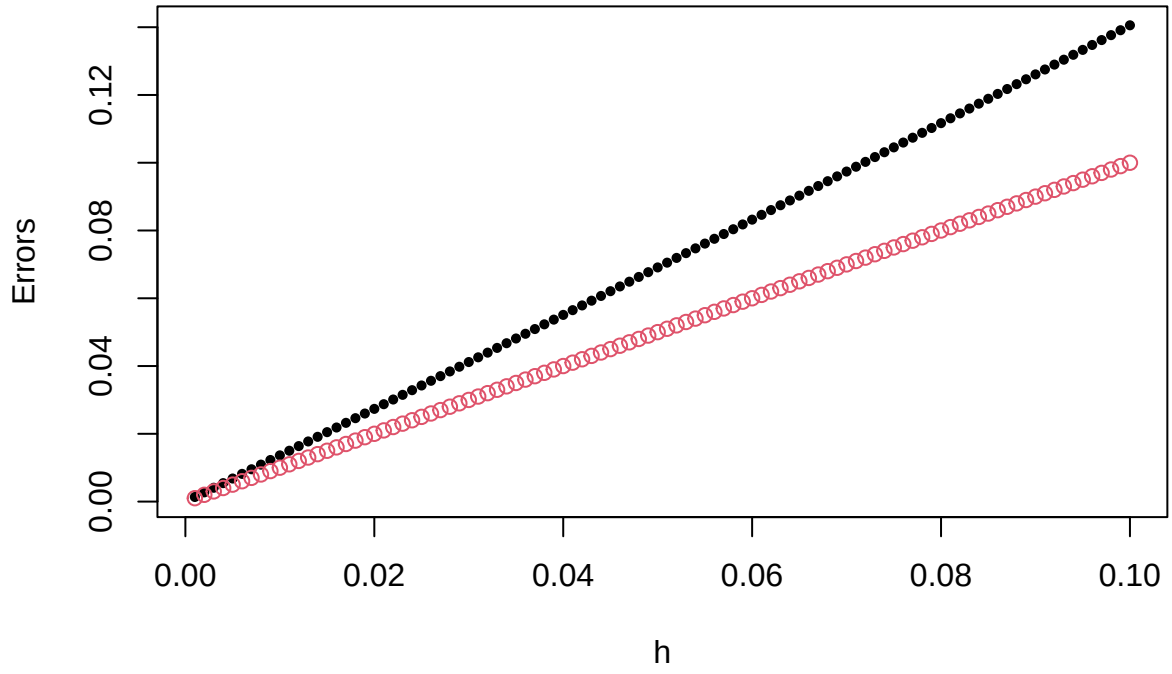
The errors for all values of h can be plotted and compared to a quadratic curve (because of the dependency on h^2). As we are comparing with h^2, which is a non negative function, we will use the absolute value of the errors.

```r
# Errors (the true derivative is exp(1))
Deltas <- abs(DfT - DfC)

# Function h^2 compared graphically to the numerical derivative
ff <- hh^2
ylim <- range(ff,Deltas)
plot(hh,Deltas,type="b",pch=16,cex=0.7,ylim=ylim,xlab="h",ylab="Errors")
points(hh,ff,type="b",col=2)
```



The two curves do not overlap but have a similar shape, the quadratic shape. When it is said that the error are $O(h^2)$, this must be intended as *order of magnitude*. A similar exercise done using forward difference derivatives, shown in the following code snippet, displays indeed a linear trend because of errors there being $O(h)$.

```r
# Loop over all values of h.
# At each loop we need to redefine the grid and tabulate f(x)
DfF <- c()
for (h in hh) {
  x <- seq(1-2*h,by=h,length.out=5)
  f <- exp(x)
  tmp <- deriv_reg(x[3],x,f,scheme="f")
  DfF <- c(DfF,tmp)
}

# Errors (the true derivative is exp(1))
Deltas <- abs(DfT - DfF)

# Function h^2 compared graphically to the numerical derivative
ff <- hh
ylim <- range(ff,Deltas)
plot(hh,Deltas,type="b",pch=16,cex=0.7,ylim=ylim,xlab="h",ylab="Errors")
points(hh,ff,type="b",col=2)
```



7.1.3 Exercise 03

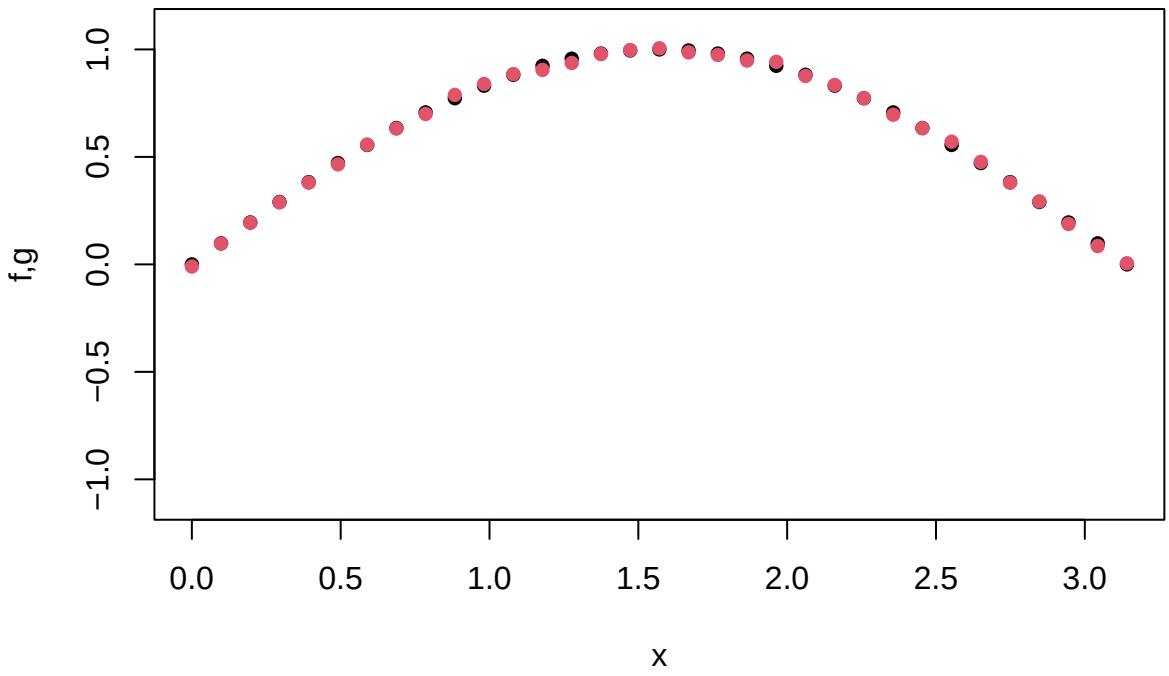
Experimental or measurement errors on sampled points of a function can amplify the errors of a numerical derivative. In this exercise, create a function, $f(x) = \sin(x)$ at 33 regularly spaced points between 0 and π (h is then roughly 0.1). Then create a function $g(x) = f(x) + \epsilon$, where ϵ is a normal random variable extracted from a distribution with mean 0 and standard deviation 0.01. Plot the two functions together to appreciate their difference. Next, calculate the centred difference derivative at all points (clearly excluding 0 and π)

for both $f(x)$ and $g(x)$. Plot both derivatives and appreciate how the differences for them are amplified, compared to the differences of the functions. Finally, find the maximum error for both derivatives with respect to the true derivative, $f'(x) = \cos(x)$.

SOLUTION

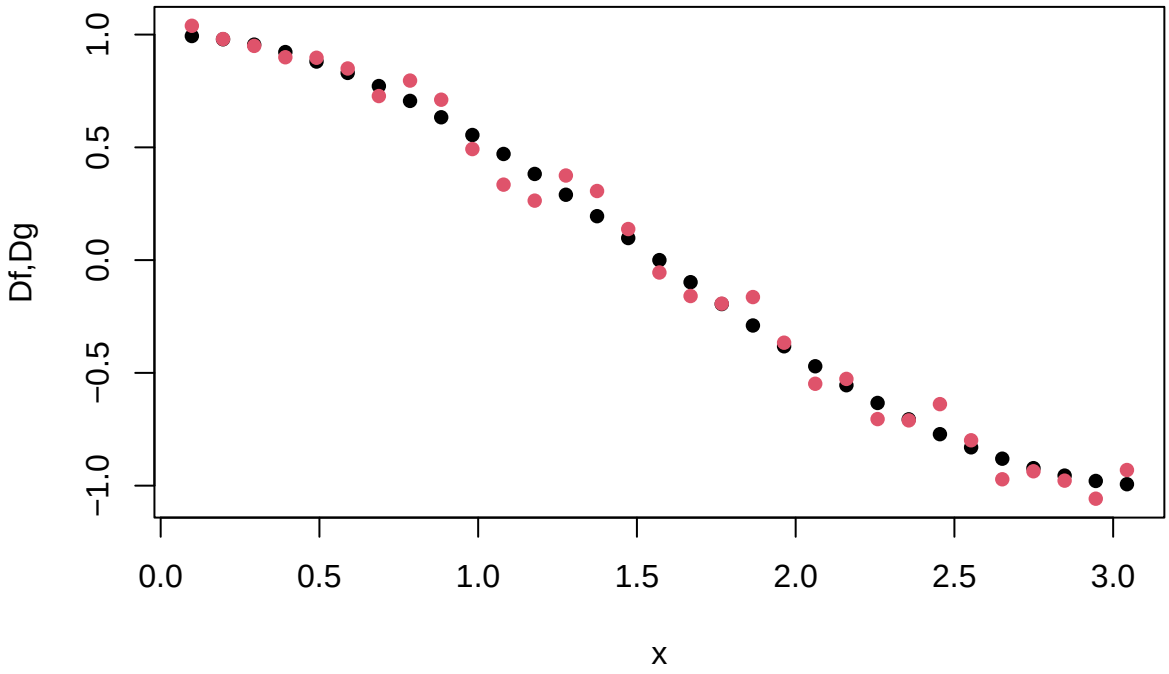
First we create the tabulated values for $f(x)$ and $g(x)$ and plot them to visually show that they are visually quite close.

```
# x grid
x <- seq(0,pi,length.out=33)

# Simulated errors. The seed is added for reproducibility
set.seed(167)
eps <- rnorm(n=33,mean=0,sd=0.01)

# Tabulated values for f and g
f <- sin(x)
g <- sin(x) + eps

# Plots
plot(x,f,pch=16,ylim=c(-1.1,1.1),xlab="x",ylab="f,g")
points(x,g,pch=16,col=2)
```



The plots of the function with and without errors are very similar, as the standard deviation is small compared to the scale of the plot, $\sigma = 0.01$. Next, we calculate the centred difference derivative for both $f(x)$ and $g(x)$. The plots show that they are substantially different.

```
# Centred difference derivative of f(x)
Df <- deriv_reg(x0=x[2:32],x=x,f=f,scheme="c")

# Centred difference derivative of g(x)
Dg <- deriv_reg(x0=x[2:32],x=x,f=g,scheme="c")

# Plots
ylim=range(Df,Dg)
plot(x[2:32],Df,pch=16,ylim=ylim,xlab="x",ylab="Df,Dg")
points(x[2:32],Dg,pch=16,col=2)
```



The two plots are visibly different. It would be nice to calculate the errors of both numerical derivatives by comparison with the correct derivative, $f'(x) = \cos(x)$. This is done in the next code snippet, where the absolute value of the maximum error for both $f'(x)$ and $g'(x)$ is calculated.

```
# Correct first derivative at the grid points of interest
Dtrue <- cos(x[2:32])

# Max error for numerical derivative of f
Emaxf <- max(abs(Df - Dtrue))
print(Emaxf)
#> [1] 0.00159787552799118

# Max error for numerical derivative of g
Emaxg <- max(abs(Dg - Dtrue))
print(Emaxg)
```

```
#> [1] 0.136777997257011
```

As we can see, the maximum error for $g'(x)$ is much larger than the numerical error due to the numerical calculation of the derivative: $0.137 \gg 0.002$. This must be taken into account when calculating derivatives of empirical functions.

7.1.4 Exercise 04

Find the numerical first derivative of $f(x) = x^2 - 3\cos(2x) + e^x$ at $x_0 = -\pi/6, 0, \pi/6$, when the function is tabulated at the 7 points $x = -\pi/2, -\pi/3, -\pi/4, 0, \pi/4, \pi/3, \pi/2$. Compare the values found with the values of the exact, analytic derivative.

SOLUTION

The tabulated values must be initially created as vectors `x` and `f`. Then the values at which the first derivative needs to be calculated will be stored in another vector, `x0`. The numerical derivative is calculated straight away with `deriv_irr`.

```
# Tabulated values of function
x <- c(-pi/2,-pi/3,-pi/4,0,pi/4,pi/3,pi/2)
f <- x^3-3*cos(2*x)+exp(x)

# Values for the derivative
x0 <- c(-pi/6,0,pi/6)

# Numerical derivative
f1 <- deriv_irr(x0,x,f)
print(f1)
#> [1] -3.78423375932125  1.00035093141224  7.70925068696071
```

The analytic expression of the derivative is $f'(x) = 3x^2 + 6\sin(2x) + e^x$. It is therefore possible to compute the exact numerical values of the derivative at the points assigned, and compare them with the corresponding approximated values.

```
# Exact analytic derivative at the x0 values
f1exact <- 3*x0^2+6*sin(2*x0)+exp(x0)

# Comparison
print(f1)
#> [1] -3.78423375932125  1.00035093141224  7.70925068696071
print(f1exact)
#> [1] -3.78130054209413  1.00000000000000  7.70671125109521
```

7.1.5 Exercise 05

Considering the case in Exercise 04, try and give an estimate of the errors associated with the numerical derivatives calculated at the 7 grid points.

SOLUTION

Let us reproduce in memory all values needed.

```
# Reproduce data
x <- c(-pi/2,-pi/3,-pi/4,0,pi/4,pi/3,pi/2)
f <- x^3-3*cos(2*x)+exp(x)

# The values for the derivative are the same grid points
f1 <- deriv_irr(x,x,f)
```

```
# Exact values
f1exact <- 3*x^2+6*sin(2*x)+exp(x)

# Comparison and absolute error
print(f1)
#> [1]  7.33050107628254 -1.53240587953911 -3.70693558509413  1.00035093141224
#> [5] 10.05679092683793 11.31333155612217 12.48681769686873
print(f1exact)
#> [1]  7.61008287716778 -1.55536448183177 -3.69351104702975  1.00000000000000
#> [5] 10.04383087594227 11.33567446462945 12.21268068178237
aerr <- abs(f1-f1exact)
print(aerr)
#> [1] 0.279581800885235765 0.022958602292654229 0.013424538064378044
#> [4] 0.000350931412239186 0.012960050895664210 0.022342908507278381
#> [7] 0.274137015086360236
```

It is interesting to observe, as expected, that the smallest errors occur in the central region of the grid because all differences of the type $x - x_i$ tend to be smaller than when x is close to the extreme of the interval.

The error is given by the formula

$$\Delta f'(x_i) = \frac{f^{(7)}(\xi)}{(n+1)!}(x_i - x_1)\cdots(x_i - x_{i-1})(x_i - x_{i+1})\cdots(x_i - x_7).$$

We don't know how to evaluate $f^{(7)}(\xi)/7!$ exactly. But we can calculate $f^{(7)}$ analytically and try and work out its minimum and maximum in the range $[-\pi/2, \pi/2]$. It turns out that

$$f^{(7)}(x)/7! = (e^x - 384\sin(2x))/5040.$$

Now, the largest value of e^x in the interval $[-\pi/2, \pi/2]$ is approximately 4.8105 and $\sin(2x)$ varies between -1 and $+1$. Therefore the largest value (in magnitude) for $f^{(7)}(\xi)/7!$ is 0.077145. This is what we need to give upper bounds for the error, as calculated in the following chunk.

```
# Maximum of f^(7)(xi)/7!
k <- (exp(pi/2)+384)/5040
print(k)
#> [1] 0.07714493598828 68

# Largest values for the error, depending on x_i used
idx <- 1:7
err <- c()
for (i in 1:7) {
  err <- c(err,k*abs(prod((x[i]-x)[idx[-i]])))
}

# Display and compare (TRUE or FALSE)
print(err)
#> [1] 0.9657071327890298 0.1112748959592297 0.0844993741190401 0.1287609510385373
#> [5] 0.0844993741190401 0.1112748959592297 0.9657071327890298
print(aerr)
#> [1] 0.279581800885235765 0.022958602292654229 0.013424538064378044
#> [4] 0.000350931412239186 0.012960050895664210 0.022342908507278381
#> [7] 0.274137015086360236
aerr <= err
#> [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

The actual errors are always less or equal than the largest expected margin. It is important to underlie, though, that it is not always possible to provide a sensible range of values to the quantity $f^{(n+1)}(\xi)/(n+1)!$.

7.1.6 Exercise 06

Using reasoning similar to the one used to derive the three-point formula for the centred difference derivative, derive a three-point formula for the second derivative. Apply the formula found to calculate numerically the second derivative of the function $f(x) = x^2$ on the regular grid from 0 to 1, using $h = 0.01$.

SOLUTION

The starting point are the Taylor expansions in powers of h and $-h$:

$$f(x_i + h) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2}h^2 + \frac{f'''(x_i)}{6}h^3 + \frac{f^{iv}(x_i)}{24}h^4 + \mathrm{O}(h^5)$$

$$f(x_i - h) = f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2}h^2 - \frac{f'''(x_i)}{6}h^3 + \frac{f^{iv}(x_i)}{24}h^4 - \mathrm{O}(h^5).$$

Adding these formula yields:

$$f(x_i + h) + f(x_i - h) = 2f(x_i) + f''(x_i)h^2 + \frac{f^{iv}(x_i)}{12}h^4 + \mathrm{O}(h^6).$$

The second derivative can be obtained by reversing this formula and it is found it is given by knowledge of the three points, $x_i - h, x_i, x_i + h$:

$$f''(x_i) = \frac{f(x_i + h) + f(x_i - h) - 2f(x_i)}{h^2} + \mathrm{O}(h^2).$$

Let us now apply the formula to calculate the second derivative of $f(x) = x^2$ between 0 and 1. We should find a constant quantity as $f''(x) = 2$. In the following section of code, given to the presence of $x_i - h$, x_i, $x_i + h$, we will not be able to calculate f^{ii} for the first and last point of the grid.

```
# Step (h)
h <- 0.01

# Grid
x <- seq(0,1,by=h)

# Function
f <- x^2

# Length of grid
n <- length(x)

# Second derivative (three-point formula)
DDf <- rep(NA,length.out=n) # First and last grid points not used
DDf[2:(n-1)] <- (f[1:(n-2)]+f[3:n]-2*f[2:(n-1)])/h^2
print(DDf)
#>   [1]                NA 2.00000000000000 2.00000000000000 2.00000000000000
#>   [5] 2.00000000000000 1.99999999999999 2.00000000000001 1.99999999999999
#>   [9] 2.00000000000001 2.00000000000002 1.99999999999995 2.00000000000002
#>  [13] 2.00000000000002 1.99999999999999 1.99999999999999 2.00000000000006
#>  [17] 1.99999999999999 1.99999999999985 2.00000000000006 2.00000000000006
#>  [21] 1.99999999999978 2.00000000000020 2.00000000000006 1.99999999999992
#>  [25] 2.00000000000006 2.00000000000006 2.00000000000006 2.00000000000006
```

```
#>  [29] 1.99999999999978 2.00000000000006 2.00000000000006 2.00000000000006
#>  [33] 2.00000000000006 2.00000000000006 2.00000000000006 1.99999999999978
#>  [37] 2.00000000000033 2.00000000000033 2.00000000000033 1.99999999999978
#>  [41] 1.99999999999978 1.99999999999978 2.00000000000033 2.00000000000033
#>  [45] 1.99999999999978 1.99999999999978 2.00000000000033 1.99999999999922
#>  [49] 2.00000000000033 2.00000000000033 1.99999999999978 1.99999999999978
#>  [53] 1.99999999999978 1.99999999999978 2.00000000000089 1.99999999999978
#>  [57] 2.00000000000089 1.99999999999867 2.00000000000089 1.99999999999978
#>  [61] 1.99999999999978 1.99999999999978 1.99999999999978 1.99999999999978
#>  [65] 2.00000000000089 1.99999999999978 2.00000000000089 1.99999999999978
#>  [69] 1.99999999999978 1.99999999999978 1.99999999999867 1.99999999999978
#>  [73] 1.99999999999978 1.99999999999978 1.99999999999978 1.99999999999978
#>  [77] 1.99999999999978 1.99999999999978 1.99999999999978 1.99999999999978
#>  [81] 1.99999999999978 1.99999999999978 1.99999999999978 1.99999999999978
#>  [85] 2.00000000000200 1.99999999999978 2.00000000000200 1.99999999999756
#>  [89] 1.99999999999978 1.99999999999978 1.99999999999978 1.99999999999978
#>  [93] 1.99999999999978 1.99999999999978 1.99999999999978 1.99999999999978
#>  [97] 1.99999999999978 1.99999999999978 1.99999999999978 1.99999999999978
#> [101]                NA
```

In this specific example errors are not immediately visible, given the simple polynomial but things are in general different with other types of functions.

It is also of some interest if the same result can be found if the formula for the first derivative is applied twice.

