Solutions to Exercises from Chapter 06

Contents

1 Exercises on the roots of nonlinear equations **1**
1.1 Exercise 01 . . . 1
1.2 Exercise 02 . . . 2
1.3 Exercise 03 . . . 3
1.4 Exercise 04 . . . 5
1.5 Exercise 05 . . . 8
1.6 Exercise 06 . . . 11

2 Exercises on the roots of systems of nonlinear equations **18**
2.1 Exercise 07 . . . 18
2.2 Exercise 08 . . . 22
2.3 Exercise 09 . . . 24
2.4 Exercise 10 . . . 27

The `comphy` package is loaded once at the beginning so to make all its functions available to this exercises session.

```
library(comphy)
```

1 Exercises on the roots of nonlinear equations

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)
```

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.

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

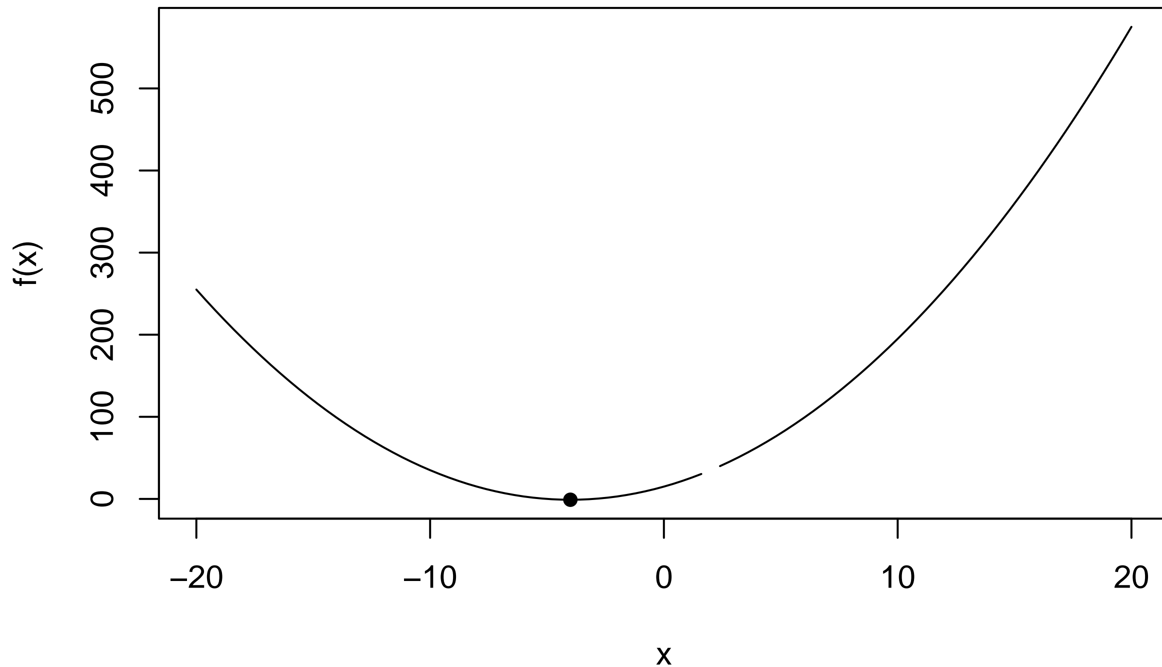
$$\frac{(3x^2+12x-1)(x-2)-(x^3+6x^2-x-30)}{(x-2)^2}.$$

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 $x = 2$.

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

$$f(x) = 2\sin(6\pi x) - e^{-x}$$

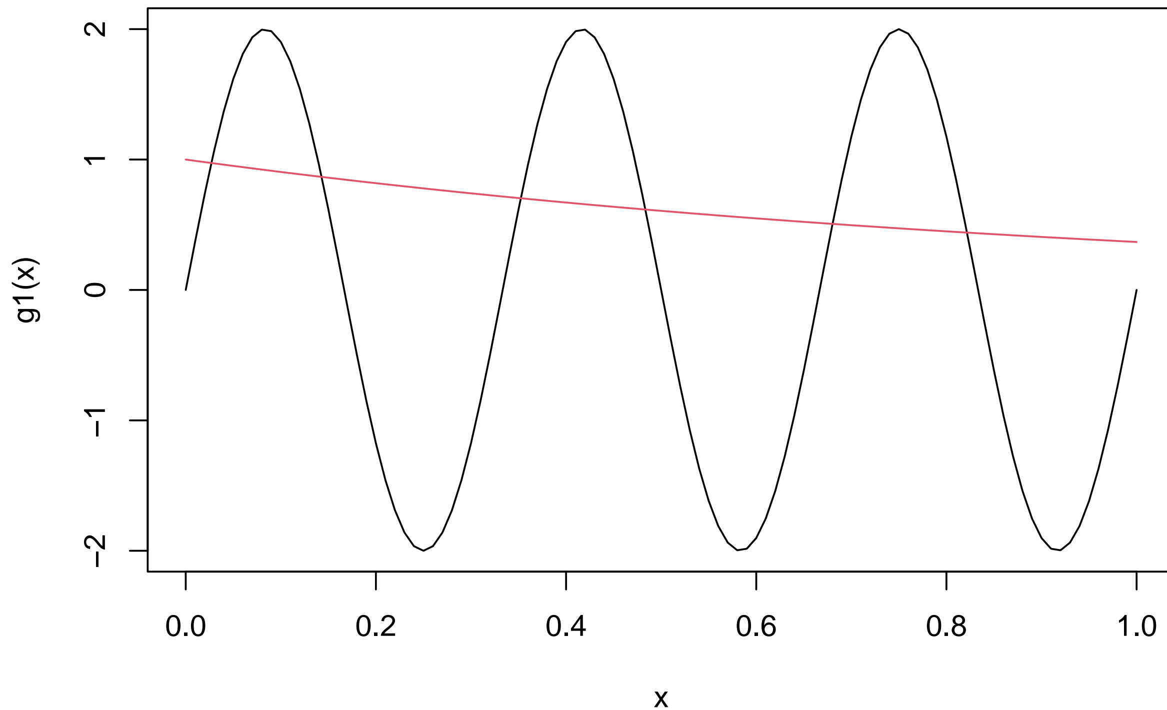
To use Newton-Raphson, we need the first derivative of this function, which is

$$f'(x) = 12\pi\cos(6\pi x) + e^{-x}$$

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))
```