```
# First derivative (centred difference)
Dcf <- deriv_reg(x,x,f)
print(Dcf)
#>   [1]                NA 0.020000000000000 0.040000000000000 0.060000000000000
#>   [5] 0.080000000000000 0.100000000000000 0.120000000000000 0.140000000000000
#>   [9] 0.160000000000000 0.180000000000000 0.200000000000000 0.220000000000000
#>  [13] 0.240000000000000 0.260000000000000 0.280000000000000 0.300000000000000
#>  [17] 0.320000000000000 0.340000000000000 0.360000000000000 0.380000000000000
#>  [21] 0.400000000000000 0.420000000000000 0.440000000000000 0.460000000000000
#>  [25] 0.480000000000000 0.500000000000000 0.520000000000000 0.540000000000000
#>  [29] 0.559999999999999 0.579999999999999 0.600000000000001 0.620000000000000
#>  [33] 0.640000000000000 0.660000000000001 0.680000000000001 0.699999999999999
#>  [37] 0.719999999999998 0.740000000000000 0.760000000000001 0.780000000000001
#>  [41] 0.800000000000001 0.819999999999997 0.839999999999998 0.860000000000001
#>  [45] 0.880000000000002 0.900000000000001 0.920000000000001 0.939999999999999
#>  [49] 0.959999999999997 0.980000000000000 1.000000000000001 1.020000000000001
#>  [53] 1.040000000000002 1.060000000000000 1.080000000000000 1.100000000000001
#>  [57] 1.120000000000001 1.139999999999997 1.159999999999994 1.180000000000000
#>  [61] 1.200000000000001 1.220000000000002 1.240000000000002 1.260000000000000
#>  [65] 1.280000000000000 1.300000000000001 1.320000000000002 1.340000000000002
#>  [69] 1.360000000000000 1.380000000000001 1.399999999999996 1.419999999999993
#>  [73] 1.439999999999997 1.460000000000000 1.480000000000004 1.500000000000001
#>  [77] 1.519999999999999 1.540000000000002 1.560000000000006 1.580000000000004
#>  [81] 1.600000000000001 1.619999999999999 1.639999999999997 1.659999999999989
#>  [85] 1.679999999999993 1.700000000000002 1.720000000000005 1.740000000000003
#>  [89] 1.760000000000000 1.780000000000004 1.800000000000002 1.819999999999999
#>  [93] 1.840000000000003 1.860000000000006 1.879999999999998 1.899999999999991
#>  [97] 1.919999999999994 1.939999999999997 1.960000000000001 1.980000000000004
#> [101]                NA
```

```
# Second derivative - One more first derivative
DDcf <- deriv_reg(x,x,Dcf)
print(DDcf)
#>   [1]                NA                NA 2.00000000000000 2.00000000000000
#>   [5] 2.00000000000000 2.00000000000000 2.00000000000000 2.00000000000000
#>   [9] 2.00000000000000 2.00000000000000 1.99999999999999 2.00000000000000
#>  [13] 2.00000000000001 1.99999999999999 1.99999999999999 2.00000000000002
#>  [17] 2.00000000000000 1.99999999999997 2.00000000000003 2.00000000000000
#>  [21] 1.99999999999995 2.00000000000004 2.00000000000002 1.99999999999997
#>  [25] 2.00000000000002 2.00000000000002 1.99999999999999 1.99999999999995
#>  [29] 1.99999999999996 2.00000000000006 2.00000000000006 1.99999999999999
#>  [33] 2.00000000000002 2.00000000000002 1.99999999999989 1.99999999999989
#>  [37] 2.00000000000009 2.00000000000012 2.00000000000006 1.99999999999999
#>  [41] 1.99999999999978 1.99999999999985 2.00000000000020 2.00000000000020
#>  [45] 1.99999999999999 1.99999999999999 1.99999999999992 1.99999999999978
#>  [49] 2.00000000000006 2.00000000000020 2.00000000000005 2.00000000000006
#>  [53] 1.99999999999992 1.99999999999991 2.00000000000006 2.00000000000006
#>  [57] 1.99999999999978 1.99999999999965 2.00000000000019 2.00000000000033
#>  [61] 2.00000000000006 2.00000000000006 1.99999999999992 1.99999999999991
#>  [65] 2.00000000000006 2.00000000000006 2.00000000000006 1.99999999999992
#>  [69] 1.99999999999991 1.99999999999978 1.99999999999965 2.00000000000006
#>  [73] 2.00000000000033 2.00000000000033 2.00000000000006 1.99999999999978
#>  [77] 2.00000000000006 2.00000000000033 2.00000000000006 1.99999999999978
#>  [81] 1.99999999999978 1.99999999999978 1.99999999999950 1.99999999999978
#>  [85] 2.00000000000061 2.00000000000061 2.00000000000006 1.99999999999978
#>  [89] 2.00000000000006 2.00000000000006 1.99999999999978 2.00000000000006
#>  [93] 2.00000000000033 1.99999999999978 1.99999999999922 1.99999999999978
#>  [97] 2.00000000000033 2.00000000000033 2.00000000000033                NA
#> [101]                NA
```

The result is the same, but now the second derivative could not be computed at four, rather than two points, having used centred difference twice.

7.2 Exercises on Integration

7.2.1 Exercise 07

Calculate the following integral,

$$\frac{1}{\sqrt{2\pi}} \int_{-1}^{+1} e^{-x^2/2} \, \mathrm{d}x,$$

numerically using the trapezoid, and Simpson's 1/3 and 3/8 rules. Compare the results obtained with those displayed with the R function `pnorm`.

SOLUTION

The given integral is a definite integral of the Gaussian function. To be more specific, the given function is normalised so that

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} e^{-x^2/2} \, \mathrm{d}x = 1.$$

Therefore, the definite integral will be smaller than 1. The integral

$$\Phi(x) \equiv \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} \, \mathrm{d}t$$

is calculated in R with the function `pnorm`. For example, the integral between $-\infty$ and 0 must be 1/2 because the integral from $-\infty$ to $+\infty$ is 1. Indeed,

```
print(pnorm(0))
#> [1] 0.5
```

Accordingly, the requested integral can be given as difference of Φ functions,

$$\frac{1}{\sqrt{2\pi}} \int_{-1}^{+1} e^{-x^2/2} \, \mathrm{d}x = \Phi(+1) - \Phi(-1)$$

that is, difference of `pnorm` at $+1$ and -1:

```
print(pnorm(1)-pnorm(-1))
#> [1] 0.682689492137086
```

Now that we know the correct value,let us calculate it with the requested approximations. The range $[-1, 1]$ can be divided into an even number of intervals which is also a multiple of 3 so that all rules can be applied in full with a same grid; this number is chosen as $n = 198$. Initially the appropriate grid and the tabulated values must be created:

```
# Grid (divisible by 3 and even)
n <- 198
x <- seq(-1,1,length.out=(n+1))

# Tabulated values
f <- exp(-x^2/2)/sqrt(2*pi)
```

then the three algorithms are applied using the function `numint_reg`.

1. **Trapezoid**

```
value <- numint_reg(x,f,scheme="trap")
print(value)
#> [1] 0.682685377394583
```

2. **Simpson's 1/3**

```
value <- numint_reg(x,f,scheme="sim13")
print(value)
#> [1] 0.682689492193062
```

3. **Simpson's 3/8**

```
value <- numint_reg(x,f,scheme="sim38")
print(value)
#> [1] 0.68268949226304
```

We can see that while the accuracy is quite satisfactory even with the trapezoid rule, Simpson's 1/3 reproduces the correct results to seven decimals. Obviously, Simpson's 3/8 does it as well, but as the accuracy of both hmethods is the same ($\mathrm{O}(h^4)$), Simpson's 1/3 is normally the preferred method of choice, given its relative algorithmic simplicity.

7.2.2 Exercise 08

Consider the following *complete elliptic integral of the first kind*,

$$K(k) \equiv \int_0^{\pi/2} \frac{\mathrm{d}\theta}{1 - k^2 \sin^2(\theta)},$$

where $k \in (-1, 1)$. Calculate $K(0.5)$ numerically using `numint_reg` and compare your result with that calculated using the R package `elliptic`.

SOLUTION

The integrand when $k = 0.5$ is given by the following function:

$$f(\theta) = \frac{1}{\sqrt{1 - 0.25\sin^2(\theta)}}.$$

The numerical integral must be calculated between 0 and $\pi/2$. We can use a grid with $n = 200$.

```
# Grid
x <- seq(0,pi/2,length.out=201)

# Function values
f <- 1/(sqrt(1-0.25*(sin(x))^2))

# Integral
nvalue <- numint_reg(x,f)
pval <- sprintf("%10.8f\n",nvalue)
cat(pval)
#> 1.68575035
```

One of the R packages that calculates the complete elliptic integrals of the first kind is *elliptic*. Its function `K.fun` does the job. The parameter `m` of the package is what in the above notation has been indicated as k^2. So:

```
# Make sure library(elliptic) has loaded the package
require(elliptic)
#> Loading required package: elliptic
#>
#> Attaching package: 'elliptic'
#> The following objects are masked from 'package:stats':
#>
#>     sd, sigma
#> The following object is masked from 'package:base':
#>
#>     is.primitive

# Integral. m=k^2=0.5^2=0.25
nvalue <- K.fun(m=0.25)
pval <- sprintf("%10.8f\n",nvalue)
cat(pval)
#> 1.68575035
```

the numeric values are identical to a high accuracy.

7.2.3 Exercise 09

Calculate the local error for Simpson's 3/8 rule and verify that the result is $-(3/80)h^5 f^{(4)}(\xi)$.

SOLUTION

The interpolation error, given that for SImpson's 3/8 rule we use x_i, x_{i+1}, x_{i+2}, and x_{i+3}, is

$$\Delta P_3(x) = \frac{f^{(4)}(\xi)}{4!}(x - x_i)(x - x_{i+1})(x - x_{i+2})(x - x_{i+3}).$$

Using the integration variable $s = (x - x_i)/h$, we can then write the local error as the following integral:

$$\text{local error} = \frac{f^{(4)}(\xi)}{4!}h^5 \int_0^3 s(s-1)(s-2)(s-3)\, \mathrm{d}s.$$

The definite integral in the above expression turns out to be $-9/10$. Therefore:

$$\text{local error} = \frac{f^{(4)}(\xi)}{4!}h^5\left(-\frac{9}{10}\right) = -\frac{3}{80}h^5 f^{(4)}(\xi),$$

as suggested by the exercise.

7.2.4 Exercise 10

The global error when applying the trapezoid rule is

$$\frac{a-b}{12}f^{(2)}(\xi)h^2.$$

While it is not possible to know the value of $f^{(2)}(\xi)$, the quantity $(a-b)/12$ is constant and, even though $f^{(2)}(\xi)$ varies across the interval (x_1, x_{n+1}), it will be bounded by a finite number, in general comparable with the values that the function takes in the integration interval. Accordingly, the global error should show a square dependency on h.

Use the trapezoid rule to integrate $f(x) = x^2 - 1$ in the interval $[-1, 1]$, for many values of h when $[-1, 1]$ is divided into $n = 20, 21, \ldots, 39, 40$ equal intervals. Plot the global error versus the corresponding values of h. You should verify visually that the set of points in the plot follows a curved, rather than straight, pattern. Can we ascertain that the curve is a quadratic?

SOLUTION

A plot of the global error vs h should appear as a parabola. To calculate the error we need to know the correct value of the integral. This is readily calculated:

$$I = \int_{-1}^{+1}(x^2-1)\,\mathrm{d}x = \left[\frac{x^3}{3} - x\right]_{-1}^{+1} = \frac{1}{3} - 1 + \frac{1}{3} - 1 = -\frac{4}{3} \approx -1.33.$$

If $J(h)$ indicates the approximated value of the integral using the trapezoid rule with a specific value of h, the global error will be

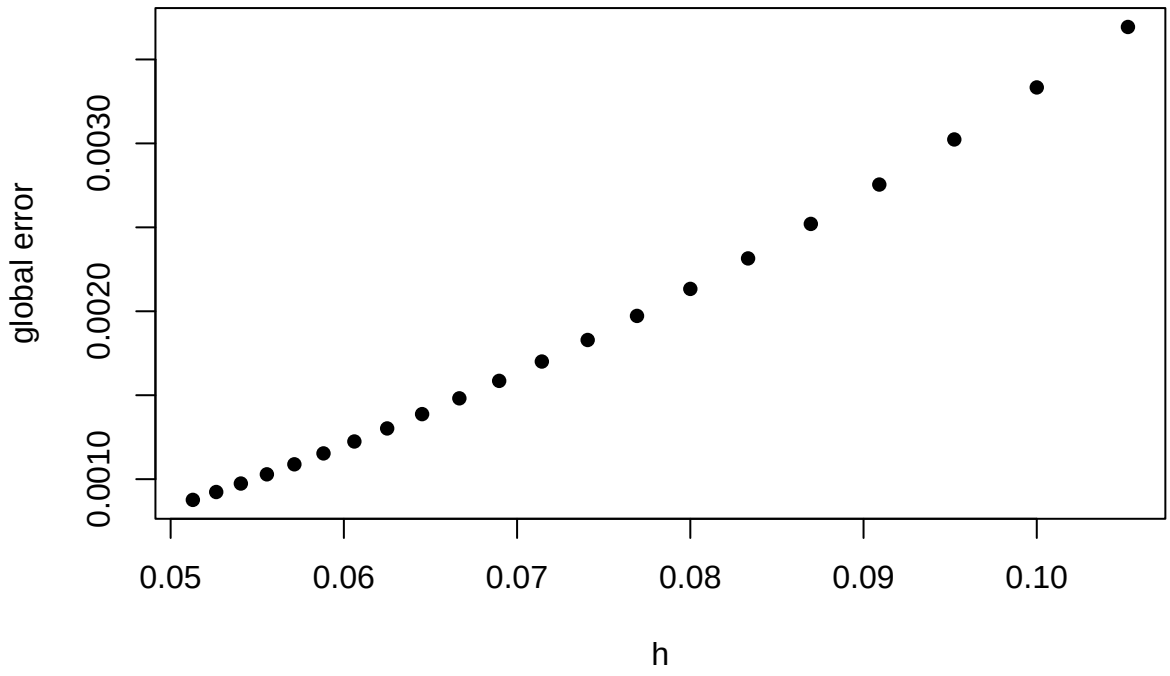
$$\Delta J(h) = J(h) - I.$$

We need to plot the values of $\Delta J(h)$ versus h.

The values of $J(h)$ can be easily obtained using `numint_reg()`, with `scheme="trap"`. Also, the way to determine the value of h, given the number of regularly-spaced intervals, is through the difference, say, of the first two grid points.

```
# Correct integral
I <- -4/3

# Define range of intervals (n)
n <- 20:40

# Vector of h
h <- c()

# Vector of global errors
ge <- c()

# Main loop
for (m in n) {
  # Define x grid and corresponding f's
  x <- seq(-1,1,length.out=m)
  f <- x^2-1
```

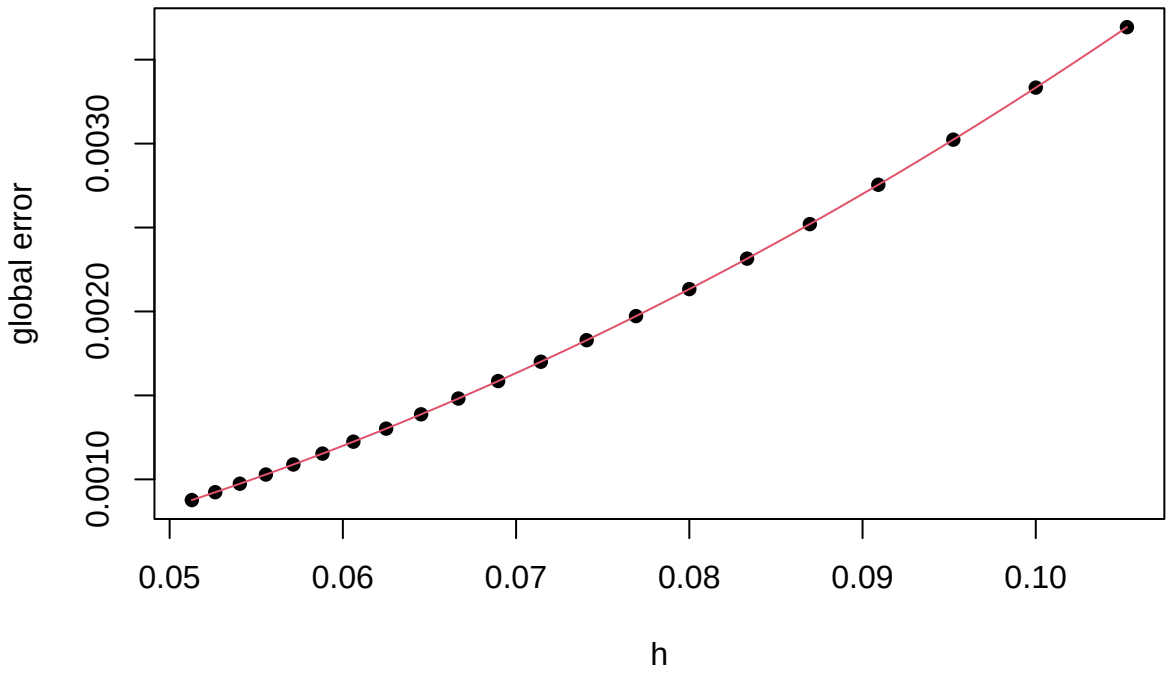
```
  # Store h
  h <- c(h,x[2]-x[1])

  # Numeric integral
  J <- numint_reg(x,f,scheme="trap")

  # Global error
  ge <- c(ge,J-I)
}

# Plot
plot(h,ge,pch=16,xlab="h",ylab="global error")
```



It indeed appears that the set of points visually curves. But to ascertain that they fit a quadratic curve, we need to carry out a linear regression with h^2.

```
# Linear regression
mdel <- lm(ge ~ I(h^2))
summary(mdel)
#>
#> Call:
#> lm(formula = ge ~ I(h^2))
#>
#> Residuals:
#>                 Min                  1Q              Median                  3Q
#> -1.13790496914e-15 -5.88818916368e-16 -1.60108694749e-16  5.57538232591e-16
```

```
#>                 Max
#>  1.24480615003e-15
#>
#> Coefficients:
#>                       Estimate        Std. Error           t value Pr(>|t|)
#> (Intercept) 6.73814900786e-17 3.91525037296e-16 1.72100000000e-01  0.86518
#> I(h^2)      3.33333333333e-01 6.53149101735e-14 5.10347993204e+12  < 2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 7.32844836224e-16 on 19 degrees of freedom
#> Multiple R-squared:               1,  Adjusted R-squared:               1
#> F-statistic: 2.60455074167e+25 on 1 and 19 DF,  p-value: < 2.220446049e-16

# Plot data and regression line
xx <- seq(min(h),max(h),length.out=100)
yy <- predict(mdel,newdata=data.frame(h=xx))
plot(h,ge,pch=16,xlab="h",ylab="global error")
points(xx,yy,type="l",col=2)
```



The regression's statistics are strongly in favour of a quadratic curve, thus conferming the theoretical results.

7.2.5 Exercise 11

Use 2-point and 3-point Gaussian quadrature to estimate numerically the integral

$$\int_{-2}^{3} x^4 \, \mathrm{d}x.$$

What difference do you observe in going from the 2-point to the 3-point quadrature?

SOLUTION

An n-point Gaussian quadrature can give the exact estimate of definite integrals of polynomials of up to degree $2n-1$. For $n=2$, the exact estimate is for polynomials of degree up to 3, while for $n=3$, the degree goes up to 5. As the integrand in the exercise is a polynomial of degree 4, the 2-point quadrature will not return the exact value but the 3-point quadrature will.

The correct value of the integral is

$$\int_{-2}^{3} x^4 \, \mathrm{d}x = \left[\frac{x^5}{5}\right]_{-2}^{3} = \frac{243}{5} + \frac{32}{5} = \frac{275}{5} = 55.$$

Let us, next, use the 2-point Gaussian quadrature:

```
# Function
f <- function(x) {ff <- x^4; return(ff)}

# 2-point quadrature
ltmp <- Gquad(f,-2,3,n=2)
print(ltmp$itg)
#> [1] 37.6388888888889
```

The result is not very close to 55, as expected. But with three points, exactness will be reached.

```
# 3-point quadrature
ltmp <- Gquad(f,-2,3,n=3)
print(ltmp$itg)
#> [1] 55
```

7.2.6 Exercise 12

Calculate numerically the integral

$$\frac{1}{\sqrt{2\pi}} \int_{-1}^{+1} e^{-x^2/2} \, \mathrm{d}x,$$

using Gaussian quadrature, what order n is necessary for the result to be comparable with the one given by the function `pnorm`?

SOLUTION

The value of the given definite integral can be calculated as difference $\Phi(1) - \Phi(-1)$, where

$$\Phi(x) = \frac{1}{\sqrt{2}} \int_{-\infty}^{x} e^{-t^2/2} \, \mathrm{d}t.$$

Clearly, $\Phi(-\infty) = 0$ and $\Phi(+\infty) = 1$. The function $\Phi(x)$ is calculated numerically with high precision using `pnorm`. For example

```
# A very large, negative value is close to -infinity
print(pnorm(-1000))
#> [1] 0
```

```r
# A very large, positive value is close to +infinity
print(pnorm(1000))
#> [1] 1
```

Therefore, an accurate numerical value for the given definite integral is

```r
Tvalue <- pnorm(1)-pnorm(-1)
res <- sprintf("Accurate value: %14.12f\n",Tvalue)
cat(res)
#> Accurate value: 0.682689492137
```

We can now observe the errors when using Gaussian quadrature as differences with the value `Tvalue` found.

```r
# Function
f <- function(x) {ff <- exp(-x^2/2)/sqrt(2*pi); return(ff)}

# 3-point quadrature
ltmp <- Gquad(f,-1,1,n=3)
res <- sprintf("3-point quadrature\n %14.12f\n",ltmp$itg)
cat(res)
#> 3-point quadrature
#>  0.682997260714
res <- sprintf("Absolute difference: %14.12f\n",abs(ltmp$itg-Tvalue))
cat(res)
#> Absolute difference: 0.000307768577

# 4-point quadrature
ltmp <- Gquad(f,-1,1,n=4)
res <- sprintf("4-point quadrature\n %14.12f\n",ltmp$itg)
cat(res)
#> 4-point quadrature
#>  0.682679806144
res <- sprintf("Absolute difference: %14.12f\n",abs(ltmp$itg-Tvalue))
cat(res)
#> Absolute difference: 0.000009685993

# 5-point quadrature (default)
ltmp <- Gquad(f,-1,1)
res <- sprintf("5-point quadrature\n %14.12f\n",ltmp$itg)
cat(res)
#> 5-point quadrature
#>  0.682689735388
res <- sprintf("Absolute difference: %14.12f\n",abs(ltmp$itg-Tvalue))
cat(res)
#> Absolute difference: 0.000000243251

# 6-point quadrature
ltmp <- Gquad(f,-1,1,n=6)
res <- sprintf("6-point quadrature\n %14.12f\n",ltmp$itg)
cat(res)
#> 6-point quadrature
#>  0.682689487053
res <- sprintf("Absolute difference: %14.12f\n",abs(ltmp$itg-Tvalue))
cat(res)
#> Absolute difference: 0.000000005084
```

Already the error is of the order of 10^{-7} for $n = 5$ and it goes down to 10^{-9} for $n = 6$.

7.2.7 Exercise 13

Adapt the code of function `Gquad` to write an algorithm to calculate numerically 2D integrals over rectangular domains. Use the algorithm developed to calculate

$$\iint_R ye^x \, \mathrm{d}x\mathrm{d}y, \qquad R = [0,2] \times [0,3].$$

SOLUTION

The code used in `Gquad` can be extracted simply by typing `Gquad` in an R console. The result yields the initial prototype shown in the following snippet.

```
function(f,a,b,n=5) {
  # The lowest quadrature is linear
  if (n < 1) stop("n must be a positive integer.")

  # n=1 must be treated separately
  if (n == 1) {
    x <- 0
    w <- 2
    xt <- (b-a)/2*x+(b+a)/2
    wt <- (b-a)/2*w

    # Compute weighted sum
    itg <- sum(wt*f(xt))

    # List containing zeros, weights, and integral
    ltmp <- list(xt=xt,wt=wt,itg=itg)

    return(ltmp)
  }

  # Golub-Welsch algorithm to avoid using a Legendre R package
  i <- 1:(n-1)
  a_diag <- rep(0,length.out=n)          # Diagonal entries (Legendre: all 0)
  b_sub <- i/sqrt(4*i^2-1)               # Sub-diagonal entries
  T <- diag(a_diag)                      # Build Jacobi matrix
  T[cbind(i+1,i)] <- b_sub               # Lower diagonal
  T[cbind(i,i+1)] <- b_sub               # Upper diagonal

  eig <- eigen(T,symmetric=TRUE)
  x <- eig$values                        # Nodes in [-1,1]
  V <- eig$vectors
  w <- 2 * (V[1,])^2                     # Weights

  # Transform from [-1,1] to [a,b]
  xt <- (b-a)/2*x+(b+a)/2
  wt <- (b-a)/2*w

  # Compute weighted sum
  itg <- sum(wt*f(xt))

  # List containing zeros, weights, and integral
```

```
  ltmp <- list(xt=xt,wt=wt,itg=itg)

  return(ltmp)
}
```

We can see that, being a function for 1D integration, `Gquad` only includes one set of extremes of integration. In the new function, we will need a set for x, `ax,bx` and a set for y, `ay,by`. Also, there is no immediate reason for the order of the quadrature for x to be the same of the order of the quadrature for y. So, we will use a `nx` and a `ny`.

Calculation of the nodes and weights can be done using the same lines as in the above code, making sure to do one set of calculations for `nx` and one set for `ny`. It is here convenient to arrange zeros, weights and sampled values for $f(x, y)$, in a same matrix form so that the type of final expression, `sum(wt*f(xt))`, does not need changing. This can be easily done using the R function `outer`, which is an *outer product of arrays*, and that includes the option to create arrays also from 2D functions, exactly what we need. Furthermore, to avoid useless complications in coding, connected to the special case $n = 1$, we will only allow values of `nx` and `ny` greater or equal than 2. These considerations lead to the new function `G2quad`, shown below.