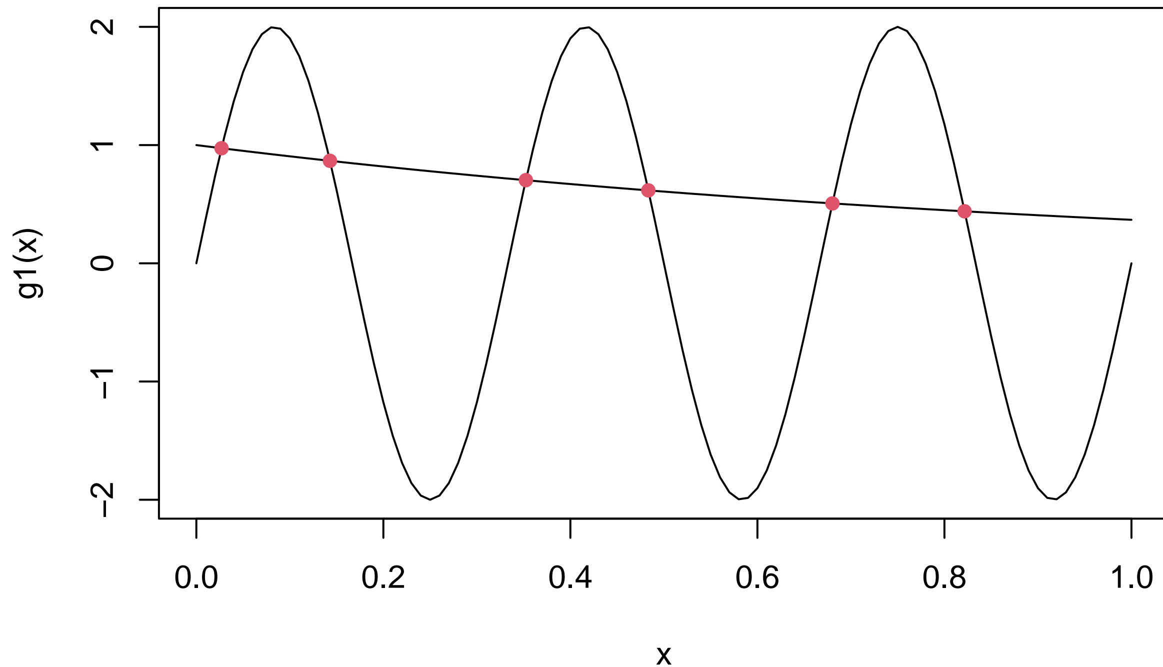
```r
# Plot functions between 0 and 1
curve(g1(x),from=0,to=1)
curve(g2(x),from=0,to=1,col=2,add=TRUE)
```