```
G2quad <- function(f,ax=-1,bx=1,ay=-1,by=1,nx=5,ny=5) {
  # The lowest quadrature has 2 points
  if (nx < 2) stop("nx must be a positive integer.")
  if (ny < 2) stop("ny must be a positive integer.")

  ## Zeros and weights for x
  i <- 1:(nx-1)
  a_diag <- rep(0,length.out=nx)
  b_sub <- i/sqrt(4*i^2-1)
  T <- diag(a_diag)
  T[cbind(i+1,i)] <- b_sub
  T[cbind(i,i+1)] <- b_sub
  eig <- eigen(T,symmetric=TRUE)
  x <- eig$values
  V <- eig$vectors
  w <- 2*(V[1,])^2
  xt <- (bx-ax)/2*x+(bx+ax)/2
  wxt <- (bx-ax)/2*w

  ## Zeros and weights for y
  i <- 1:(ny-1)
  a_diag <- rep(0,length.out=ny)
  b_sub <- i/sqrt(4*i^2-1)
  T <- diag(a_diag)
  T[cbind(i+1,i)] <- b_sub
  T[cbind(i,i+1)] <- b_sub
  eig <- eigen(T,symmetric=TRUE)
  y <- eig$values
  U <- eig$vectors
  w <- 2*(U[1,])^2
  yt <- (by-ay)/2*y+(by+ay)/2
  wyt <- (by-ay)/2*w

  # Gaussian quadrature
  fM <- outer(xt,yt,f)
  W <- outer(wxt,wyt)
  itg <- sum(W * fM)
```

```
  return(itg)
}

print(class(G2quad))
#> [1] "function"
```

Make sure to examine and scrutinise the above code with care. It is important, when writing R code, to make use as much as possible of structural tools like `outer` in order to exploit parallel operations like the multiplication in `W*fM`.

Let us, next, use the function just implemented to calculate the given integral. The integral can also be easily calculated analytically:

$$\iint_R ye^x \, \mathrm{d}x\mathrm{d}y = \int_0^2 e^x \, \mathrm{d}x \int_0^3 y \, \mathrm{d}y = \frac{9}{2}(e^2 - 1) \approx 28.751.$$

The function is relatively easy to use, once the 2D integrand function is defined.

```
# Define 2D function
f <- function(x,y) {ff <- y*exp(x); return(ff)}

# 5-point quadrature: 2D integration
itg <- G2quad(f,0,2,0,3)
print(itg)
#> [1] 28.750752435099
```

The result confirms that the code works. The relative complexity of what coded above should give some ideas of the type of algorithms and the variety of code possible to calculate multiple integrals using the Gaussian quadrature.

8 Chapter 08

8.1 Exercises on IVPs

8.1.1 Exercise 01

The solution to the following IVPs,

$$ty' + y = 2t, \quad y(1) = 0,$$

is

$$y(t) = t - \frac{1}{t}, \quad t \neq 0.$$

Solve this ODE numerically using `EulerODE` and compare the result visually with the exact solution fot $t \in [1, 3]$, when using step size $h = 0.4, 0.2, 0.1$. What error is expected for the solution at $t = 3$? Is this reasonable?

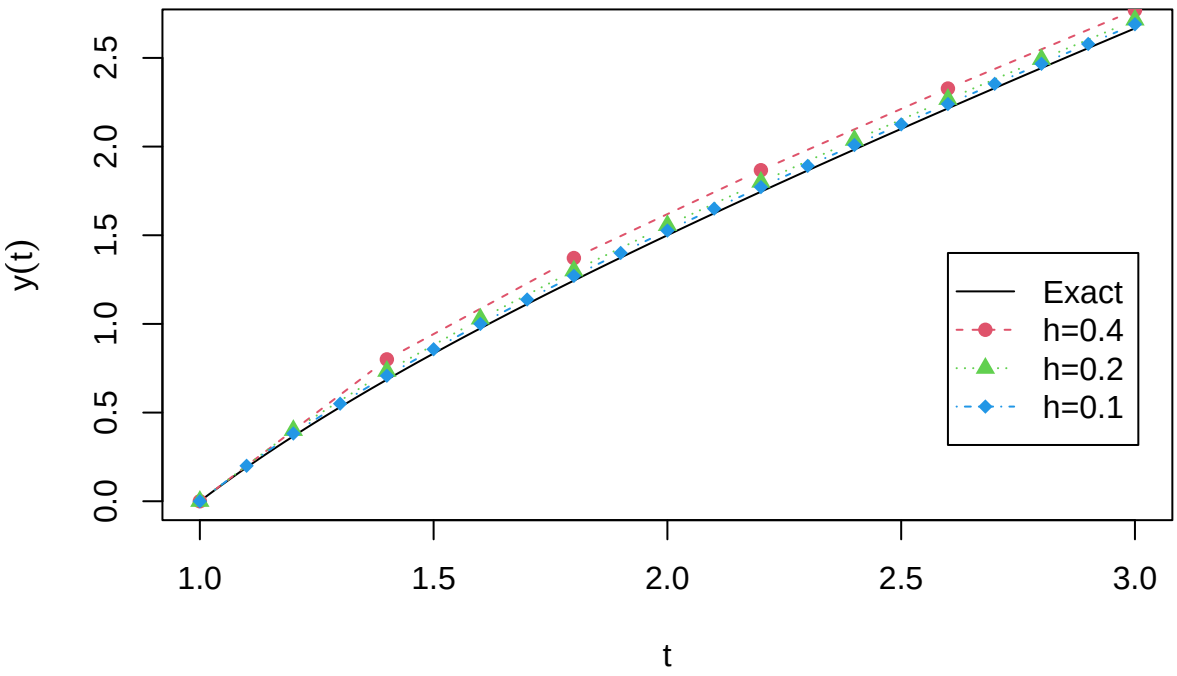
SOLUTION

The application of the solver `EulerODE` is relatively straightforward, once the gradient of the ODE is defined. The ODE can, in fact, be re-written as

$$y' \equiv \frac{\mathrm{d}y}{\mathrm{d}t} = \frac{1}{t}\left(2t - y\right).$$

It is also important to remember that here $t_0 = 1$ (and not $t_0 = 0$, a common mistake) and $t_f = 3$.

```r
# Define the gradient
f <- function(t,y) {ff <- (2*t-y)/t; return(ff)}

# Solution interval
t0 <- 1
tf <- 3

# Initial conditions
y0 <- 0

# Step sizes
h1 <- 0.4; h2 <- 0.2; h3 <- 0.1

# Euler solver
ltmp1 <- EulerODE(f,t0,tf,y0,h1)
ltmp2 <- EulerODE(f,t0,tf,y0,h2)
ltmp3 <- EulerODE(f,t0,tf,y0,h3)

# Exact solution
tt <- seq(1,3,length.out=100)
yy <- tt-1/tt

# Visual comparisons
plot(tt,yy,type="l",
     xlab=expression(t),ylab=expression(y(t)))
points(ltmp1$t,ltmp1$y,type="b",pch=16,col=2,lty=2)
points(ltmp2$t,ltmp2$y,type="b",pch=17,col=3,lty=3)
points(ltmp3$t,ltmp3$y,type="b",pch=18,col=4,lty=4)
legend(2.6,1.4,legend=c("Exact","h=0.4","h=0.2","h=0.1"),
       pch=c(-1,16,17,18),col=c(1,2,3,4),lty=c(1,2,3,4))
```

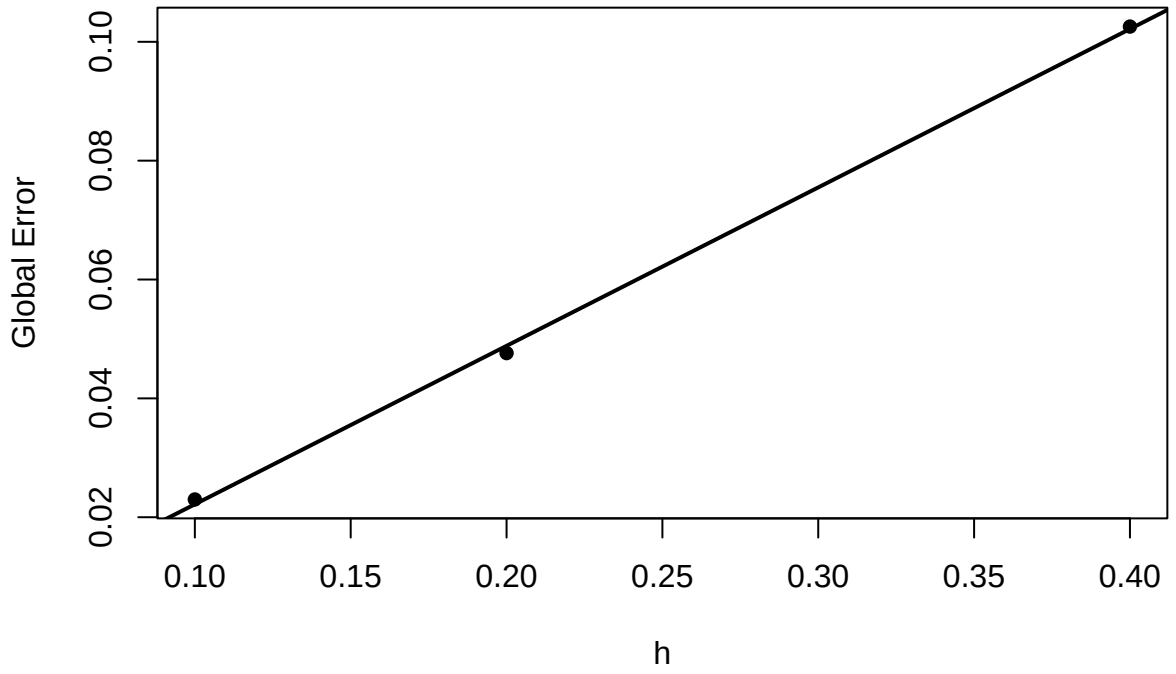


It is visually evident that the accuracy of the numerical solution increases with the decreasing size of h.

We know that the global error for the Euler method is $\mathrm{O}(h)$, which means that the difference $E = |y(t_f) - y_n|$ should decrease linearly with the decrease of h.

```
# Values at t=tf
yk <- yy[length(yy)]
yk1 <- ltmp1$y[length(ltmp1$y)]
yk2 <- ltmp2$y[length(ltmp2$y)]
yk3 <- ltmp3$y[length(ltmp3$y)]

# Errors
D1 <- abs(yk-yk1)
D2 <- abs(yk-yk2)
D3 <- abs(yk-yk3)

# Regression (linear dependency on h)
h <- c(h1,h2,h3)
D <- c(D1,D2,D3)
mdel <- lm(D ~ h)
print(summary(mdel))
#>
#> Call:
#> lm(formula = D ~ h)
#>
#> Residuals:
#>                 1                 2                 3
```

```
#>  0.000405997942944 -0.001217993828831  0.000811995885887
#>
#> Coefficients:
#>                      Estimate        Std. Error  t value Pr(>|t|)
#> (Intercept) -0.00448402172540  0.00186051630554 -2.41010 0.250385
#> h            0.26660531586640  0.00703209064947 37.91267 0.016788 *
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.00151910520223 on 1 degrees of freedom
#> Multiple R-squared:  0.999304768771, Adjusted R-squared:  0.998609537542
#> F-statistic: 1437.37037037 on 1 and 1 DF,  p-value: 0.0167878505438
plot(c(h1,h2,h3),c(D1,D2,D3),pch=16,xlab="h",ylab="Global Error")
abline(mdel,lwd=2)
```



The linear dependency of the global error on h is clearly visible.

8.1.2 Exercise 02

The solution to the following IVPs,

$$y' + \frac{1}{x}y = xy^2, \quad y(4) = -\frac{1}{4},$$

is

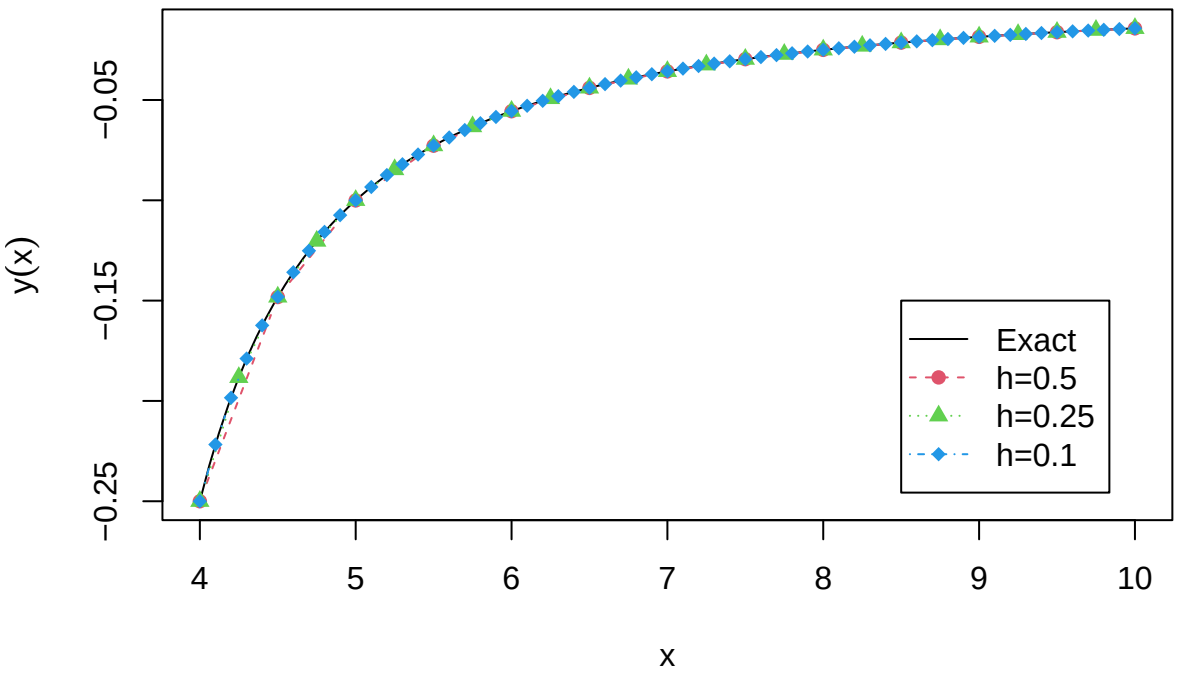
$$y(x) = \frac{1}{3x - x^2}.$$

Solve this ODE numerically using `RK4ODE` and compare the result visually with the exact solution fot $t \in [4, 10]$, when using step size $h = 0.5, 0.25, 0.1$. How can you extract the number of steps used by the method?

SOLUTION

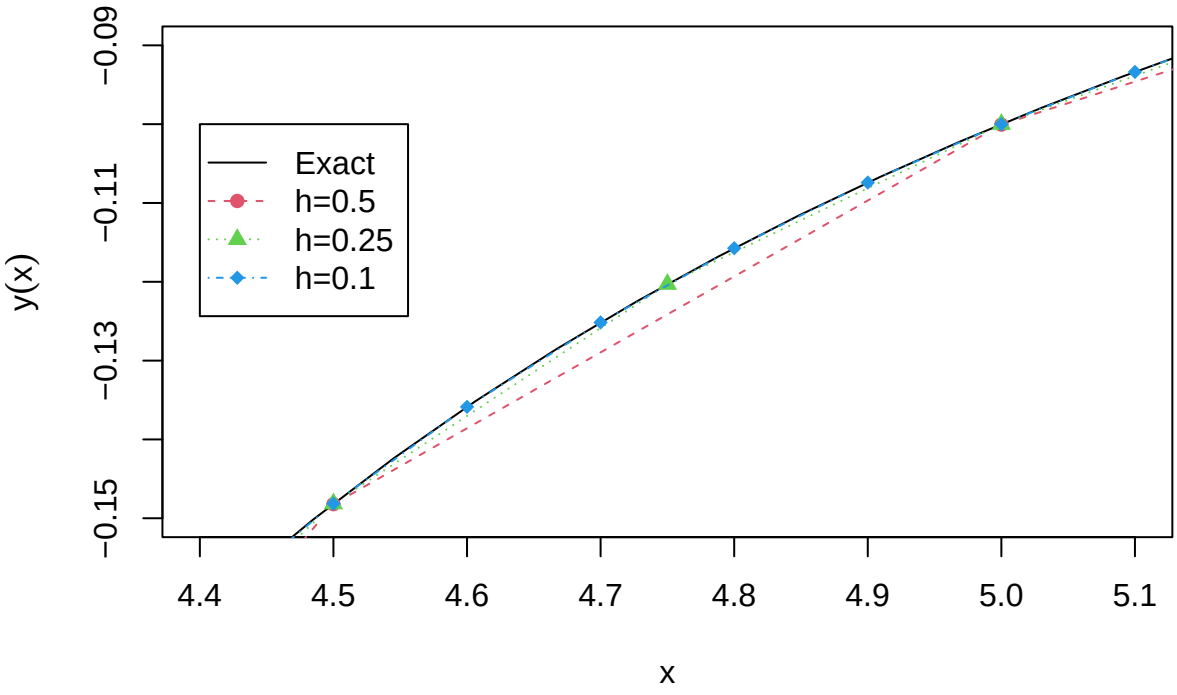
The exercise is straightforward. Functions, interval's extremes and initial conditions must be defined first. Then the algorithm is applied using `RK4ODE`.

```
# Define the gradient
f <- function(x,y) {ff <- x*y^2-y/x; return(ff)}

# Solution interval
t0 <- 4
tf <- 10

# Initial conditions
y0 <- -0.25

# Step sizes
h1 <- 0.5; h2 <- 0.25; h3 <- 0.1

# RK4 solver
ltmp1 <- RK4ODE(f,t0,tf,y0,h1)
ltmp2 <- RK4ODE(f,t0,tf,y0,h2)
ltmp3 <- RK4ODE(f,t0,tf,y0,h3)

# Exact solution
xx <- seq(4,10,length.out=100)
yy <- 1/(3*xx-xx^2)

# Visual comparisons
plot(xx,yy,type="l",
     xlab=expression(x),ylab=expression(y(x)))
points(ltmp1$t,ltmp1$y,type="b",pch=16,col=2,lty=2)
points(ltmp2$t,ltmp2$y,type="b",pch=17,col=3,lty=3)
points(ltmp3$t,ltmp3$y,type="b",pch=18,col=4,lty=4)
legend(8.5,-0.15,legend=c("Exact","h=0.5","h=0.25","h=0.1"),
       pch=c(-1,16,17,18),col=c(1,2,3,4),lty=c(1,2,3,4))
```



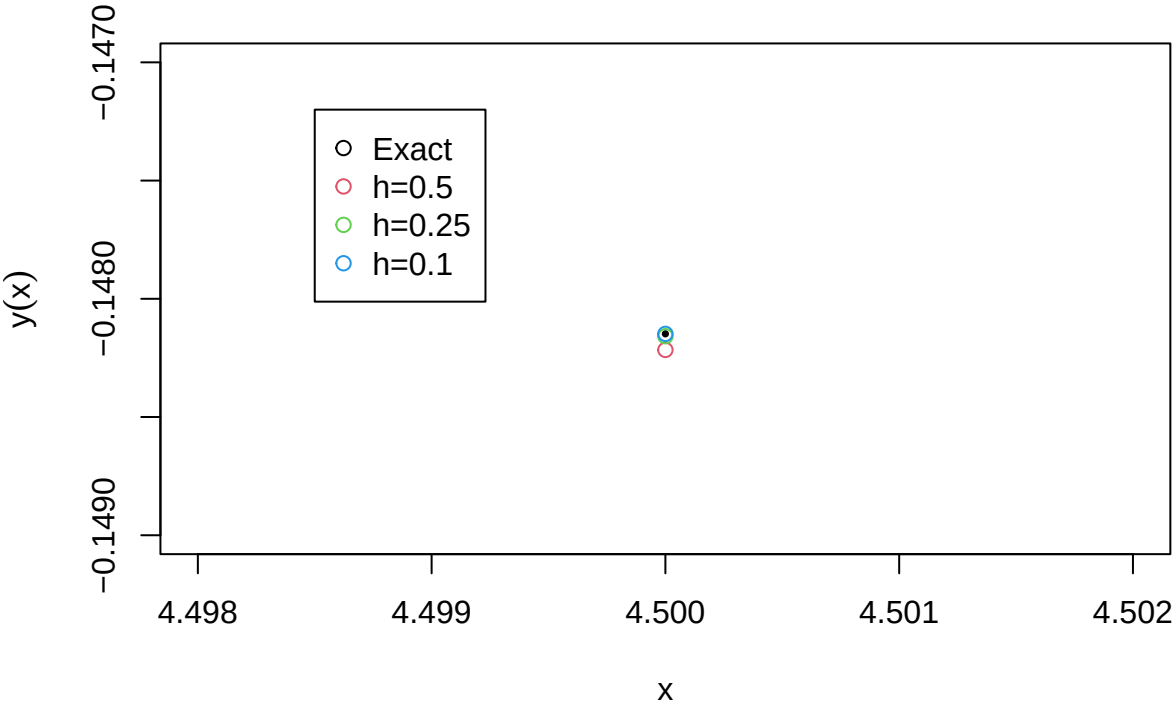
Details on the actual distance of the numerical solution from the exact one can be better appreciated when the plot is zoomed in, for example close to the elbow of the curve.

```
plot(xx,yy,type="l",xlim=c(4.4,5.1),ylim=c(-0.15,-0.09),
     xlab=expression(x),ylab=expression(y(x)))
points(ltmp1$t,ltmp1$y,type="b",pch=16,col=2,lty=2)
points(ltmp2$t,ltmp2$y,type="b",pch=17,col=3,lty=3)
points(ltmp3$t,ltmp3$y,type="b",pch=18,col=4,lty=4)
legend(4.4,-0.1,legend=c("Exact","h=0.5","h=0.25","h=0.1"),
       pch=c(-1,16,17,18),col=c(1,2,3,4),lty=c(1,2,3,4))
```



It is not a sign of poor accuracy that the lines connecting the different points stick out of the curve more evidently for the solution with $h = 0.5$, because these points are far apart. This is less evident, but still the case, for the solution with $h = 0.25$ and $h = 0.1$. But the points itself are relatively close to the correct value of the solution, as shown, for example, in the following plot around the value $x = 4.5$.

```
# Exact value at x=4.5
plot(4.5,1/(3*4.5-4.5^2),pch=16,cex=0.5,
     xlim=c(4.498,4.502),ylim=c(-0.149,-0.147),
     xlab=expression(x),ylab=expression(y(x)))
points(ltmp1$t,ltmp1$y,pch=1,col=2,lty=2)
points(ltmp2$t,ltmp2$y,pch=1,col=3,lty=3)
points(ltmp3$t,ltmp3$y,pch=1,col=4,lty=4)
legend(4.4985,-0.1472,legend=c("Exact","h=0.5","h=0.25","h=0.1"),
       pch=c(1,1,1,1),col=c(1,2,3,4))
```



All points, even the one corresponding to $h = 0.5$, are close to the exact solution. That gives a measure of the accuracy of the RK4 solver.

The number of steps required by a method can be simply extracted as the size of any of the arrays returned by the solvers. For example:

```
# Number of steps when h=0.5
print(length(ltmp1$t))
#> [1] 13

# Number of steps when h=0.25
print(length(ltmp2$t))
#> [1] 25

# Number of steps when h=0.1
print(length(ltmp3$t))
#> [1] 61
```

8.1.3 Exercise 03

In a closed environment, a biological population may grow quickly at first, but then slow down as resources become limited. This process can be modelled by the *logistic growth equation*:

$$\frac{dn}{dt} = rn\left(1 - \frac{n}{K}\right),$$

where:

- $n(t)$ is the population size at time t,

- $r > 0$ is the intrinsic growth rate,
- $K > 0$ is the *carrying capacity* (the maximum population that the environment can sustain).

The equation assumes that the population grows approximately exponentially when small, $n \ll K$, but that growth slows and eventually stops as n approaches K.

In this exercise, we consider the case:

- Growth rate: $r = 0.5$,
- Carrying capacity: $K = 1000$,
- Initial population: $n(0) = 50$.

The analytical solution to the logistic equation is:

$$n(t) = \frac{K n_0 e^{rt}}{K + n_0(e^{rt} - 1)}.$$

1. Use the analytical solution to compute the exact population values at $t = 5$, $t = 10$, and $t = 20$.
2. Implement Euler's method to solve the equation numerically on the interval $t \in [0, 20]$ using step size $h = 1$. Compare your numerical results with the exact values from part 1.
3. Repeat the numerical solution using the Heun method and the classical Runge-Kutta method (RK4). Report and compare the results at $t = 5$, $t = 10$, and $t = 20$.
4. What do your numerical methods predict for large t? Does the population approach the expected limiting value K?

SOLUTION

1. It is worth defining the analytical solution as an R function and then use it to calculate the values at the suggested times.

```
# Solution of the logistic eqn
nsol <- function(t,r,K,n0) {
  nn <- K*n0*exp(r*t)/(K+n0*(exp(r*t)-1))

  return(nn)
}

# Given parameters
r <- 0.5
K <- 1000
n0 <- 50

# Growth at t=5, 10, 20
stmp <- sprintf("Population at t=5: %8.0f\n",nsol(5,r,K,n0))
cat(stmp)
#> Population at t=5:      391
stmp <- sprintf("Population at t=10: %8.0f\n",nsol(10,r,K,n0))
cat(stmp)
#> Population at t=10:      887
stmp <- sprintf("Population at t=20: %8.0f\n",nsol(20,r,K,n0))
cat(stmp)
#> Population at t=20:      999
```

2.

```
# Gradient
f <- function(t,n,r,K) {
  ff <- r*n*(1-n/K)

  return(ff)
}

# Interval
t0 <- 0
tf <- 20

# Initial conditions
n0 <- 50

# Step size
h <- 1

# Euler method
ltmp <- EulerODE(f=f,t0=t0,tf=tf,y0=n0,h=h,r=r,K=K)

# Output all values as a table
tble <- data.frame(t=ltmp$t,n=ltmp$y)
print(tble)
#>     t                n
#> 1   0  50.000000000000
#> 2   1  73.750000000000
#> 3   2 107.905468750000
#> 4   3 156.036408031921
#> 5   4 221.880931732130
#> 6   5 308.205823665036
#> 7   6 414.813320627032
#> 8   7 536.184935455736
#> 9   8 660.530260678768
#> 10  9 772.645278381971
#> 11 10 860.477554469980
#> 12 11 920.505520831651
#> 13 12 957.093074306702
#> 14 13 977.626035017126
#> 15 14 988.562720354036
#> 16 15 994.215954494168
#> 17 16 997.091249655877
#> 18 17 998.541394413656
#> 19 18 999.269633441700
#> 20 19 999.634550003195
#> 21 20 999.817208224748
```

The solution at $t = 5, 10, 20$ is not exactly the correct one, which is not surprising, given the nature of the approximation. Note how the parameters `r` and `K` could be passed by `EulerODE` to the gradient function because *ellipses*, `...`, were appropriately included in the `comphy` code.

3. The same calculations can be repeated using the Heun and 4th-order Runge-Kutta methods.

```
# Gradient already defined

# Heun
```

```
ltmpH <- HeunODE(f=f,t0=t0,tf=tf,y0=n0,h=h,r=r,K=K)

# 4th order Runge-Kutta
ltmpR <- RK4ODE(f=f,t0=t0,tf=tf,y0=n0,h=h,r=r,K=K)

# Comparison (Here ltmpH$t=ltmpR$t)
tble <- data.frame(t=ltmpR$t,nH=ltmpH$y,nR=ltmpR$y)
print(tble)
#>     t                nH                 nR
#> 1   0  50.000000000000  50.0000000000000
#> 2   1  78.952734375000  79.8366356662027
#> 3   2 122.636384391976 125.1347827495155
#> 4   3 185.861939649447 190.8087815779571
#> 5   4 271.973150869278 279.9273035192167
#> 6   5 379.812229096771 390.5841362450072
#> 7   6 501.199501616220 513.7772720932244
#> 8   7 622.217608757104 635.3190056707775
#> 9   8 729.113422621463 741.7455890534044
#> 10  9 814.115999534009 825.6298471122348
#> 11 10 876.466366369451 886.4321171353380
#> 12 11 919.679919913194 927.8802779937581
#> 13 12 948.522955846895 954.9674739641816
#> 14 13 967.312503858973 972.1849735070400
#> 15 14 979.365539273086 982.9367910529999
#> 16 15 987.022518563223 989.5769020672612
#> 17 16 991.857221311849 993.6496521243686
#> 18 17 994.898263398532 996.1371900116618
#> 19 18 996.806517786607 997.6526095170644
#> 20 19 998.002157352485 998.5743617332377
#> 21 20 998.750598964682 999.1344811993246
```

The values produced with the Heun and 4th-order Runge-Kutta methods are closer to the correct values than those produced with the Euler method.

4. Clearly, the analytical expression $n(t)$ goes to K when $t \to \infty$. Let us check this is also the case with the numerical solution provided by, say, `RK4ODE`. It will suffice to extend tf to a large value, say 1000.

```
# Just change tf
tf <- 1000

# Apply RK4
ltmpR <- RK4ODE(f,t0,tf,n0,h,r=r,K=K)

# Only show last 10 values
# (992 to 1001 - because the first value is 0)
tble <- data.frame(t=ltmpR$t[992:1001],
                   nR=ltmpR$y[992:1001])
print(tble)
#>       t   nR
#> 1   991 1000
#> 2   992 1000
#> 3   993 1000
#> 4   994 1000
#> 5   995 1000
#> 6   996 1000
```

```
#> 7   997 1000
#> 8   998 1000
#> 9   999 1000
#> 10 1000 1000
```

The numerical algorithm behaves well because it displays the asymptotic value. In fact, this is reached quite soon during growth.

8.1.4 Exercise 04

Consider the classical *Lotka–Volterra system*, modelling the interaction between a prey population $x(t)$ and a predator population $y(t)$. The governing equations are:

$$\mathrm{d}x/\mathrm{d}t = \alpha x - \beta xy$$
$$\mathrm{d}y/\mathrm{d}t = \delta xy - \gamma y$$

The meaning of the variables and parameters is as follows:

- $x(t)$: number of *prey* (e.g., rabbits) at time t
- $y(t)$: number of *predators* (e.g., foxes) at time t
- α: natural *growth rate of prey* in the absence of predators
- β: *predation rate coefficient* (how often predators encounter and eat prey)
- δ: *growth rate of predators* per prey eaten
- γ: *natural death rate of predators* in the absence of prey

The task to be carried out in this exercise are:

1. Implement a gradient function `f(t,u)` where `u=c(x,y)` and `f` returns the derivatives $\mathrm{d}x/\mathrm{d}t$ and $\mathrm{d}y/\mathrm{d}t$.
2. Use the `RK4ODE` solver to solve the system numerically over the interval $t \in [0, 30]$, where the parameters are
$$\alpha = 1.0, \quad \beta = 0.1, \quad \delta = 0.075, \quad \gamma = 1.5,$$
and the initial conditions are
$$x(0) = 40, \quad y(0) = 9.$$
Use a step size $h = 0.1$.
3. Plot both populations as functions of time.
4. Plot the *phase portrait*, i.e., a plot of $y(t)$ versus $x(t)$.
5. Try changing the parameters and observe whether the solution remains periodic or tends to a steady state.

SOLUTION

1. It is convenient to implement the gradient with generic parameters, as this can be then used for as many models as needed.

```
# Gradient (two ODEs)
f <- function(t,u,alpha,beta,gamma,delta) {
  dx <- alpha*u[1]-beta*u[1]*u[2]
  dy <- delta*u[1]*u[2]-gamma*u[2]

  return(c(dx,dy))
}
```

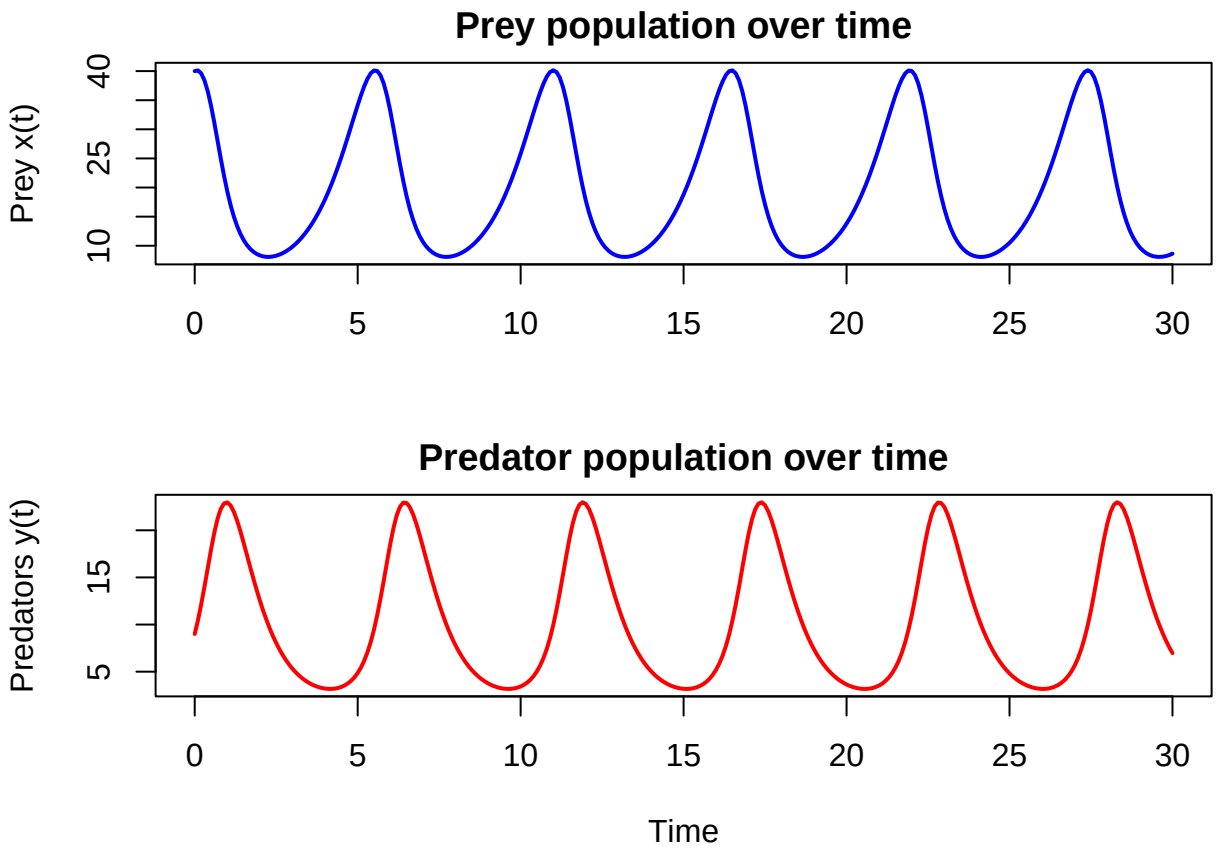
```r
# Test
print(f(t=0,u=c(40,9),alpha=1,beta=0.1,gamma=1.5,delta=0.075))
#> [1]  4.0 13.5
```

2. Solution using `RK4ODE`.

```r
# Gradient already prepared

# Use assigned values of parameters
alpha <- 1.0
beta <- 0.1
gamma <- 1.5
delta <- 0.075

# Solution interval
t0 <- 0
tf <- 30

# Initial conditions
u0 <- c(40,9)

# Step size
h <- 0.1

# 4th order Runge-Kutta
ltmp <- RK4ODE(f,t0,tf,u0,h,
               alpha=alpha,beta=beta,gamma=gamma,delta=delta)

# Find out number of steps
n <- length(ltmp$t)-1

# Check initial and final values
print(ltmp$t[1])
#> [1] 0
print(ltmp$t[n+1])
#> [1] 30.0000000000002
print(ltmp$y[1,])
#> [1] 40  9
print(ltmp$y[n+1,])
#> [1] 8.63239964622188 6.97165225927498
```

3. Let's plot the populations one on top of the other, for better comparison.

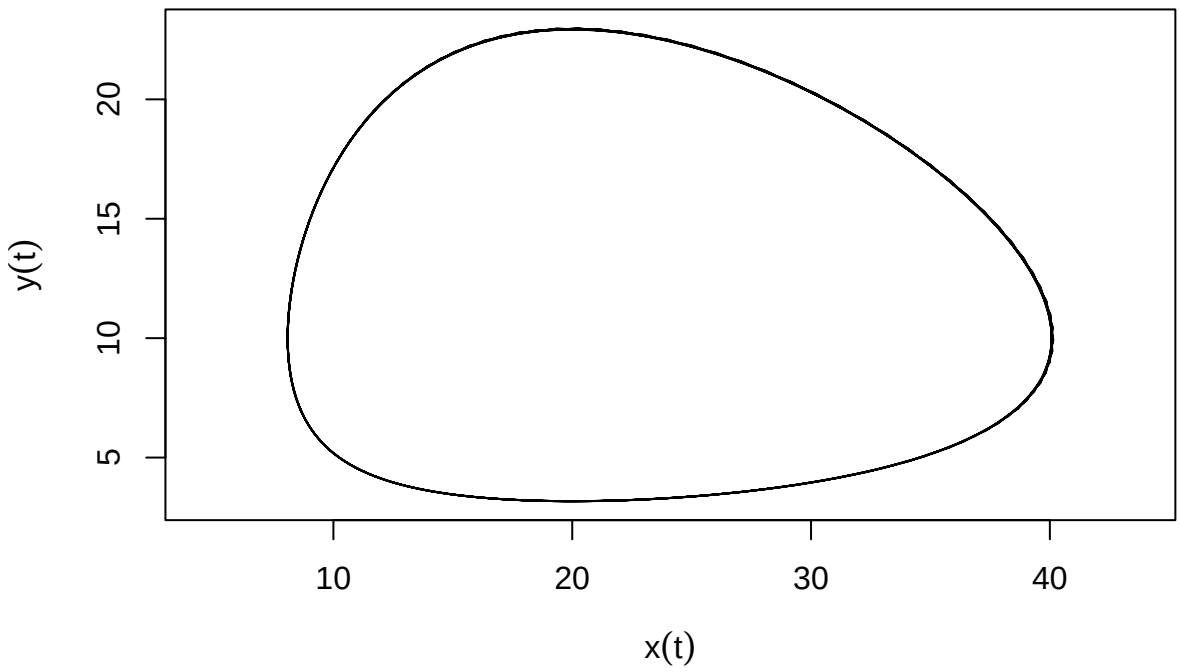
```r
# Two stacked plots
par(mfrow=c(2, 1),mar=c(4,4,2,1))

# Prey plot (at the top)
plot(ltmp$t,ltmp$y[,1],type="l",col="blue",lwd=2,
     xlab="",ylab="Prey x(t)",
     main="Prey population over time")

# Predator plot (at the bottom)
plot(ltmp$t,ltmp$y[,2],type="l",col="red",lwd=2,
     xlab="Time",ylab="Predators y(t)",
     main="Predator population over time")
```



We can admire in the plots above the typical dynamics of the predator-prey model. First, abundance of preys happens because there were not many predators to hunt and kill preys. But then prey abundance means that more food is available for predators and they can prosper. But too many predators means competition and less food (preys) available. So predators starve and begin to die. Then preys can breed more easily, their number starts increasing, and the cycle repeats.

4. The phase portrait is readily done by plotting $x(t)$ on the horizontal axis and $y(t)$ on the vertical one. These quantities were calculated in the previous snippet.

```
# Back to one graphical window
par(mfrow=c(1,1),mar=c(5.1,4.1,4.1,2.1))

# Phase portrait
plot(ltmp$y[,1],ltmp$y[,2],type="l",asp=1,
     xlab=expression(x(t)),ylab=expression(y(t)))
```

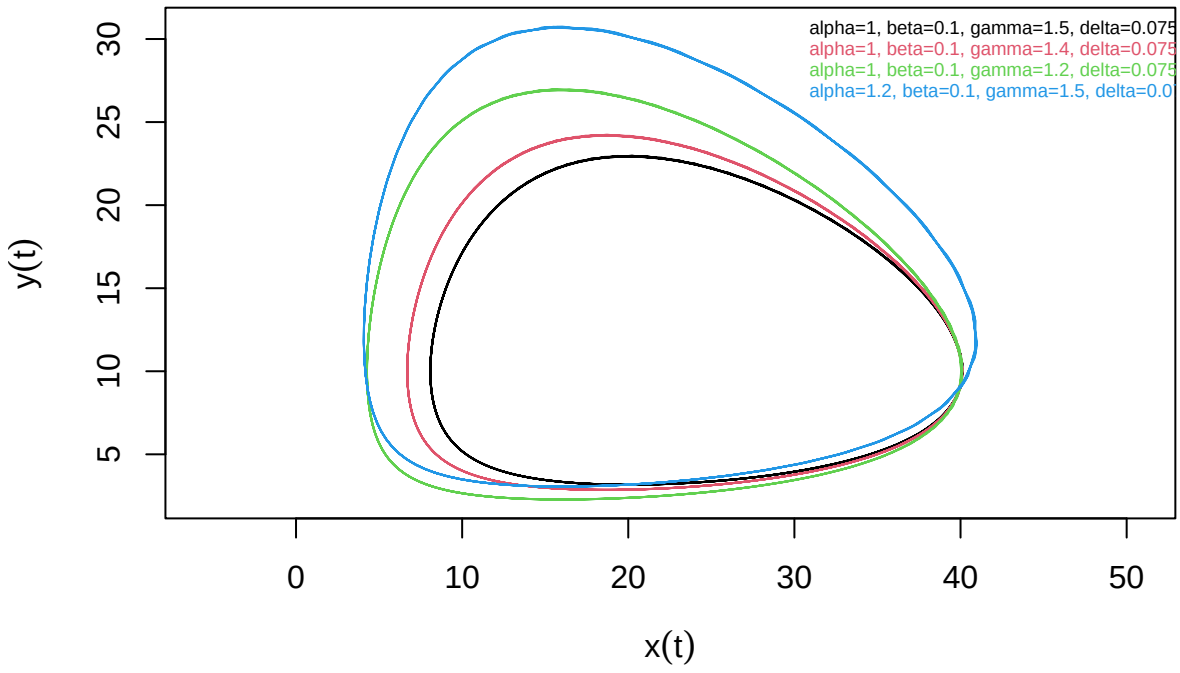


The closed orbit observed in the above phase portrait is the signature of a cyclic dynamics.

5. We can try and modify parameters slightly. The phase portrait, in our case will still contain closed orbits, but different from each other.

```
# First choice of parameters
ltmp0 <- ltmp

# Second choice of parameters
gamma <- 1.4

# Recalculate dynamics
ltmp1 <- RK4ODE(f,t0,tf,u0,h,
                alpha=alpha,beta=beta,gamma=gamma,delta=delta)

# Third choice of parameters
gamma <- 1.2

# Recalculate dynamics
ltmp2 <- RK4ODE(f,t0,tf,u0,h,
                alpha=alpha,beta=beta,gamma=gamma,delta=delta)

# Fourth choice of parameters
alpha <- 1.2

# Recalculate dynamics
ltmp3 <- RK4ODE(f,t0,tf,u0,h,
```

```
                alpha=alpha,beta=beta,gamma=gamma,delta=delta)

# Find extents of plot
rangeX <- range(ltmp0$y[,1],ltmp1$y[,1],ltmp2$y[,1],ltmp3$y[,1])
rangeY <- range(ltmp0$y[,2],ltmp1$y[,2],ltmp2$y[,2],ltmp3$y[,2])

# Phase portraits
plot(ltmp0$y[,1],ltmp0$y[,2],type="l",asp=1,
     xlab=expression(x(t)),ylab=expression(y(t)),
     xlim=rangeX,ylim=rangeY)
lines(ltmp1$y[,1],ltmp1$y[,2],col=2)
lines(ltmp2$y[,1],ltmp2$y[,2],col=3)
lines(ltmp3$y[,1],ltmp3$y[,2],col=4)