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.

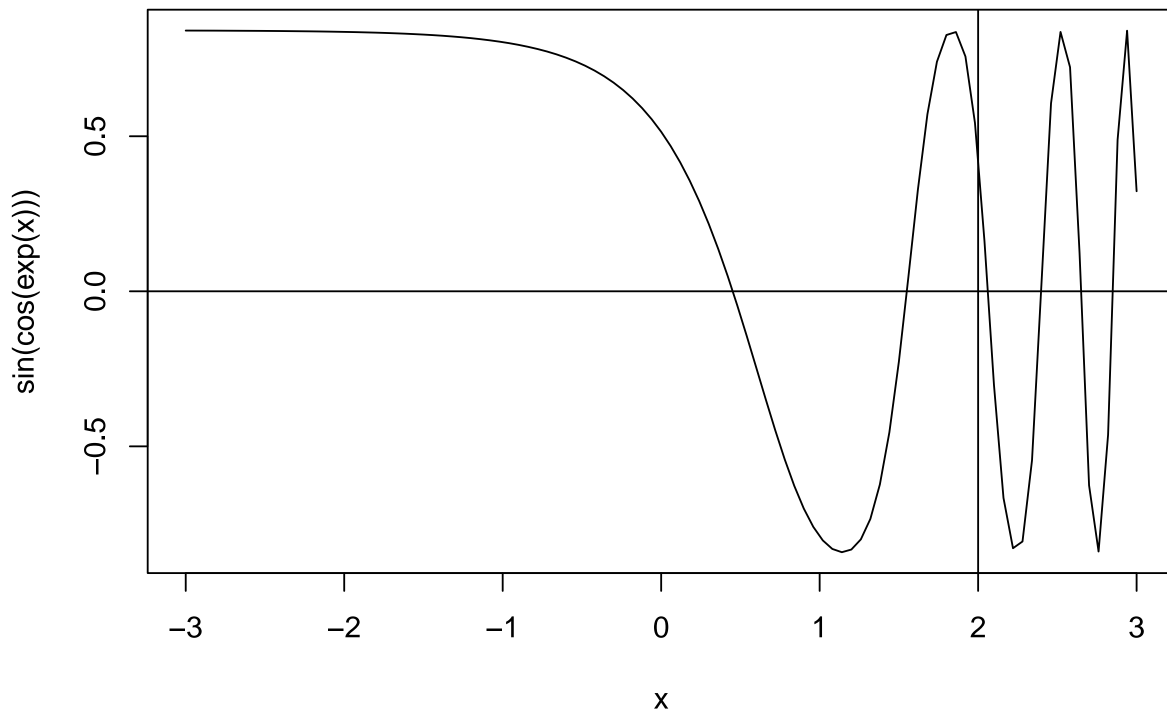
1.5 Exercise 05

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
```

```r
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 $[1, 2]$.

```r
# 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.