# Legend
legend(29,32,legend=c("alpha=1, beta=0.1, gamma=1.5, delta=0.075",
                      "alpha=1, beta=0.1, gamma=1.4, delta=0.075",
                      "alpha=1, beta=0.1, gamma=1.2, delta=0.075",
                      "alpha=1.2, beta=0.1, gamma=1.5, delta=0.075"),
       text.col=c(1,2,3,4),bty="n",cex=0.55)
```



The choice of values is obviously arbitrary. Changing these systematically can help connecting them with how they determine the dynamics of the model.

8.1.5 Exercise 05

In mechanics and engineering, the third derivative of position with respect to time is known as *jerk*, and it plays an important role in the modelling of motion where smoothness or mechanical stress is a concern. For instance, jerk arises in motion planning for vehicles or robotic arms, where sudden changes in acceleration must be avoided.

Consider the following third-order differential equation:

$$\frac{\mathrm{d}^3x}{\mathrm{d}t^3} + 4\frac{\mathrm{d}x}{\mathrm{d}t} + x = 0,$$

with initial conditions:

$$x(0) = 1, \quad \frac{\mathrm{d}x(0)}{\mathrm{d}t} = 0, \quad \frac{\mathrm{d}^2x(0)}{\mathrm{d}t^2} = 0.$$

In this exercise, you must:

1. Introduce new variables
$$y_1 = x, \quad y_2 = \frac{\mathrm{d}x}{\mathrm{d}t}, \quad y_3 = \frac{\mathrm{d}^2x}{\mathrm{d}t^2},$$
and rewrite the equation as a system of three coupled first-order differential equations.
2. Write down the corresponding initial conditions for the variables y_1, y_2, y_3.
3. Use a numerical method (e.g. Euler or Runge-Kutta of order 4) to solve the system on the interval $t \in [0, 10]$ with a step size of your choice.
4. Plot the numerical solution for position $x(t)$, velocity $\mathrm{d}x/\mathrm{d}t$, and acceleration $\mathrm{d}^2x/\mathrm{d}t^2$. Discuss the overall behaviour of the system.
5. What role does the jerk term (third derivative) appear to play in the motion over time?

SOLUTION

1. With the new suggested variables, the system is built in a cascading fashion. As $x = y_1$, the assignment

$$y_2 = \frac{\mathrm{d}x}{\mathrm{d}t}$$

becomes

$$\frac{\mathrm{d}y_1}{\mathrm{d}t} = y_2.$$

The assignment

$$y_3 = \frac{\mathrm{d}^2x}{\mathrm{d}t^2} = \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)$$

becomes

$$\frac{\mathrm{d}y_2}{\mathrm{d}t} = y_3.$$

The last equation of the system is found replacing all variables in the original ODE:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathrm{d}^2x}{\mathrm{d}t^2}\right) + 4\frac{\mathrm{d}x}{\mathrm{d}t} + x = 0$$

$$\Downarrow$$

$$\frac{\mathrm{d}y_3}{\mathrm{d}t} + 4y_2 + y_1 = 0 \;\Rightarrow\; \frac{\mathrm{d}y_3}{\mathrm{d}t} = -y_1 - 4y_2.$$

The original third order ODE is thus transformed into the following system of first order ODEs:

$$\begin{cases} \mathrm{d}y_1/\mathrm{d}t = y_2 \\ \mathrm{d}y_2/\mathrm{d}t = y_3 \\ \mathrm{d}y_3/\mathrm{d}t = -y_1 - 4y_2. \end{cases}$$

2. The associated initial conditions are:

$$y_1(0) = 1, \quad y_2(0) = 0, \quad y_3(0) = 0.$$

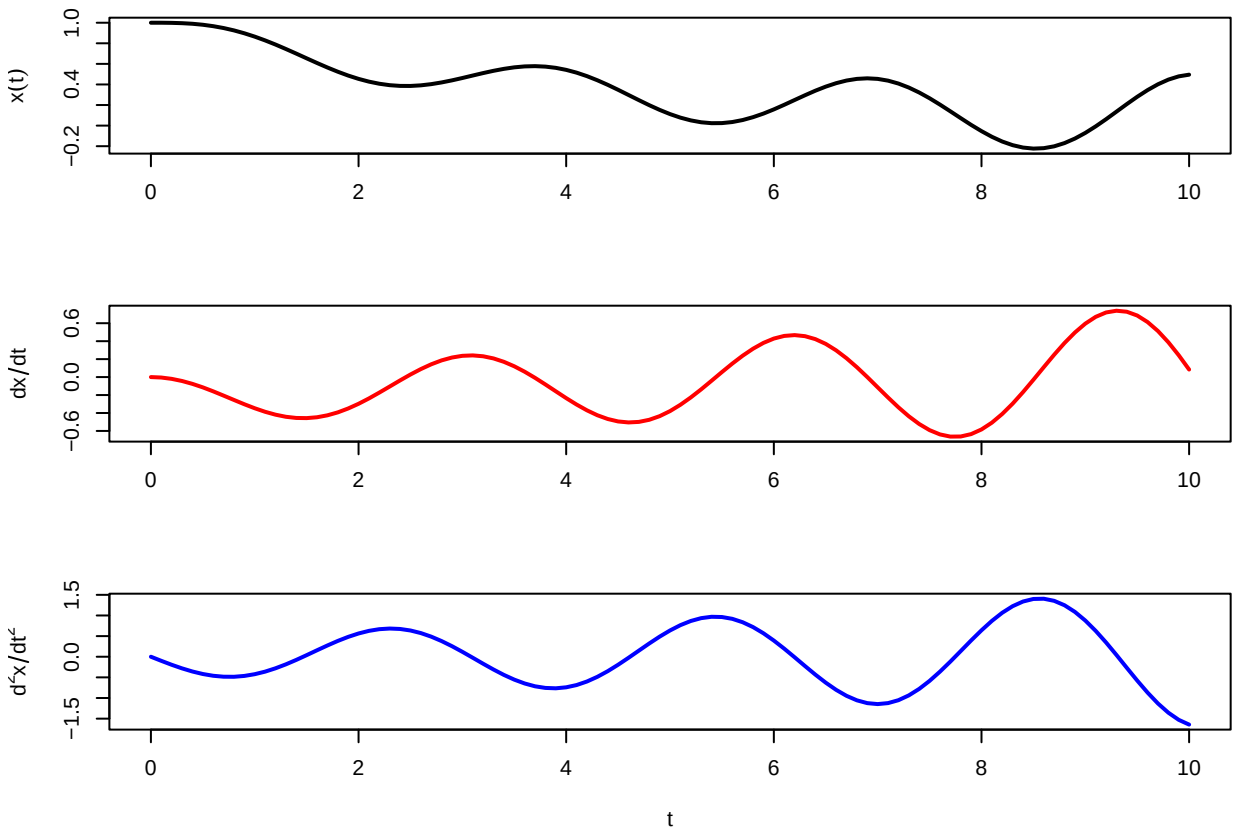
3. We apply the 4th-order Runge-Kutta method with step $h = 0.1$ to find the solutions for y_1, y_2, y_3 in the suggested range, $t \in [0, 10]$.

```
# Gradient
f <- function(t,y) {
  dy1 <- y[2]
  dy2 <- y[3]
  dy3 <- -y[1]-4*y[2]

  return(c(dy1,dy2,dy3))
}

# Solution interval
t0 <- 0
tf <- 10

# Initial conditions
y0 <- c(1,0,0)

# Step size
h <- 0.1

# Solver
ltmp <- RK4ODE(f,t0,tf,y0,h)

# Check
print(ltmp$t[1])
#> [1] 0
print(ltmp$y[1,])
#> [1] 1 0 0
```

The application seems to have been written in the right way. Always take care in counting the components of the initial conditions (three in this case).

4. Let's plot the three components on top of each other, for comparison.

```
# Three stacked plots
par(mfrow=c(3, 1),mar=c(4,4,2,1))

# x plot (at the top)
plot(ltmp$t,ltmp$y[,1],type="l",col="black",lwd=2,
     xlab="",ylab=expression(x(t)))
plot(ltmp$t,ltmp$y[,2],type="l",col="red",lwd=2,
     xlab="",ylab=expression(dx/dt))
plot(ltmp$t,ltmp$y[,3],type="l",col="blue",lwd=2,
     xlab=expression(t),ylab=expression(d^2 * x / dt^2))
```



The numerical solutions for position $x(t)$, velocity $\mathrm{d}x/\mathrm{d}dt$, and acceleration $\mathrm{d}^2x/\mathrm{d}t^2$ all exhibit a wavy, oscillatory behaviour over the interval $t \in [0, 10]$. This is consistent with the structure of the differential equation:

$$\frac{\mathrm{d}^3x}{\mathrm{d}t^3} + 4\frac{\mathrm{d}x}{\mathrm{d}t} + x = 0,$$

which combines inertial effects with a form of damping acting on the velocity.

- The *position* $x(t)$ oscillates around zero, with smooth curves and alternating concavity, indicating repeated acceleration and deceleration.
- The *velocity* $\mathrm{d}x/\mathrm{d}t$ also oscillates, with a phase shift relative to position. Its peaks and troughs align with points where position curvature changes.
- The *acceleration* $\mathrm{d}^2x/\mathrm{d}t^2$ shows sharper oscillations, reflecting the rapid changes in the rate of velocity due to the influence of jerk.

The plots reveal a system that does not settle to equilibrium or diverge, but instead exhibits sustained oscillations. The interplay of the damping and jerk terms produces motion that is smooth but highly dynamic.

5. The third derivative term, known as *jerk*, plays a critical role in shaping the system's dynamics. While a second-order equation would describe motion based solely on position and velocity, the presence of jerk introduces an additional layer of inertia, the system now responds not only to forces and velocities, but also to how rapidly the acceleration is changing. This manifests as:

- Smoother, more continuous transitions in motion,
- Greater resistance to abrupt changes in acceleration,
- More complex oscillatory behaviour, involving higher-order phase interactions between $x(t)$, $\mathrm{d}x/\mathrm{d}t$, and $\mathrm{d}^2x/\mathrm{d}t^2$.

Physically, such models are used in systems where motion needs to be smooth and controlled, for example, in robotics, vehicle suspension systems, or precision mechanisms. The oscillations seen in the plots reflect the fact that the system tries to damp motion through velocity, but is slowed down in how quickly it can adjust due to the influence of jerk.

8.2 Exercises on BVPs

8.2.1 Exercise 06

Consider the following linear BVP:

$$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} - \frac{2}{x}\frac{\mathrm{d}y}{\mathrm{d}x} + y(x) = x, \qquad y(1) = 0,\ y(2) = 1.$$

1. Use BVPlinshoot2 to compute the solution on $[1, 2]$. Try with step sizes $h = 0.01$, $h = 0.005$, $h = 0.0025$ and record the values of $y(1.5)$.
2. Plot the solution for the three different step sizes, using different colours/line traits.

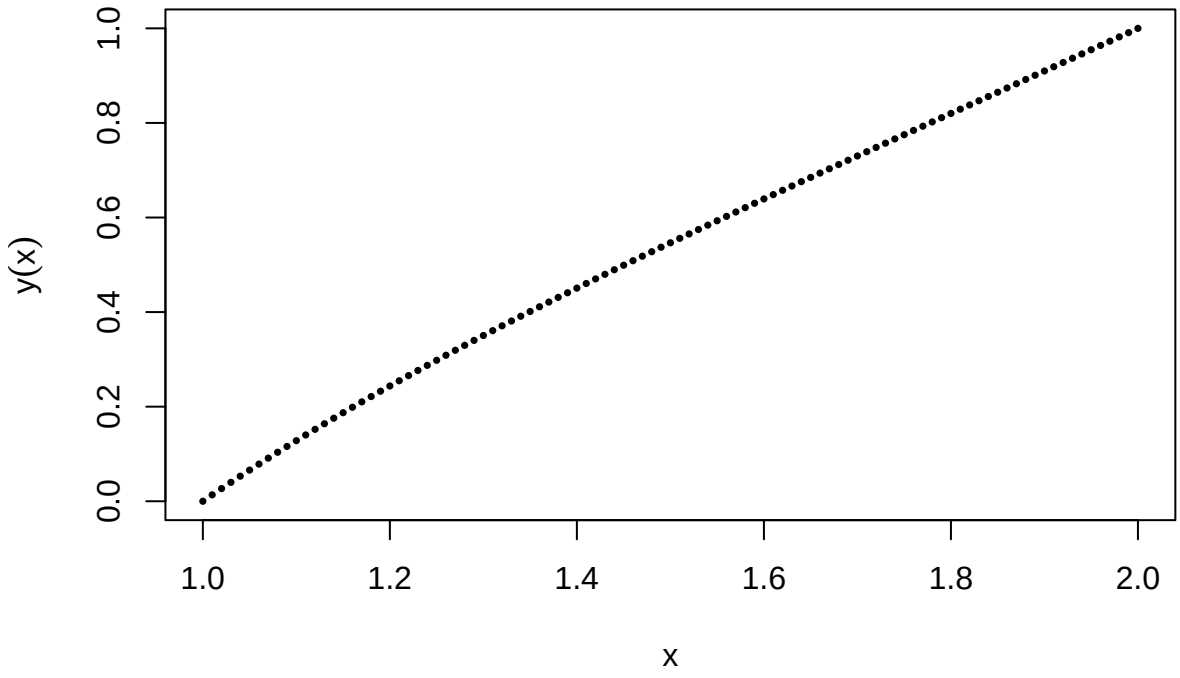
SOLUTION

1. The function $f(t, y, y')$ needed in `BVPlinshoot2` uses x rather than t:

$$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} = f(x, y, y') = x - y + \frac{2}{x}y'.$$

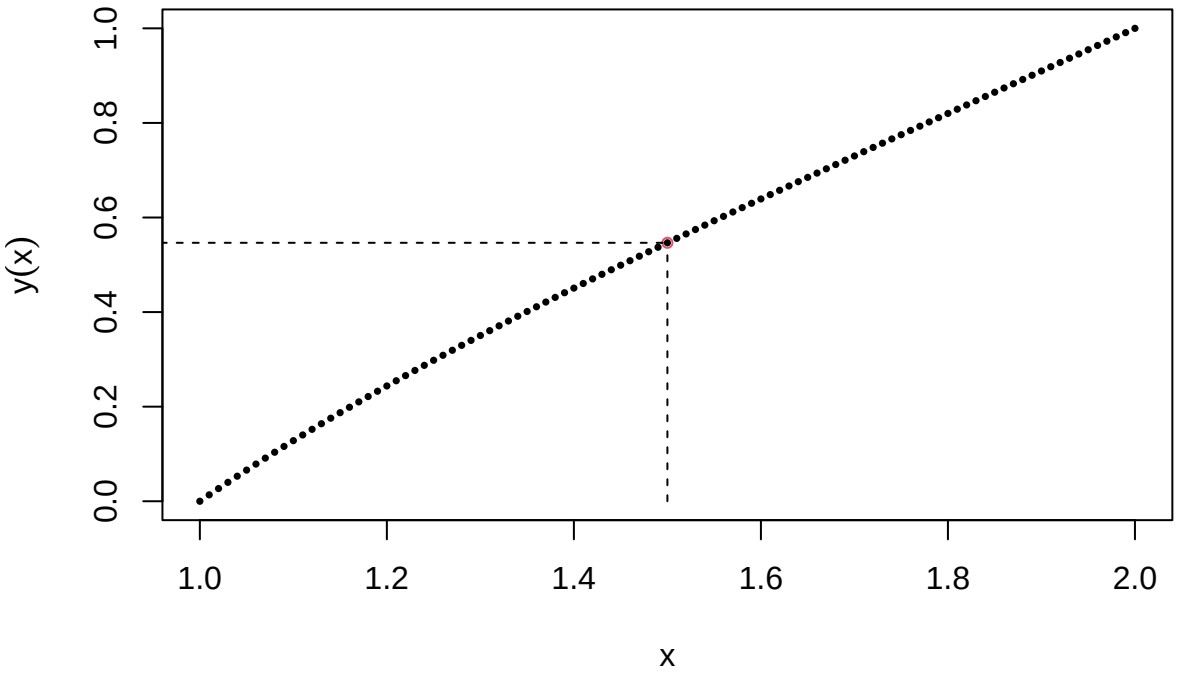
The values of x are away from 0, so we do not have to worry about the presence of $2/x$ in the equation. Let us try with $h = 0.01$ first.

```
# Make sure comphy is loaded in memory
require(comphy)

# Define "gradient" f(x,y,y')
f <- function(x,y,y1) {
  ff <- x-y-2*y1/x

  return(ff)
}

# Interval
x0 <- 1
xf <- 2

# BVPs
y0 <- 0
yf <- 1

# Step size
hA <- 0.01

# Solution. Linear shooting method
ltmpA <- BVPlinshoot2(f,x0,xf,y0,yf,hA)

# Plot
plot(ltmpA$t,ltmpA$y[,1],type="b",pch=16,cex=0.5,
     xlab=expression(x),ylab=expression(y(x)))
```



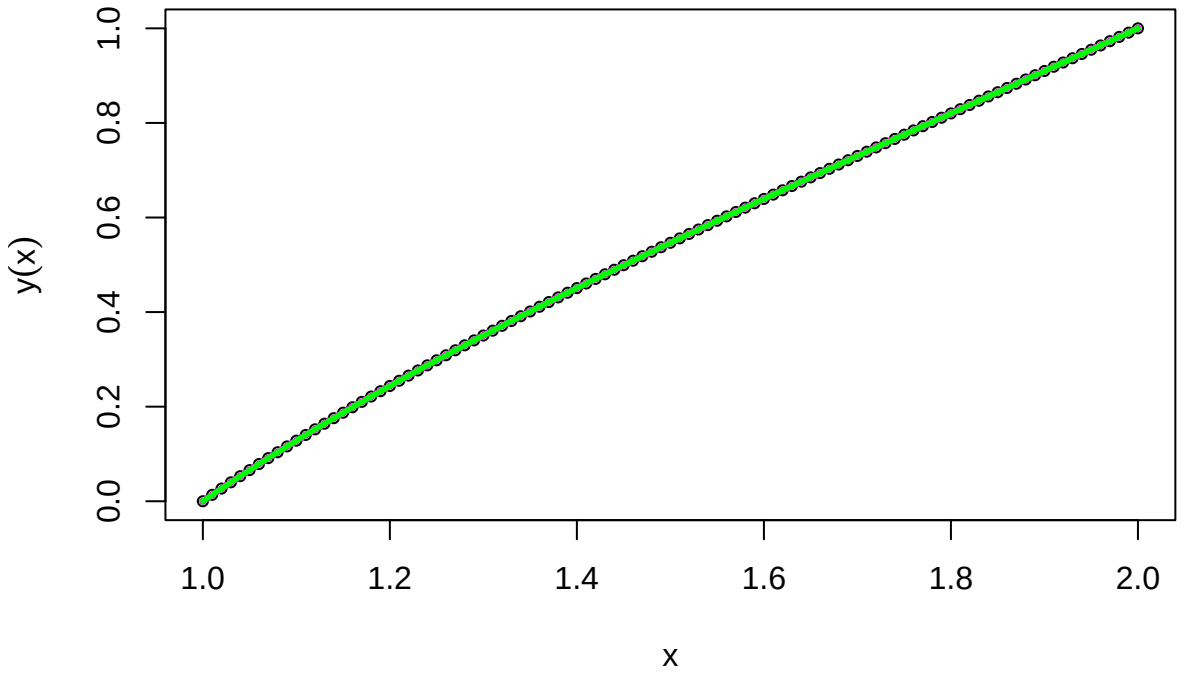
To extract the value at $x = 1.5$ we identify the position corresponding to 1.5 in the array `ltmpA$t`.

```
# Which index corresponds to x=1.5?
idxA <- which(abs(ltmpA$t-1.5) < 0.000001)
xA15 <- ltmpA$t[idxA]
print(idxA)
#> [1] 51
print(xA15)
#> [1] 1.5


# Value y(1.5) requested
yA15 <- ltmpA$y[idxA,1]
print(yA15) # y_combined is the name assigned to first column by BVPlinshoot2
#>        y_combined
#> 0.5464979975703993

# Indicate point in the plot
plot(ltmpA$t,ltmpA$y[,1],type="b",pch=16,cex=0.5,
     xlab=expression(x),ylab=expression(y(x)))
points(xA15,yA15,pch=1,col=2,cex=0.7)
segments(x0=xA15,x1=xA15,y0=0,y1=yA15,lty=2)
segments(x0=0,x1=xA15,y0=yA15,y1=yA15,lty=2)
```



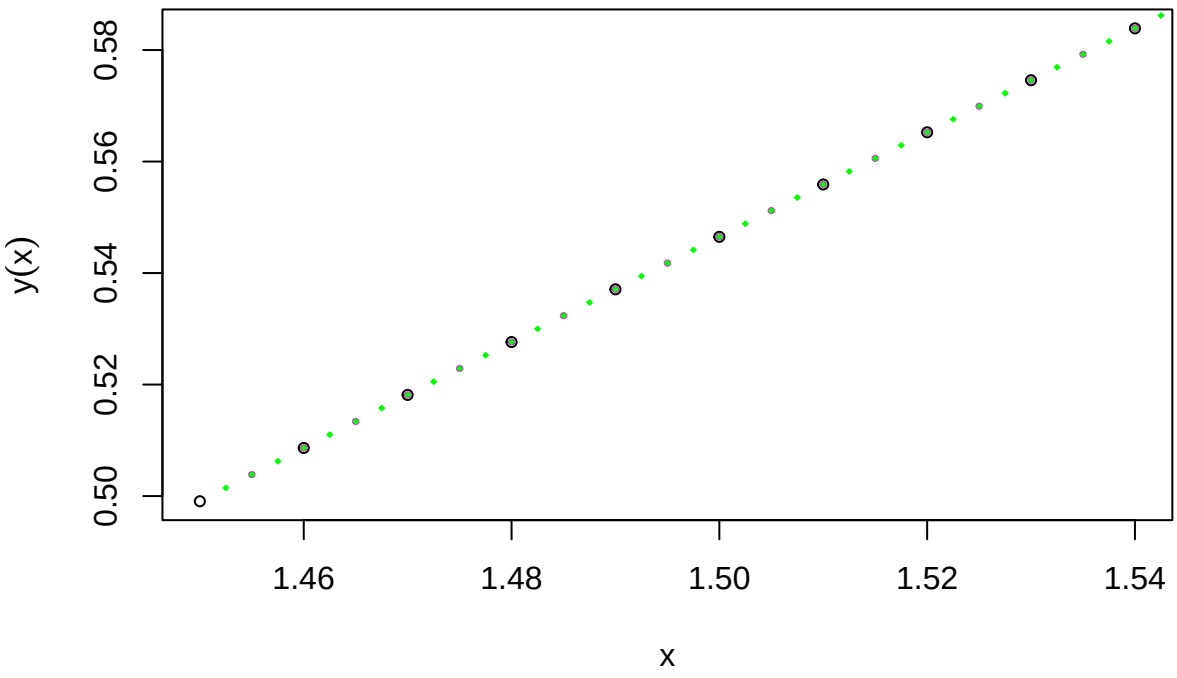
We can now repeat solutions for the other two step sizes and carry out a comparison of results.

```
# Step size 0.005
hB <- 0.005

# Solution. Linear shooting method
ltmpB <- BVPlinshoot2(f,x0,xf,y0,yf,hB)

# Step size 0.0025
hC <- 0.0025

# Solution. Linear shooting method
ltmpC <- BVPlinshoot2(f,x0,xf,y0,yf,hC)

# Comparison
plot(ltmpA$t,ltmpA$y[,1],type="b",pch=1,cex=0.7,
     xlab=expression(x),ylab=expression(y(x)))
points(ltmpB$t,ltmpB$y[,1],type="b",pch=16,cex=0.5,col="magenta")
points(ltmpC$t,ltmpC$y[,1],type="b",pch=18,cex=0.5,col="green")
```



It is difficult to see how solutions compare, although it is clear that they are very close with each other. We can *zoom in* by restricting the x range to the interval $[1.45, 1.55]$.

```
# Identify indices corresponding to x in [1.4,1.6]
jdxA <- which(ltmpA$t >= 1.45 & ltmpA$t <= 1.55)
jdxB <- which(ltmpB$t >= 1.45 & ltmpB$t <= 1.55)
jdxC <- which(ltmpC$t >= 1.45 & ltmpC$t <= 1.55)

# Plot to compare
plot(ltmpA$t[jdxA],ltmpA$y[jdxA,1],pch=1,cex=0.7,
     xlab=expression(x),ylab=expression(y(x)))
points(ltmpB$t[jdxB],ltmpB$y[jdxB,1],pch=16,cex=0.5,col="magenta")
points(ltmpC$t[jdxC],ltmpC$y[jdxC,1],pch=18,cex=0.5,col="green")
```



It is clear that the numerical solutions are very close with each other, which means that already $h = 0.01$ is good enough to approximate the solution.

Let us conclude with comparing the values at $x = 1.5$.

```
# Which index corresponds to x=1.5?
idxB <- which(abs(ltmpB$t-1.5) < 0.000001)
xB15 <- ltmpB$t[idxB]
print(idxB)
#> [1] 101
print(xB15)
#> [1] 1.49999999999999
idxC <- which(abs(ltmpC$t-1.5) < 0.000001)
xC15 <- ltmpC$t[idxC]
print(idxC)
#> [1] 201
print(xC15)
#> [1] 1.49999999999999


# Value y(1.5) requested
yB15 <- ltmpB$y[idxB,1]
print(yB15)
#>        y_combined
#> 0.546497975770438
yC15 <- ltmpC$y[idxC,1]
print(yC15)
```

```
#>        y_combined
#> 0.5464979775774576

# Comparison
cmp <- cbind(yA15,yB15,yC15)
print(cmp)
#>                          yA15              yB15              yC15
#> y_combined 0.5464979775703993 0.5464979775770438 0.5464979775774576
```

The values at $x = 1.5$ coincide, which corresponds to the experience we have had with the accuracy of the linear shooting method in the text book. In fact ,even though we cannot compare the numerical results with the analytic solution in this case, we can calculate the *mean local error* (see main text book) between the numerical solutions with different setp sizes. Care must be taken, though, to match indices as each step size correspond to a different number of points.

```
# Tolerance for matching points
tol <- 1e-12

# Mean local error between A and B
# Matching points. Row i, Column j equal TRUE means there's a match
M <- abs(outer(ltmpA$t,ltmpB$t,`-`)) <= tol

# Transform in index pairs. Matrix with two columns: (i "matches" j)
pairs <- which(M,arr.ind=TRUE)

# Use columns
MLerrorAB <- mean(abs(ltmpA$y[pairs[,1],1]-ltmpB$y[pairs[,2],1]))
print(MLerrorAB)
#> [1] 5.22665401917059e-11

# The result should correspond with the difference between O(h^5)
OA <- hA^5
OB <- hB^5
DAB <- abs(OA-OB)
print(DAB)
#> [1] 9.6875e-11
```

As we can see we have an agreement in orders of magnitude. The student can repeat the same reasoning for the other two comparisons (between A and C and between B and C).

8.2.2 Exercise 07

Solve the following BVP analytically,

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = -e^x, \qquad y(0) = 0,\ y(1) = 0,$$

and compare your solution with the numerical solution obtained with the linear shooting method. Do errors behave according to expectation?

SOLUTION

This simple ODE can be integrated twice to yield

$$y(x) = -e^x + k_1 x + k_2,$$

where k_1, k_2 are integration constants to be found using the boundary conditions. This is readily done:

$$y(0) = 0 \ \Leftrightarrow\ -1 + k_2 = 0 \ \Rightarrow\ k_2 = 1,$$

$$y(1) = 0 \;\Leftrightarrow\; -e + k_1 + 1 = 0 \;\Rightarrow\; k_1 = e - 1.$$

The analytic solution is therefore:

$$y(x) = 1 + (e-1)x - e^x.$$

We will use this solution to compare the accuracy of the numerical solution.

Let us use a step size $h = 0.01$ in the linear shooting method. This should correspond to an accuracy $\mathrm{O}(h^5) = 10^{-10}$.

```
# Define "gradient" f(x,y,y')
f <- function(x,y,y1) {
  ff <- -exp(x)

  return(ff)
}

# Interval
x0 <- 0
xf <- 1

# BVPs
y0 <- 0
yf <- 0

# Step size
h <- 0.01

# Solution. Linear shooting method
ltmp <- BVPlinshoot2(f,x0,xf,y0,yf,h)

# Analytic solution at same grid points as the numerical solution
xx <- ltmp$t
yy <- 1+(exp(1)-1)*xx-exp(xx)

# Compare solutions using the mean local error
MLerror <- mean(abs(ltmp$y[,1]-yy))
print(MLerror)
#> [1] 1.45227234734679e-12
```

The mean local error does not exceed expectation, which means that the numerical solution is very close to the analytic one.

8.2.3 Exercise 08

A rod of length $L = 10$ m has its ends held at temperatures

$$T(0) = T_A = 300 \text{ K}, \qquad T(L) = T_B = 350 \text{ K}.$$

Assume steady one–dimensional conduction with

$$\kappa(T) = 1 + \alpha T, \qquad \alpha = 10^{-3} \text{ K}^{-1},$$

where κ is the variable *thermal conductivity* of the rod (measured in $\mathrm{Wm^{-1}K^{-1}}$) and where no internal heat sources are present. The steady state equation is

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(\kappa(T)\,\frac{\mathrm{d}T}{\mathrm{d}x}\right) = 0,$$

which, written in the form $\mathrm{d}^2T/\mathrm{d}x^2 = f(x, T, T')$, becomes

$$\frac{\mathrm{d}^2T}{\mathrm{d}x^2} = -\frac{\alpha}{1+\alpha T}\left(\frac{\mathrm{d}T}{\mathrm{d}x}\right)^2.$$

1. Define $f(x, T, T') = -\alpha/(1+\alpha T)\,(T')^2$ and solve on $[0, 10]$ with `BVPshoot2`, using $T_A = 300$, $T_B = 350$.
2. Compute and report the numerical value of $T(5)$.
3. Produce a line plot of $T(x)$ on $[0, 10]$.
4. Repeat with $\alpha = 0$ (constant conductivity). Plot both solutions together and comment briefly on how the temperature–dependent conductivity bends the profile away from the straight line.

SOLUTION

1,2.

```
# Define "gradient" f(x,T,T')
f <- function(x,T,T1,alpha) {
  ff <- -alpha/(1+alpha*T)*T1^2

  return(ff)
}

# Interval
x0 <- 0
xf <- 10

# BVPs
T0 <- 300
Tf <- 350

# Step size
h <- 0.01

# Solution. Nonlinear shooting method
ltmpN <- BVPshoot2(f,x0,xf,T0,Tf,h,alpha=0.001)

# Value of T in the middle of the rod
idxN <- which(abs(ltmpN$t-5) < 0.000001)
x5N <- ltmpN$t[idxN]
T5N <- ltmpN$y[idxN,1]
print(cbind(idxN,x5N,T5N))
#>      idxN              x5N              T5N
#> [1,]  501 4.99999999999994 325.235828074523
```

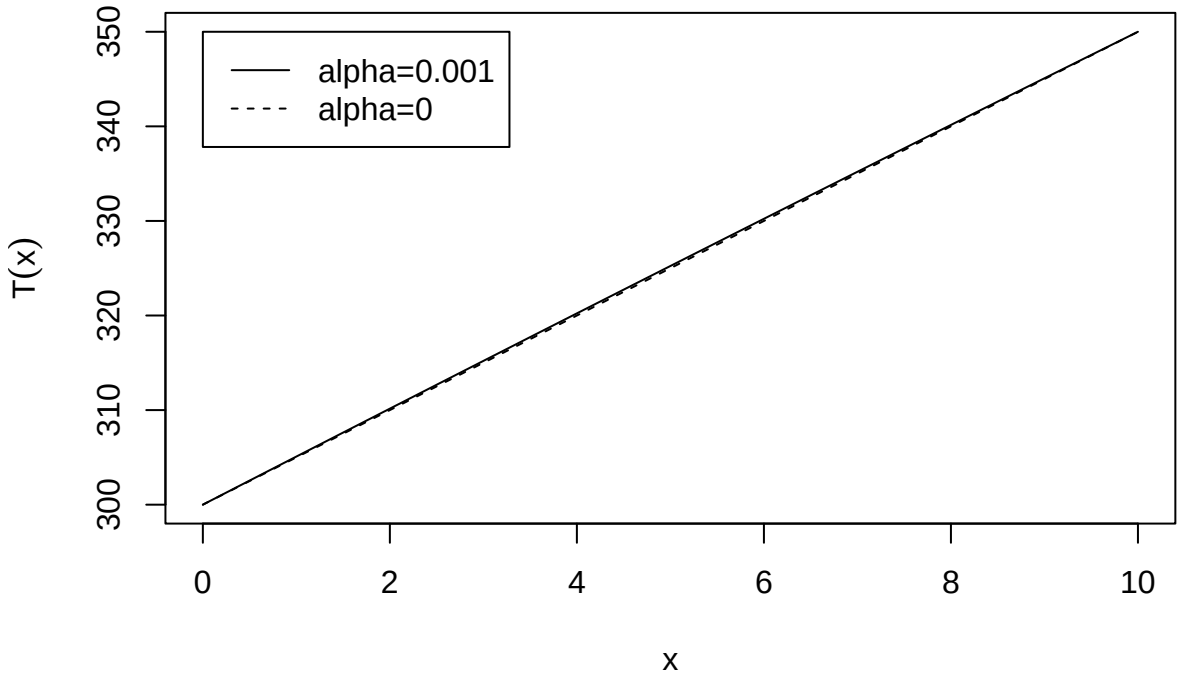
The value of the temperature in the steady state is slightly more than half way the temperature at the two extremes. It should be exactly equal to 325 if $\kappa(T) = 1$ ($\alpha = 0$):

```
# Solution. Nonlinear shooting method
ltmp0 <- BVPshoot2(f,x0,xf,T0,Tf,h,alpha=0)

# Value of T in the middle of the rod
idx0 <- which(abs(ltmp0$t-5) < 0.000001)
x50 <- ltmp0$t[idx0]
T50 <- ltmp0$y[idx0,1]
print(cbind(idx0,x50,T50))
```

```
#>      idx0              x50              T50
#> [1,]  501 4.9999999999994 324.999999995345
```

3,4. Let us compare the two solutions graphically.

```
# First plot alpha=0.001
plot(ltmpN$t,ltmpN$y[,1],type="l",
     xlab=expression(x),ylab=expression(T(x)))

# Second plot alpha=0
points(ltmp0$t,ltmp0$y[,1],type="l",lty=2)

# Legend
legend(x=0,y=350,legend=c("alpha=0.001","alpha=0"),lty=c(1,2))
```



There seems to be very little difference between the two curves. In fact, α is nudging the temperature profile in a minimal way, i.e. the profile is not significantly sensible to changes in α. We can reason quantitatively on this. First of all, the ODE states that a derivative is equal to zero; this means that its argument is a constant, say A:

$$\frac{\mathrm{d}}{\mathrm{d}x}\left((1+\alpha T)\frac{\mathrm{d}T}{\mathrm{d}x}\right)=0 \;\Rightarrow\; (1+\alpha T)\frac{\mathrm{d}T}{\mathrm{d}x}=A.$$

The differential equation is with separable variables:

$$(1+\alpha T)\mathrm{d}T=A\mathrm{d}x \;\Rightarrow\; T+\frac{\alpha}{2}T^2=Ax+B.$$

The two constants can be found using $T(0)=300$ and $T(10)=350$. It is then left as an exercise to verify

that this leads to:

$$\frac{\alpha}{2}T^2 + T - (1625\alpha + 5)x - \frac{\alpha}{2}300^2 - 300 = 0.$$

This second degree equation in the variable T has two roots and only one is physical because the temperature can never be negative:

$$T(x,\alpha) = \frac{-1 + \sqrt{(1+300\alpha)^2 + 2\alpha(1625\alpha+5)x}}{\alpha}, \quad \alpha \neq 0.$$

This is, in fact, the analytical solution of the BVP. We can use it, for example, to compare the effectiveness of the numerical method. But we can also use it to verify the variability of the curves for various values of α. For example, it is clear that the second degree equation becomes

$$T - 5x - 300 = 0 \;\Rightarrow\; T(x,0) = 300 + 5x,$$

when $\alpha \to 0$, which is the straight line profile already observed. But we can also think at how the physical root defining $T(x,\alpha)$ changes for $\alpha \to \infty$. The square root becomes the significant quantity at the numerator so that the whole expression is dominated by

$$\sqrt{\frac{300^2\alpha^2 + 3250\alpha^2 x}{\alpha^2}} = \sqrt{300^2 + 3250x}.$$

So, when $\alpha \to \infty$, the temperature profile becomes

$$T(x,\infty) = \sqrt{300^2 + 3250x}.$$

This still obeys the boundary conditions:

$$T(0,\infty) = \sqrt{300^2} = 300, \qquad T(10,\infty) = \sqrt{300^2 + 32500} = 350.$$

And, when $x = 5$ we have $T(5,\infty) = \sqrt{300^2 + 3250(5)} \approx 325.96$. We can verify this using a large value for α in the numerical solution:

```
# Large alpha (infty)
ltmp <- BVPshoot2(f,x0,xf,T0,Tf,h,alpha=1000)

# Value of T in the middle of the rod
idx <- which(abs(ltmp$t-5) < 0.000001)
x5 <- ltmp$t[idx]
T5 <- ltmp$y[idx,1]
print(cbind(idx,x5,T5))
#>      idx               x5                T5
#> [1,] 501 4.99999999999994 325.960117312265
```

The numerical value matches the theoretical one.

8.3 Exercises on EPs

8.3.1 Exercise 09

The stationary Schrödinger equation for a particle of mass m in one dimension is

$$-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2\psi}{\mathrm{d}x^2} + V(x)\psi = E\,\psi.$$

Consider the *infinite square well* potential

$$V(x) = \begin{cases} 0, & 0 < x < L, \\ +\infty, & \text{otherwise}, \end{cases} \qquad \psi(0) = 0,\ \psi(L) = 0.$$

Inside the well ($0 < x < L$), this reduces to the Sturm–Liouville problem

$$\frac{\mathrm{d}^2\psi}{\mathrm{d}x^2} + \lambda\,\psi = 0, \qquad \psi(0) = \psi(L) = 0,$$

with parameter $\lambda = 2mE/\hbar^2$ (λ has dimensions of inverse length squared, L^{-2}; physical energies are recovered via $E = (\hbar^2/2m)\lambda$).

It is known that the eigenpairs are

$$\lambda_n = \left(\frac{n\pi}{L}\right)^2, \quad \psi_n(x) = \sqrt{\frac{2}{L}}\sin\left(\frac{n\pi x}{L}\right), \quad E_n = \frac{\hbar^2\pi^2}{2mL^2}n^2, \qquad n = 1, 2, 3, \ldots$$

1. Use `EPSturmLiouville2` to compute the first four numerical eigenvalues and eigenfunctions on $[0, L]$ with $L = 1$.
2. Compare the numerical eigenvalues with the exact values $\lambda_n = (n\pi)^2$. Report the relative errors for $n = 1, \ldots, 4$.
3. Plot the first four eigenfunctions obtained numerically and overlay the exact sine functions for visual comparison.

SOLUTION

1. We have seen a similar EP in the text book. The Sturm–Liouville problem represented has

$$p(x) = 1,\ q(x) = 0,\ w(x) = 1.$$

It is therefore straightforward to set function `EPStrurmLiouville2` up as done in the following snippet. The stepsize can be $h = 0.01$, which means that $[0, 1]$ will have $n = 1 = 101$.

```
# Make sure comphy is loaded in memory
require(comphy)

# Define the interval and number of grid points
a <- 0
b <- 1
n <- 100
x <- seq(a,b,length.out=n+1)

# Define constant coefficient functions
p <- function(s) 1
q <- function(s) 0
w <- function(s) 1

# Solve the Sturm-Liouville eigenproblem (4 eigenvalues wanted)
ep <- EPSturmLiouville2(p,q,w,x,nev=4,
                        normalize=TRUE,return_matrices=TRUE)

# Print the eigenvalues
print(round(ep$values,3))
#> [1]   9.869  39.465  88.761 157.706
```

2. The theoretical values λ_n, given that $L = 1$, are $\lambda_n = n^2\pi^2$. Let us look at the numerical comparison and the relative error.

```
# Theoretical quantities
lbdas <- ((1:4)*pi)^2
```

```
# Compare numerical and theoretical eigenvalues
print(cbind(numerical=ep$values,theoretical=lbdas))
#>               numerical        theoretical
#> [1,]   9.86879268537808   9.86960440108936
#> [2,]  39.46543143456795  39.47841760435743
#> [3,]  88.76070793840336  88.82643960980423
#> [4,] 157.70597371044278 157.91367041742973

# Relative errors
Rers <- abs(lbdas-ep$values)/abs(lbdas)
print(cbind(n=1:4,RelErr=Rers))
#>      n               RelErr
#> [1,] 1 8.22439966477156e-05
#> [2,] 2 3.28943523512575e-04
#> [3,] 3 7.40001194347246e-04
#> [4,] 4 1.31525476190836e-03
```

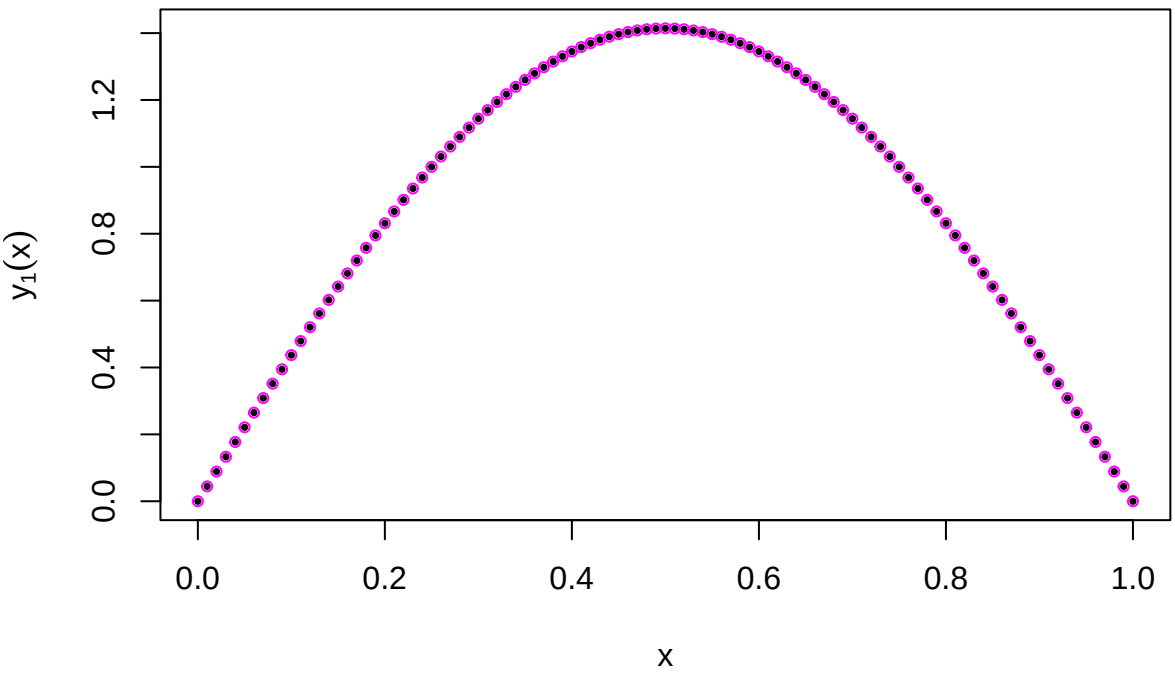
The relative errors increase with increasing n. This is to be expected (see main text).

3. The exact eigenfunctions are

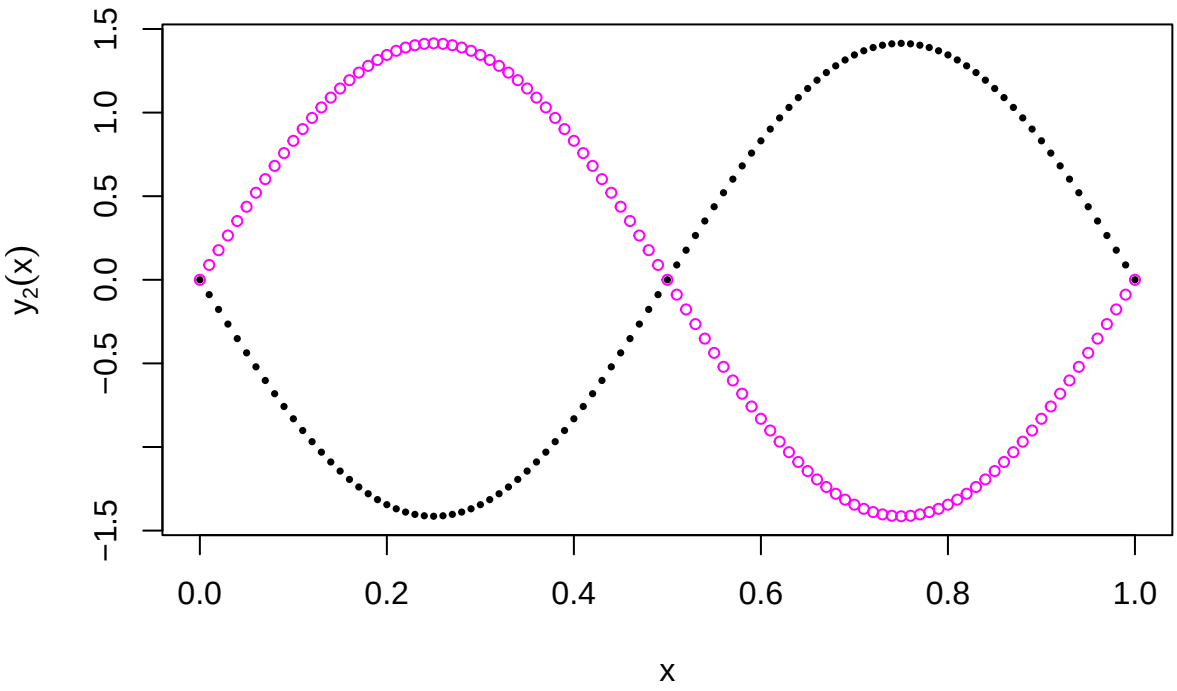
$$\psi_n(x) = \sqrt{2}\sin(n\pi x), \qquad n = 1, 2, 3, 4.$$

For the graphical comparison, let us look at the following code.

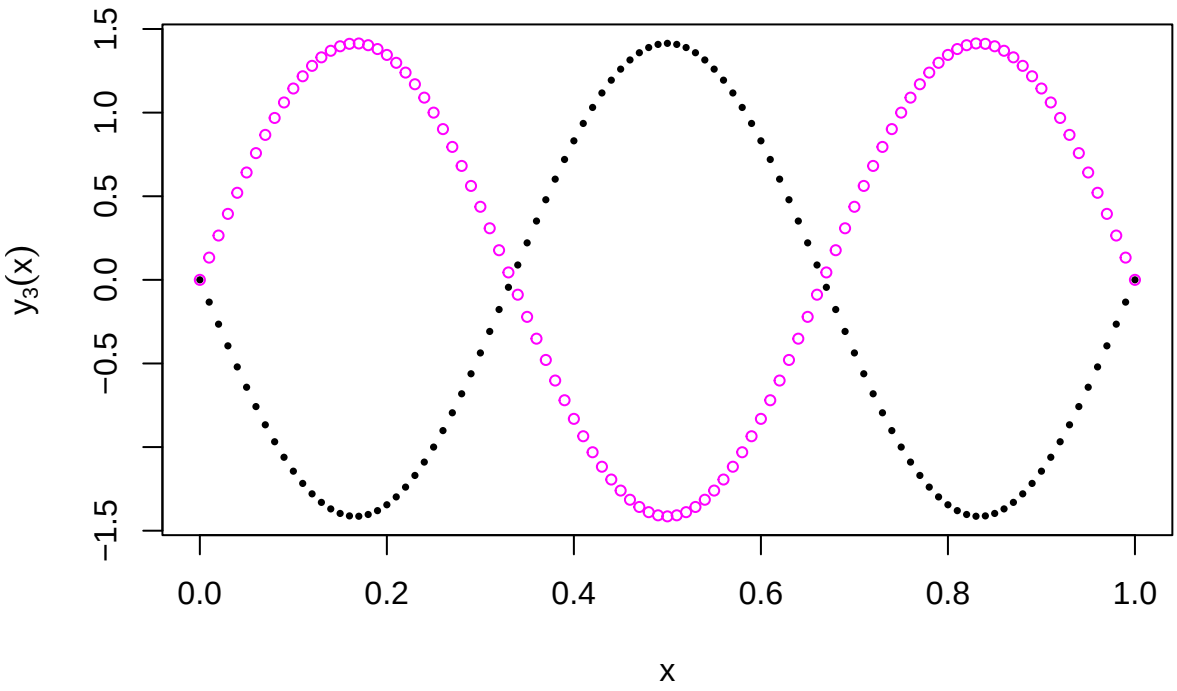
```
# Evaluate true eigenfunctions at n+1 points
xx <- seq(0,1,length.out=n+1)
yy1 <- sqrt(2)*sin(pi*xx)
yy2 <- sqrt(2)*sin(2*pi*xx)
yy3 <- sqrt(2)*sin(3*pi*xx)
yy4 <- sqrt(2)*sin(4*pi*xx)