```r
# 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.

```r
# 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.

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
```

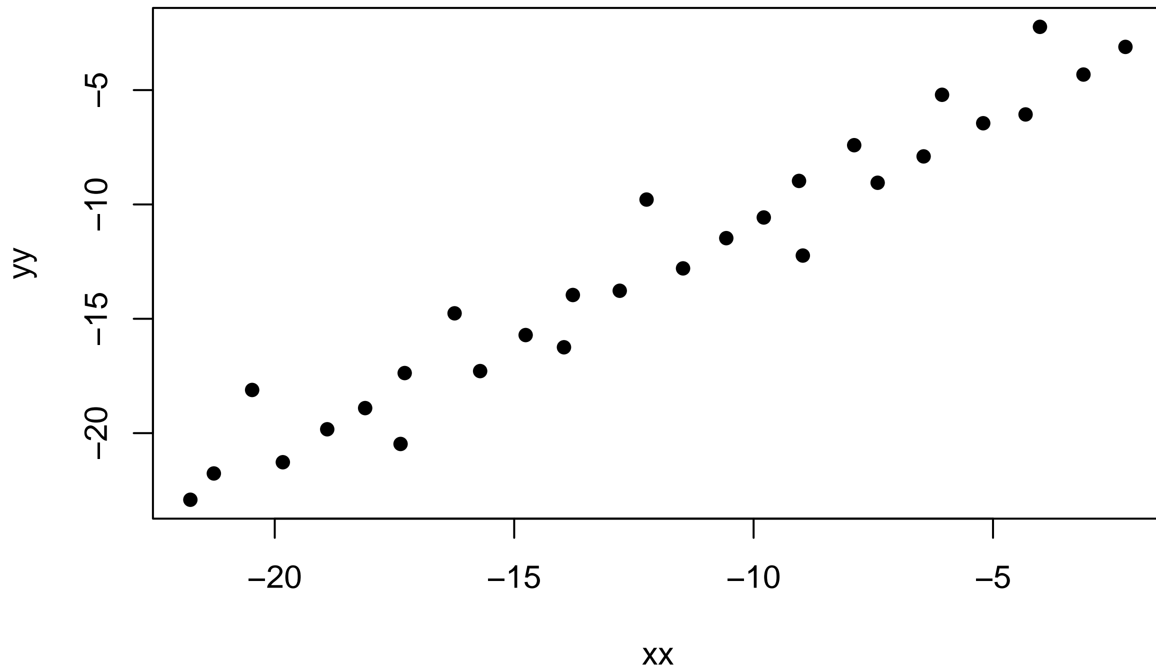
```
#> 16 1.73204040527344 1.73205566406250 1.73204803466797 1.52587890625000e-05
#> 17 1.73204803466797 1.73205566406250 1.73205184936523 7.62939453125000e-06
#> 18 1.73204803466797 1.73205184936523 1.73204994201660 3.81469726562500e-06
#> 19 1.73204994201660 1.73205184936523 1.73205089569092 1.90734863281250e-06
#> 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 1.49011611938477e-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 $\log(|\epsilon_n|)$.

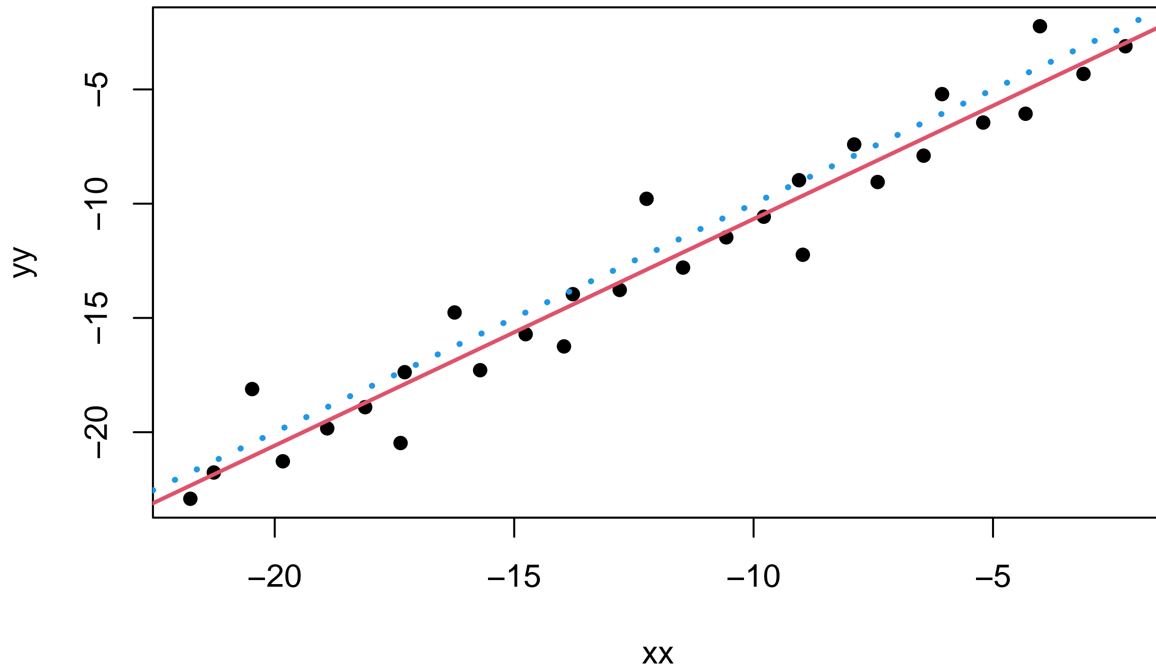
```
# x coordinates
xx <- log(abs(ers[1:29]))