# First comparison plot
plot(xx,ep$vectors_full[,1],type="b",pch=16,cex=0.5,
     xlab=expression(x),ylab=expression(y[1](x)))
points(xx,yy1,type="b",pch=1,col="magenta",cex=0.7)
```



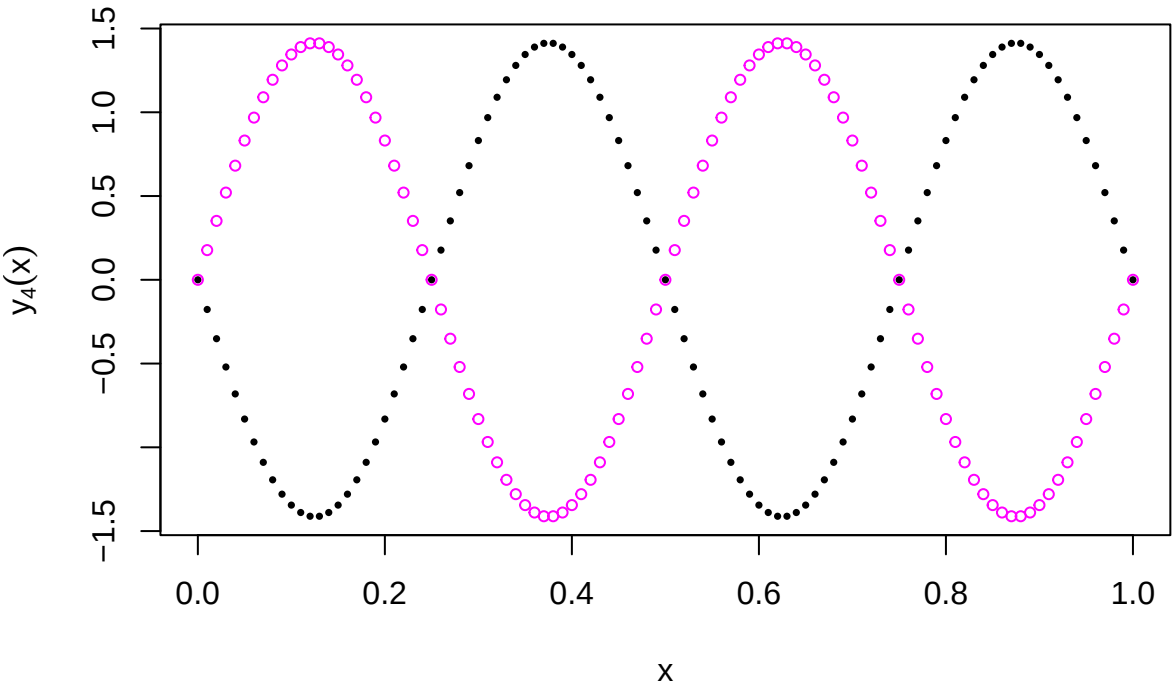

```
# Secons comparison plot
plot(xx,ep$vectors_full[,2],type="b",pch=16,cex=0.5,
     xlab=expression(x),ylab=expression(y[2](x)))
points(xx,yy2,type="b",pch=1,col="magenta",cex=0.7)
```




```
# Third comparison plot
plot(xx,ep$vectors_full[,3],type="b",pch=16,cex=0.5,
     xlab=expression(x),ylab=expression(y[3](x)))
points(xx,yy3,type="b",pch=1,col="magenta",cex=0.7)
```




```
# Fourth comparison plot
plot(xx,ep$vectors_full[,4],type="b",pch=16,cex=0.5,
     xlab=expression(x),ylab=expression(y[4](x)))
points(xx,yy4,type="b",pch=1,col="magenta",cex=0.7)
```



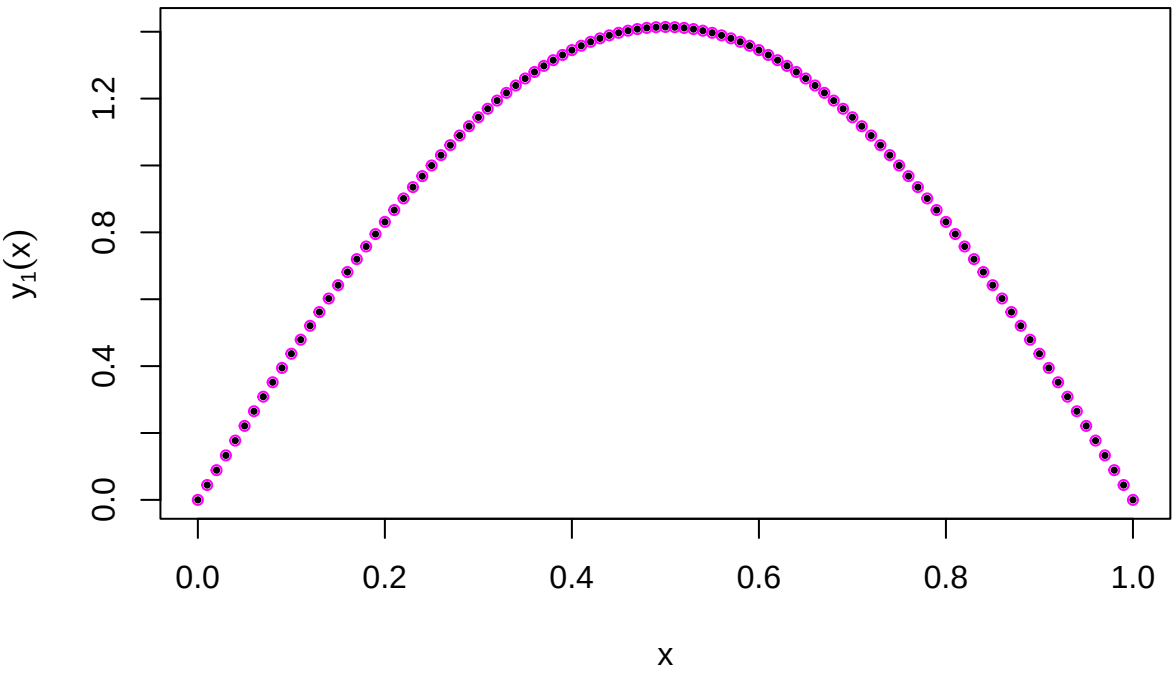
While the fundamental eigenfunctions matches exactly, the other modes appear "inverted'': why? Eigenvalue problems determine eigenfunctions only *up to a nonzero scalar multiple*. In our real, self–adjoint setting (Sturm–Liouville with homogeneous Dirichlet conditions and its standard centred–difference discretisation), each simple eigenpair (λ_k, ϕ_k) can be represented equally well by $(\lambda_k, -\phi_k)$. The continuous eigenfunctions are thus unique only up to a global sign, and the same holds for the discrete eigenvectors of the symmetric tridiagonal matrix.

Numerical eigensolvers enforce a norm (e.g. $\|\phi_k\| = 1$) but do not enforce a sign. Implementation details and roundoff decide whether the solver returns ϕ_k or $-\phi_k$. As a result, when we overlay numerical eigenfunctions with the analytic ones (e.g. $\sin(n\pi x)$ on $[0, 1]$), some curves may appear upside–down. This is not an error; physical quantities (like $|\psi|^2$) and orthogonality are unchanged by a global sign.

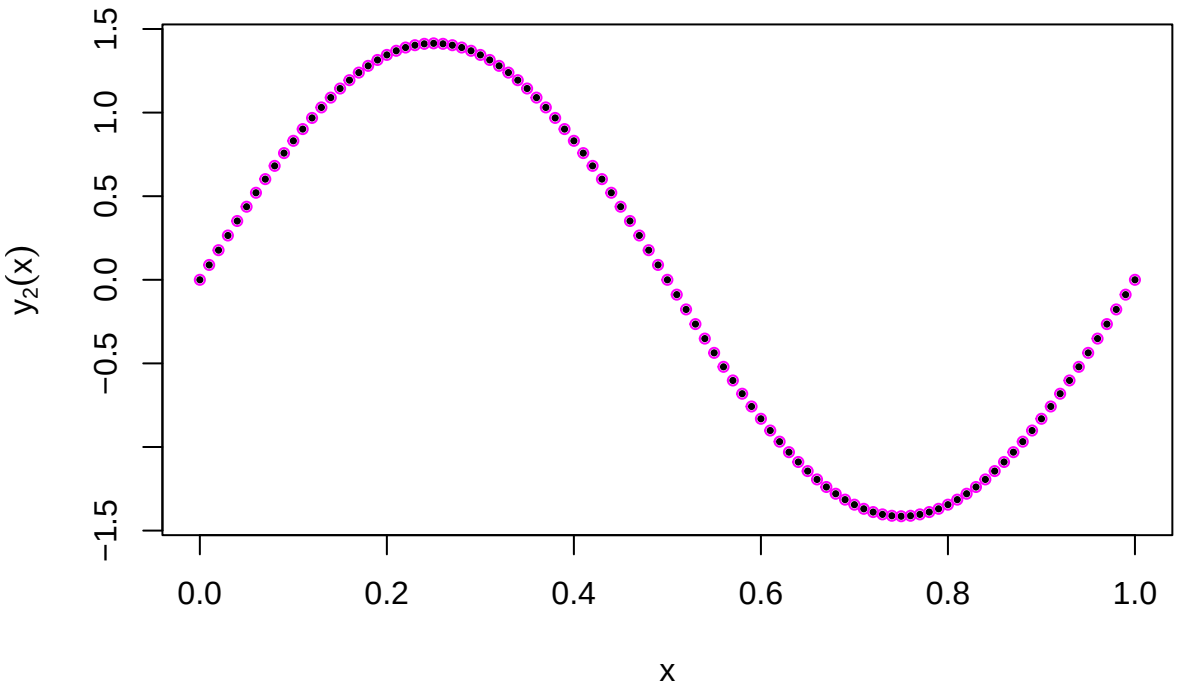
Thus, before plotting or comparing with a reference function ϕ_k^{ref}, one should choose the orientation of the numerical eigenvector $y^{(k)}$ by a sign that maximises their correlation with respect to the appropriate inner product. Equivalently, in the absence of a reference curve, one could enforce a simple rule such as "make the component of largest magnitude positive''. Either convention removes the apparent inversions and yields consistent plots across modes and mesh refinements. For the case under study we can, for example, use the following:

```
# Align by correlation with known eigenfunctions
Ytrue <- cbind(yy1,yy2,yy3,yy4)
V <- ep$vectors_full
for (k in 1:4) {
  if (sum(V[,k]*Ytrue[,k]) < 0) V[,k] <- -V[,k]
}

# Plot again
```

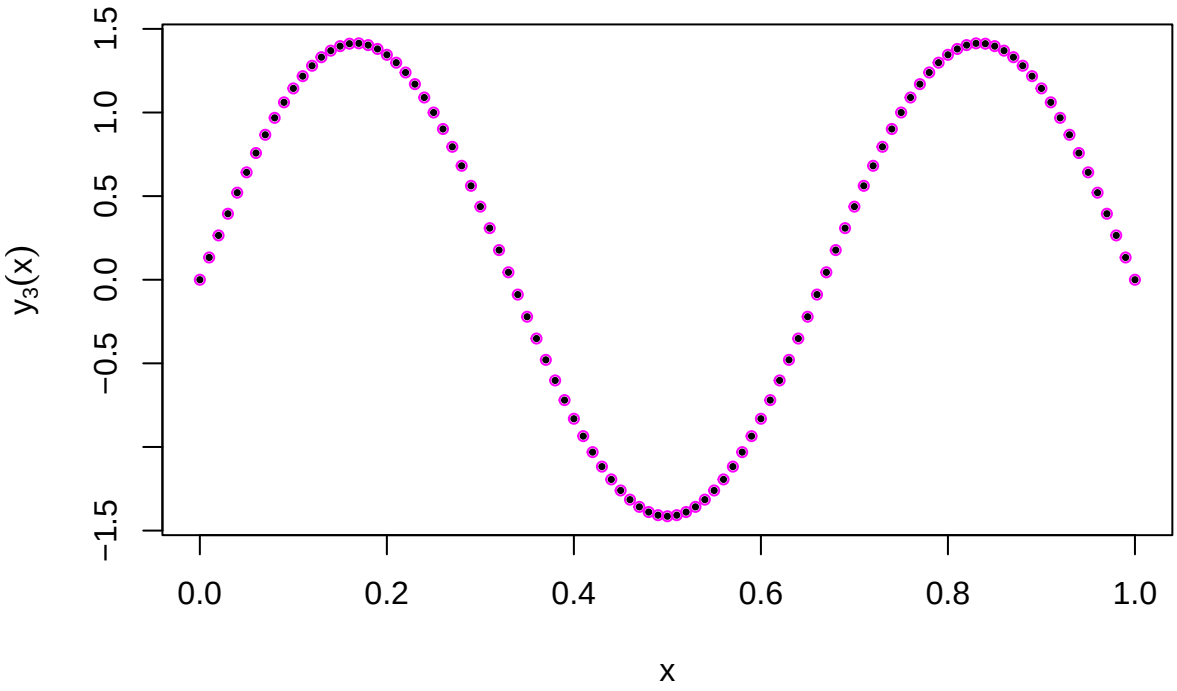
```r
# First comparison plot
plot(xx,V[,1],type="b",pch=16,cex=0.5,
     xlab=expression(x),ylab=expression(y[1](x)))
points(xx,yy1,type="b",pch=1,col="magenta",cex=0.7)
```



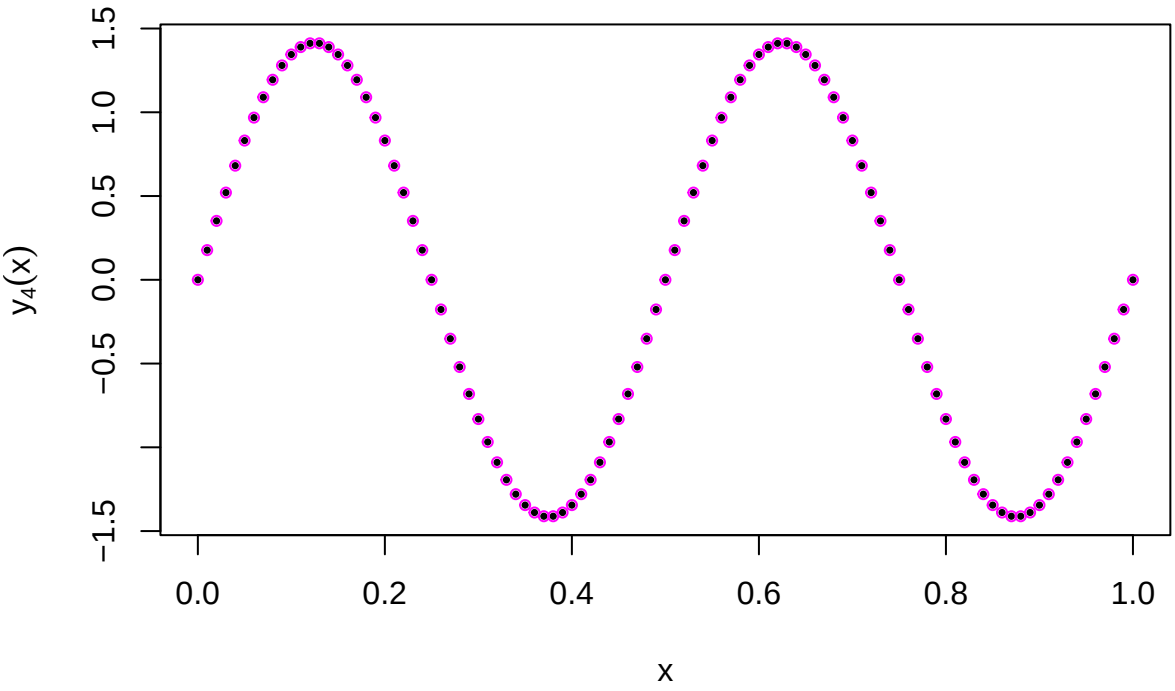

```r
# Secons comparison plot
plot(xx,V[,2],type="b",pch=16,cex=0.5,
     xlab=expression(x),ylab=expression(y[2](x)))
points(xx,yy2,type="b",pch=1,col="magenta",cex=0.7)
```




```
# Third comparison plot
plot(xx,V[,3],type="b",pch=16,cex=0.5,
     xlab=expression(x),ylab=expression(y[3](x)))
points(xx,yy3,type="b",pch=1,col="magenta",cex=0.7)
```




```
# Fourth comparison plot
plot(xx,V[,4],type="b",pch=16,cex=0.5,
     xlab=expression(x),ylab=expression(y[4](x)))
points(xx,yy4,type="b",pch=1,col="magenta",cex=0.7)
```



Remark. For problems with degenerate eigenvalues (not the case here), any orthonormal basis of the eigenspace is acceptable; numerical solvers may return different orthonormal combinations. A sign (or, in complex settings, a phase) alignment step against a chosen reference basis restores visual consistency.

8.3.2 Exercise 10

The BVP related to the *modified Legendre equation*,

$$-\frac{\mathrm{d}}{\mathrm{d}x}\left((1-x^2)\frac{\mathrm{d}y}{\mathrm{d}x}\right) = \lambda(1-x^2)y, \qquad y(-1) = y(1) = 0.$$

is a Sturm–Liouville problem with homogeneous Dirichlet conditions but nonconstant weight function:

$$p(x) = 1 - x^2, \qquad q(x) = 0, \qquad w(x) = 1 - x^2.$$

1. Use `EPSturmLiouville2` to find the first four eigenvalues and eigenvectors of the problem, using a step size $h = 0.01$.
2. Plot the eigenvectors in the interval $[-1, 1]$.
3. Repeat using a coarser grid with $h = 0.1$. Compare eigenvalues and eigenvectors with those of the first numerical solution.

SOLUTION

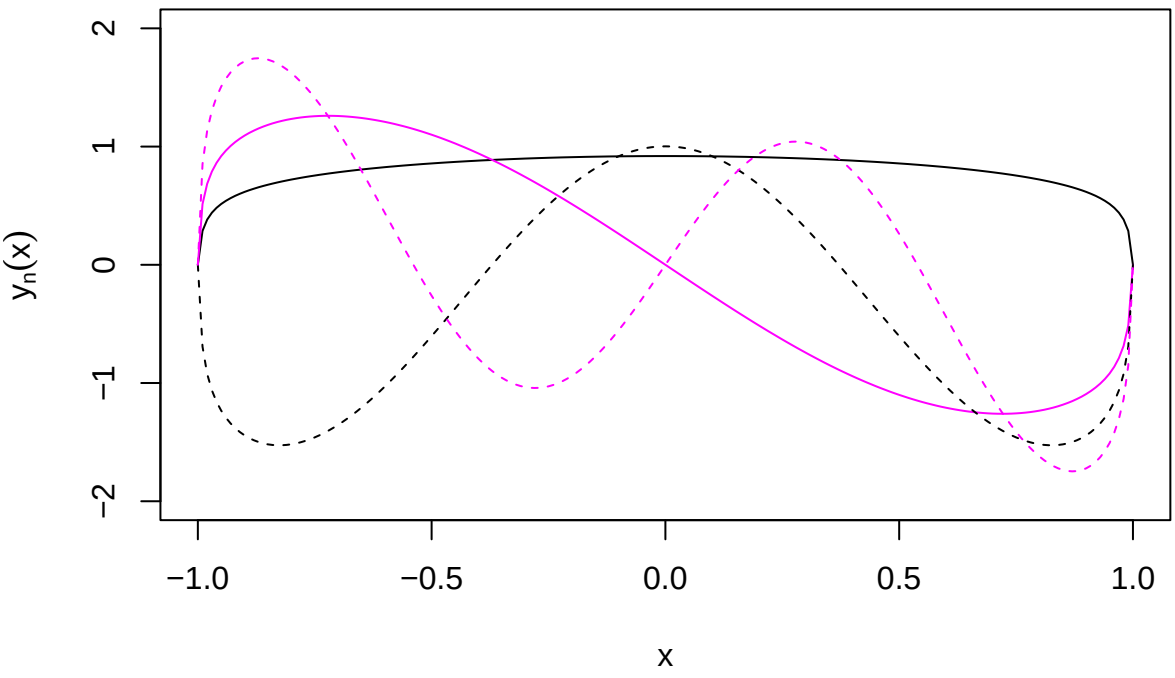
1. For the numerical solution we can follow the same structure used in the main text.