# 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)
```

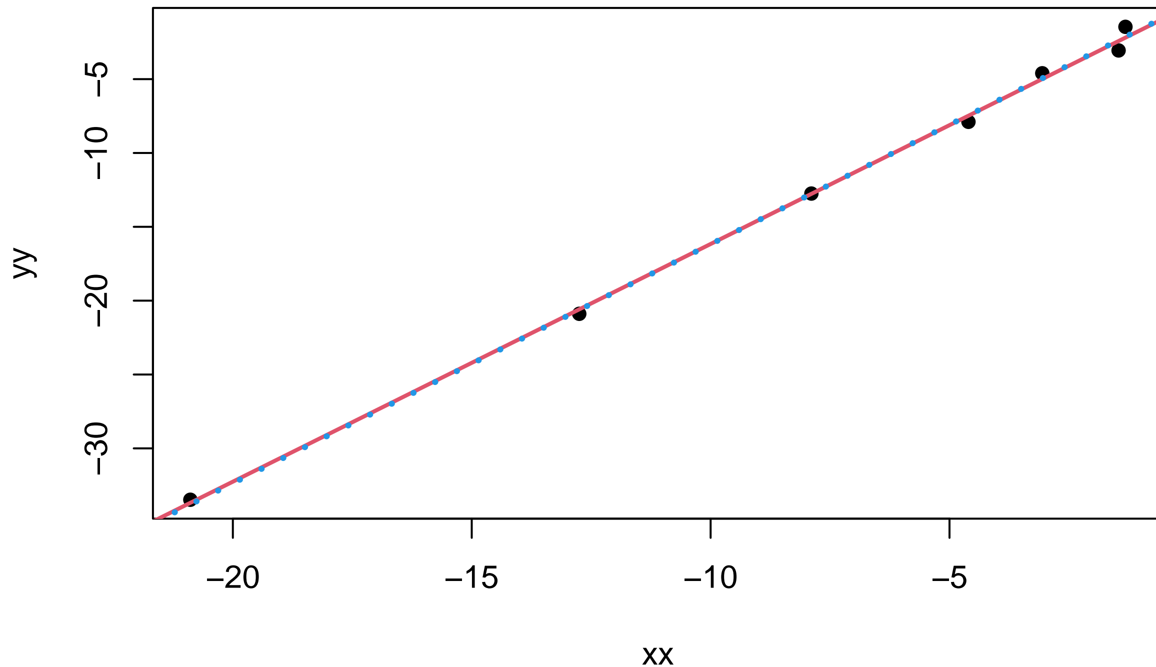
```r
abline(model,lwd=2,col=2)
abline(a=0,b=1,lwd=3,lty=3,col=4)
```



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.