```
# Define the interval and grid (h=0.01)
a <- -1
b <- 1
```

```
h <- 0.01
x <- seq(a,b,by=h) # Should have n+1=201 grid points

# Define functions
p <- function(s) 1-s^2  # p(x)
q <- function(s) 0  # q(x)
w <- function(s) 1-s^2  # w(x)

# Solve the Sturm-Liouville eigenproblem
ep <- EPSturmLiouville2(p,q,w,x,nev=4,normalize=TRUE)

# Print the eigenvalues
print(round(ep$values,3))
#> [1]  0.497  6.125 16.924 32.774
```

2. We will plot the four eigenfunctions using a vertical range $[-2, 2]$, in a same window. An interesting observation is that not all modes are symmetric.

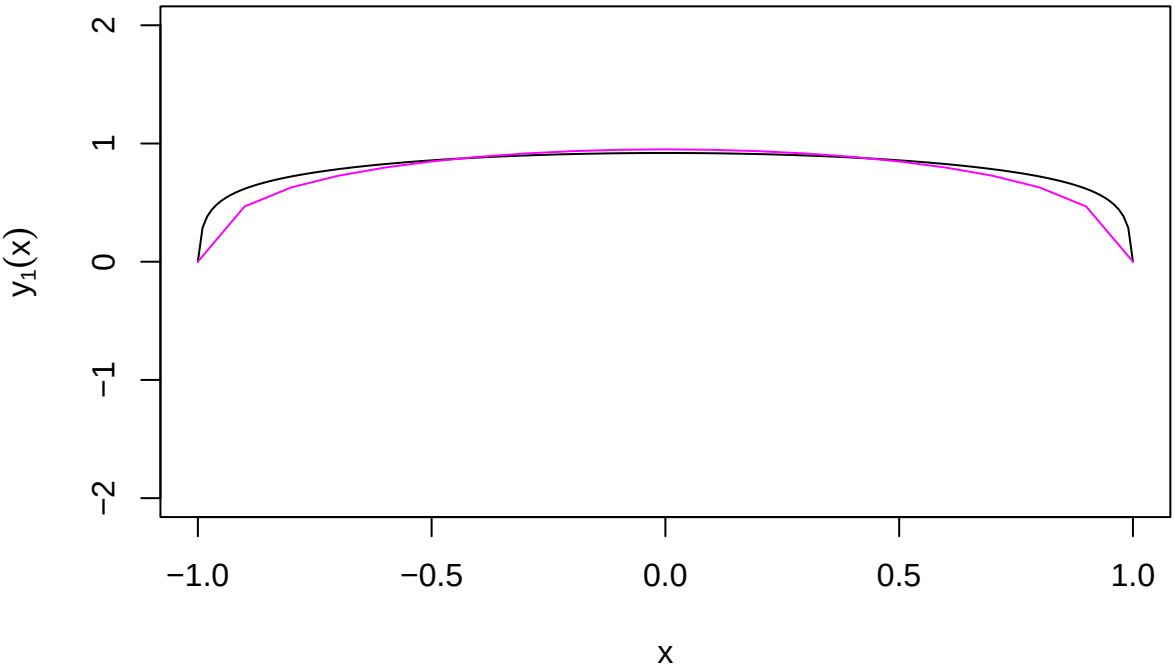
```
# Copy the four eigenfunctions unto separate vectors
y1 <- ep$vectors_full[,1]
y2 <- ep$vectors_full[,2]
y3 <- ep$vectors_full[,3]
y4 <- ep$vectors_full[,4]

# Plot using different colours/line traits
plot(x,y1,type="l",ylim=c(-2,2),
     xlab=expression(x),ylab=expression(y[n](x)))
points(x,y2,type="l",col="magenta")
points(x,y3,type="l",lty=2)
points(x,y4,type="l",col="magenta",lty=2)
```



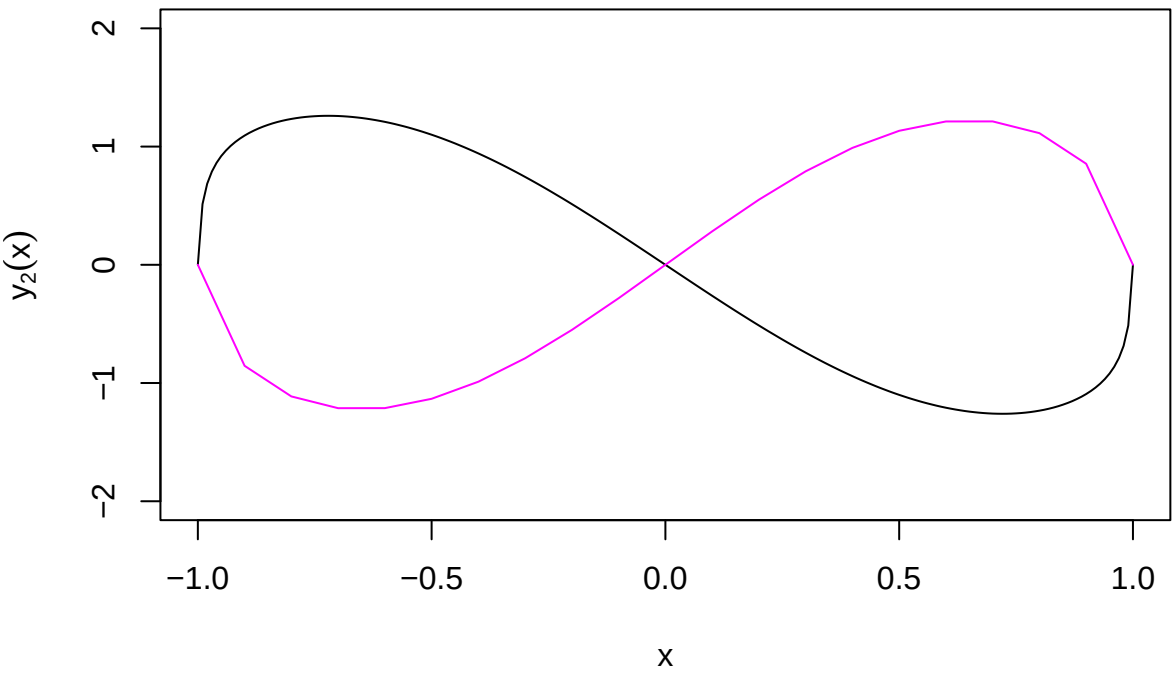
3. Use $h = 0.1$. The steps are the same and we can compare both eigenvalues (nuerically) and eigenvectors (graphically into four separate plots).

```
# Need only to modify the x grid
h <- 0.1
x2 <- seq(a,b,by=h) # Should have n+1=21 grid points

# Solve the Sturm-Liouville eigenproblem
ep2 <- EPSturmLiouville2(p,q,w,x2,nev=4,normalize=TRUE)

# Print the eigenvalues
print(round(ep2$values,3))
#> [1]  0.796  7.070 18.484 34.611

# Compare eigenvalues
cmp <- cbind(n=1:4,h001=ep$values,h01=ep2$values)
print(round(cmp,3))
#>      n   h001    h01
#> [1,] 1  0.497  0.796
#> [2,] 2  6.125  7.070
#> [3,] 3 16.924 18.484
#> [4,] 4 32.774 34.611

# Eigenfunctions
y2_1 <- ep2$vectors_full[,1]
y2_2 <- ep2$vectors_full[,2]
y2_3 <- ep2$vectors_full[,3]
```

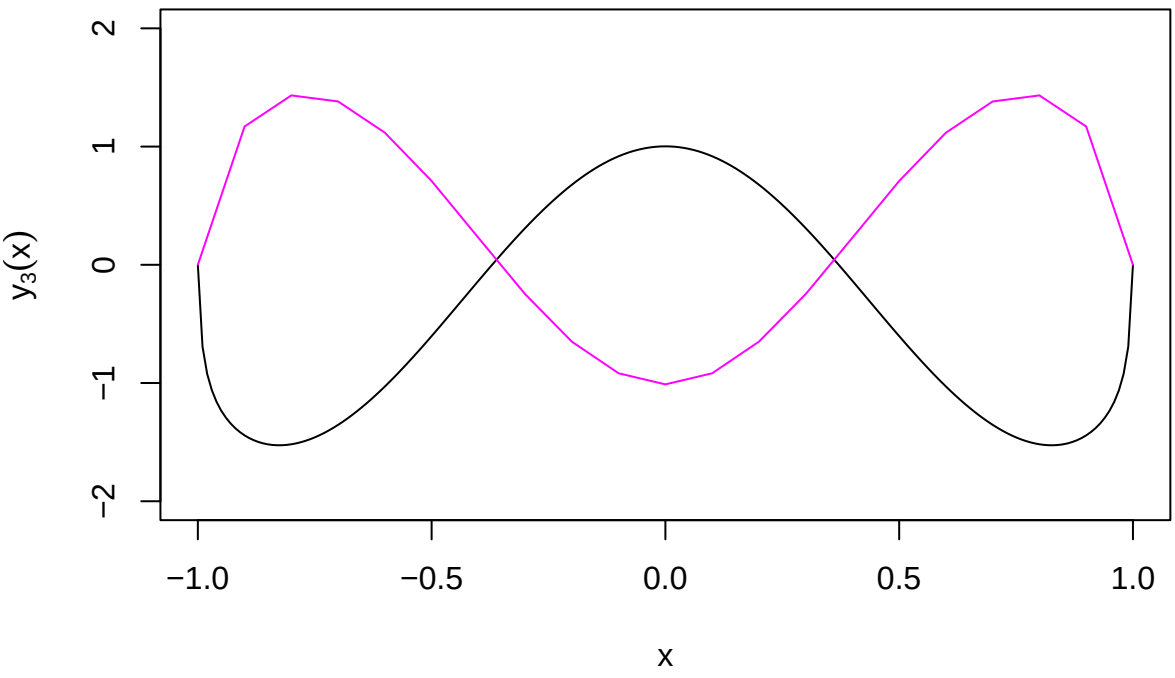
```
y2_4 <- ep2$vectors_full[,4]

# Compare eigenfunctions
plot(x,y1,type="l",ylim=c(-2,2),
     xlab=expression(x),ylab=expression(y[1](x)))
points(x2,y2_1,type="l",col="magenta")
```



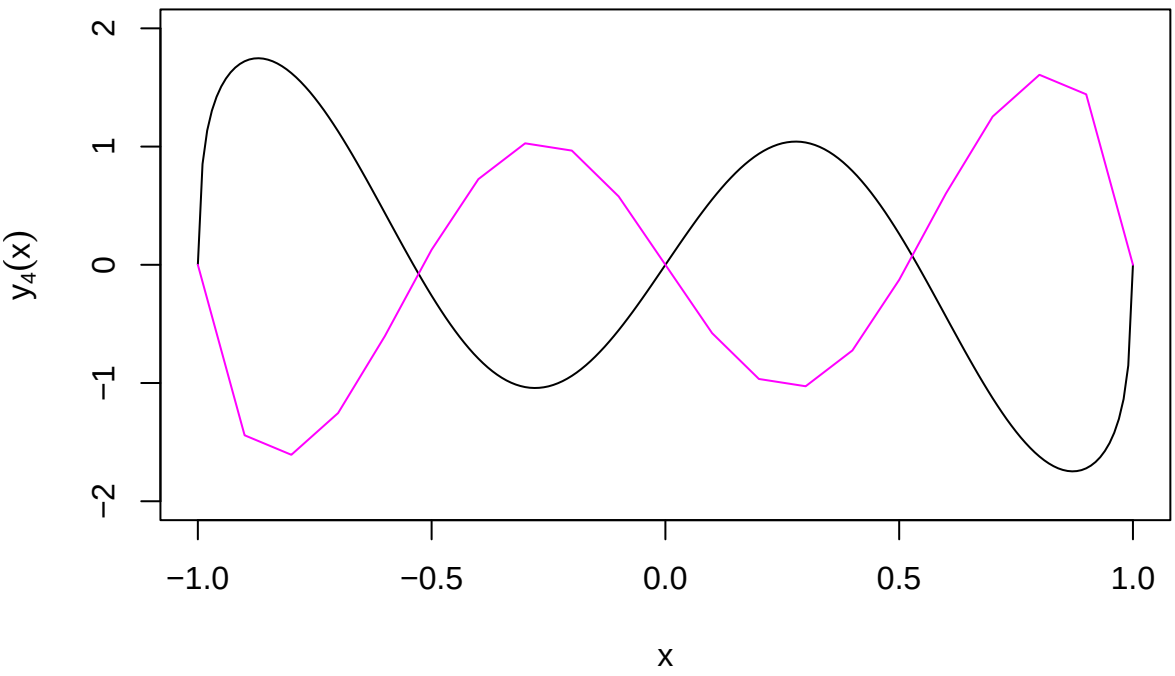

```
plot(x,y2,type="l",ylim=c(-2,2),
     xlab=expression(x),ylab=expression(y[2](x)))
points(x2,y2_2,type="l",col="magenta")
```




```
plot(x,y3,type="l",ylim=c(-2,2),
     xlab=expression(x),ylab=expression(y[3](x)))
points(x2,y2_3,type="l",col="magenta")
```

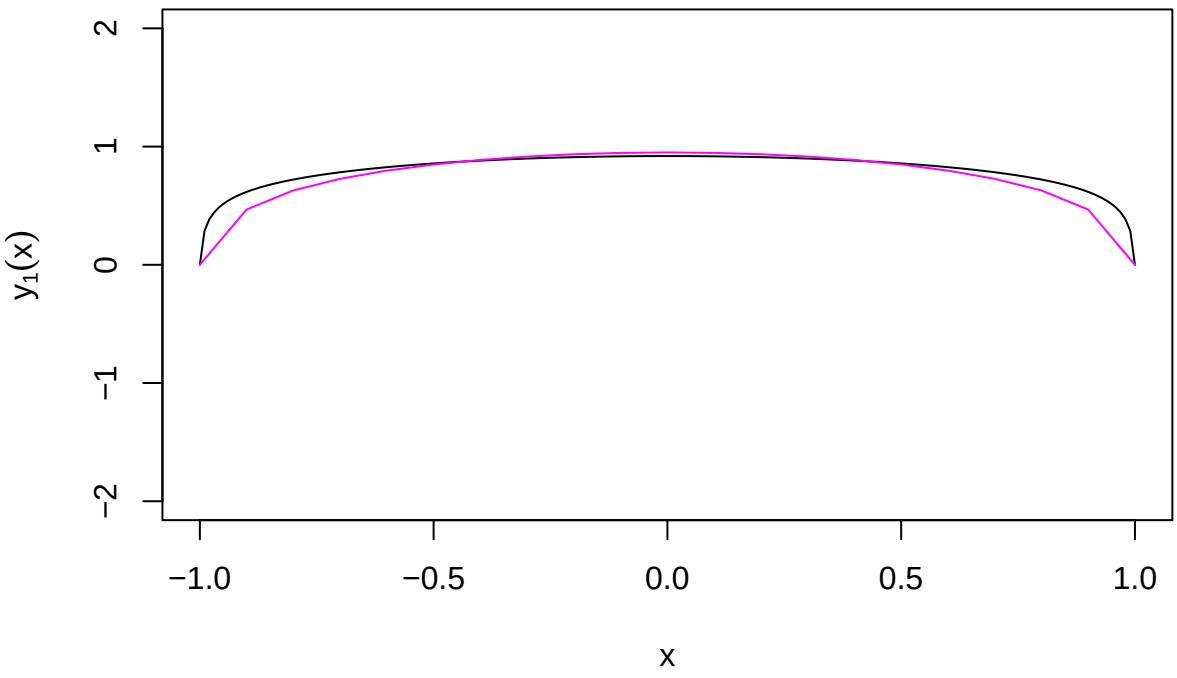



```
plot(x,y4,type="l",ylim=c(-2,2),
     xlab=expression(x),ylab=expression(y[4](x)))
points(x2,y2_4,type="l",col="magenta")
```

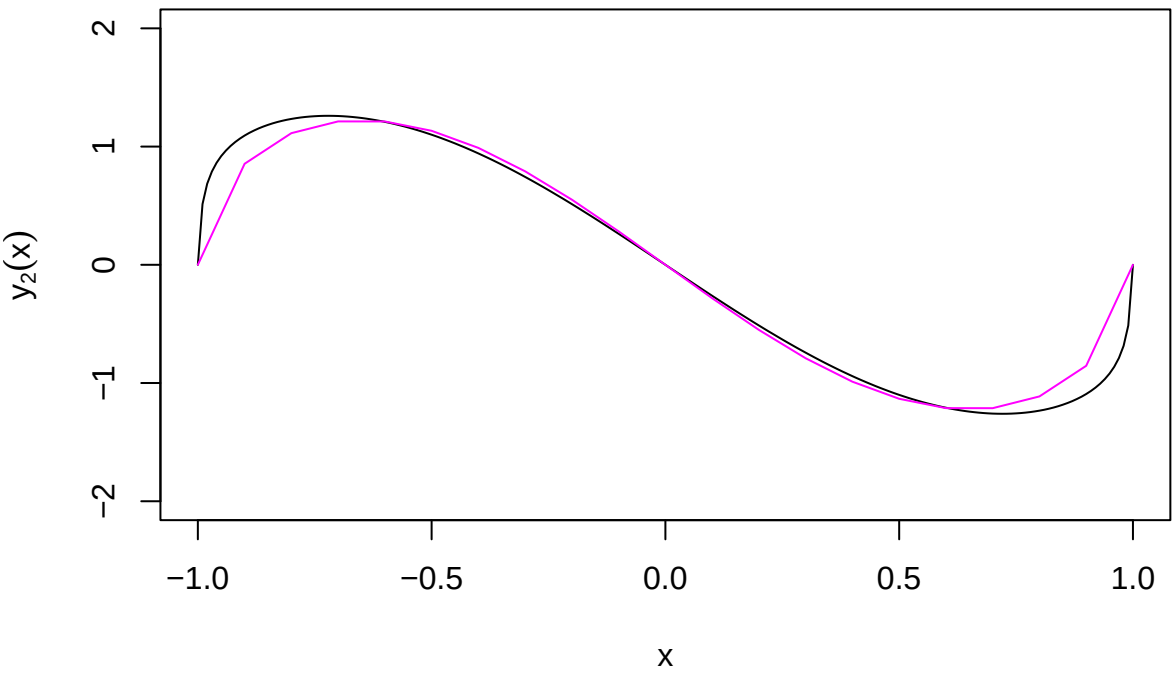


The same problem of inverted sign for eigenfunctions, has presented itself here. Let us *cure* the problem as done in the previous exercise, with the extra complication that we will have to match points for the correlation, as the grids are different (see Exercise 06).

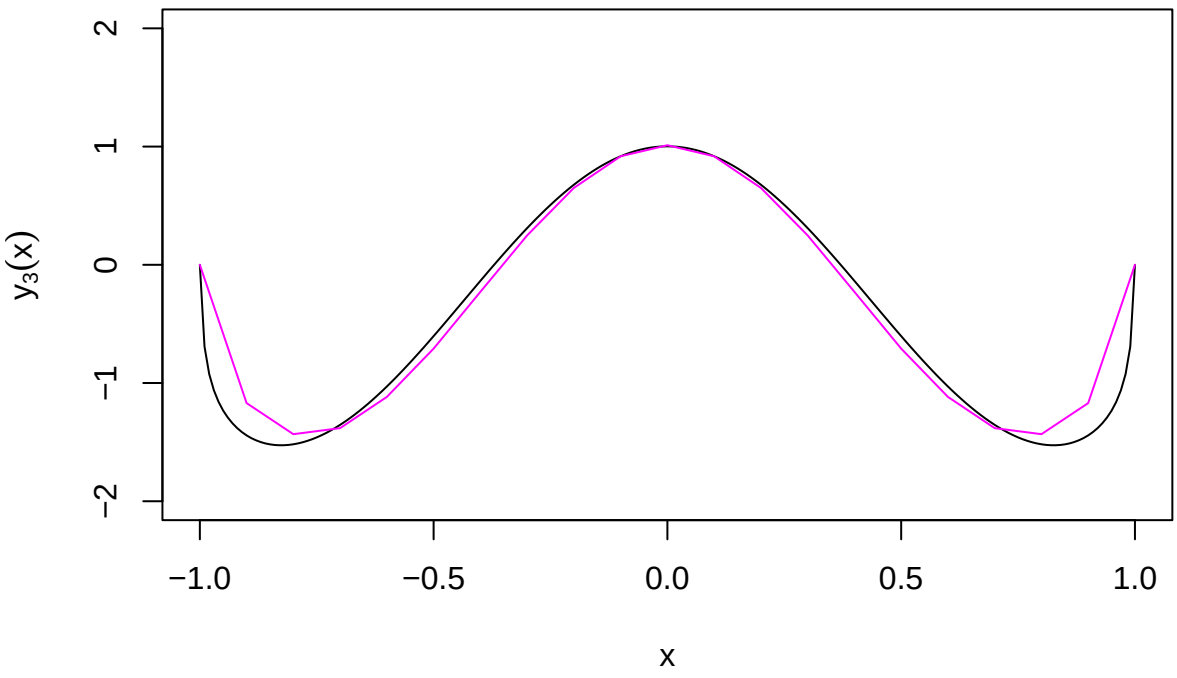
```
# Matching points. Row i, Column j equal TRUE means there's a match
tol <- 1e-12
M <- abs(outer(x,x2,`-`)) <= tol

# Transform in index pairs. Matrix with two columns: (i "matches" j)
pairs <- which(M,arr.ind=TRUE)

# Align by correlation with known eigenfunctions
Ytrue <- cbind(y1[pairs[,1]],y2[pairs[,1]],
               y3[pairs[,1]],y4[pairs[,1]])
V <- ep2$vectors_full
for (k in 1:4) {
  if (sum(V[,k]*Ytrue[,k]) < 0) V[,k] <- -V[,k]
}

# Now plot comparisons again.
plot(x,y1,type="l",ylim=c(-2,2),
     xlab=expression(x),ylab=expression(y[1](x)))
points(x2,V[,1],type="l",col="magenta")
```



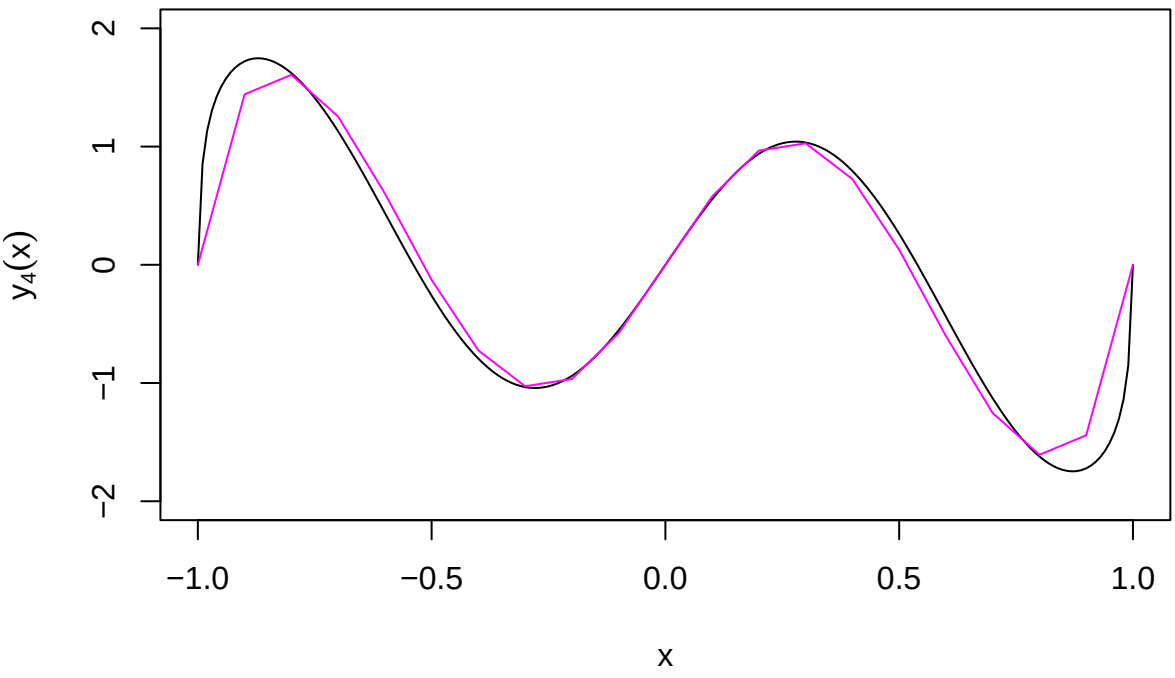

```
plot(x,y2,type="l",ylim=c(-2,2),
     xlab=expression(x),ylab=expression(y[2](x)))
points(x2,V[,2],type="l",col="magenta")
```




```
plot(x,y3,type="l",ylim=c(-2,2),
     xlab=expression(x),ylab=expression(y[3](x)))
points(x2,V[,3],type="l",col="magenta")
```




```
plot(x,y4,type="l",ylim=c(-2,2),
     xlab=expression(x),ylab=expression(y[4](x)))
points(x2,V[,4],type="l",col="magenta")
```



We can observe, as it is generally the case in all problems with a discretised integration interval, that the accuracy decreases with the coarser grid.

We can also try and observe empirically the difference in eigenvalues when the grid is refined further.

```
# Need only to modify the x grid
h <- 0.005
x3 <- seq(a,b,by=h) # Should have n+1=401 grid points

# Solve the Sturm-Liouville eigenproblem
ep3 <- EPSturmLiouville2(p,q,w,x3,nev=4,normalize=TRUE)

# Need only to modify the x grid
h <- 0.0025
x4 <- seq(a,b,by=h) # Should have n+1=801 grid points

# Solve the Sturm-Liouville eigenproblem
ep4 <- EPSturmLiouville2(p,q,w,x4,nev=4,normalize=TRUE)

# Compare eigenvalues
cmp <- cbind(n=1:4,h01=ep2$values,h001=ep$values,h0005=ep3$values,
             h00025=ep4$values)
print(round(cmp,3))
#>      n    h01   h001  h0005 h00025
#> [1,] 1  0.796  0.497  0.446  0.405
#> [2,] 2  7.070  6.125  5.963  5.832
#> [3,] 3 18.484 16.924 16.633 16.399
```

```
#> [4,] 4 34.611 32.774 32.338 31.991
```

In this case we can see that values are converging from above. The appropriate step size can be decided based, for example, on a minimal drop in value between corresponding eigenvalues for different step sizes.