```r
# 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)
```

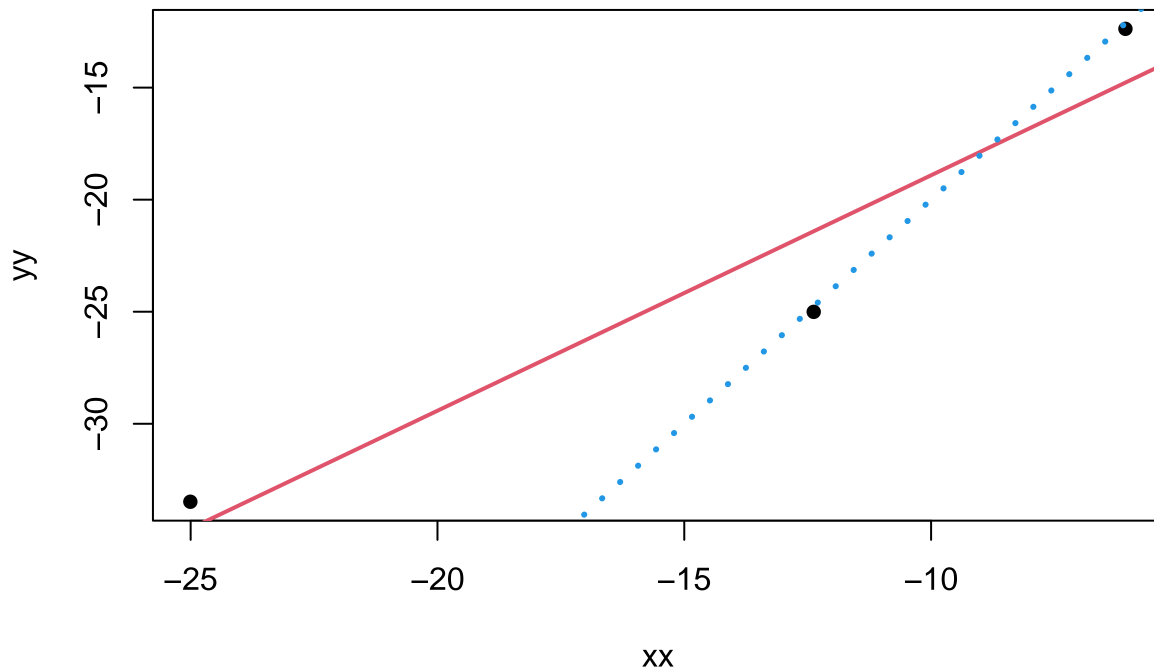
```r
# x coordinates
xx <- log(abs(ers[1:7]))

# 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.

2 Exercises on the roots of systems of nonlinear equations

2.1 Exercise 07

Consider the following system of nonlinear equations:

$$\left\{ \begin{array}{lcl} x^2 + y^2 & = & 4 \\ e^x + y & = & 1. \end{array} \right.$$

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

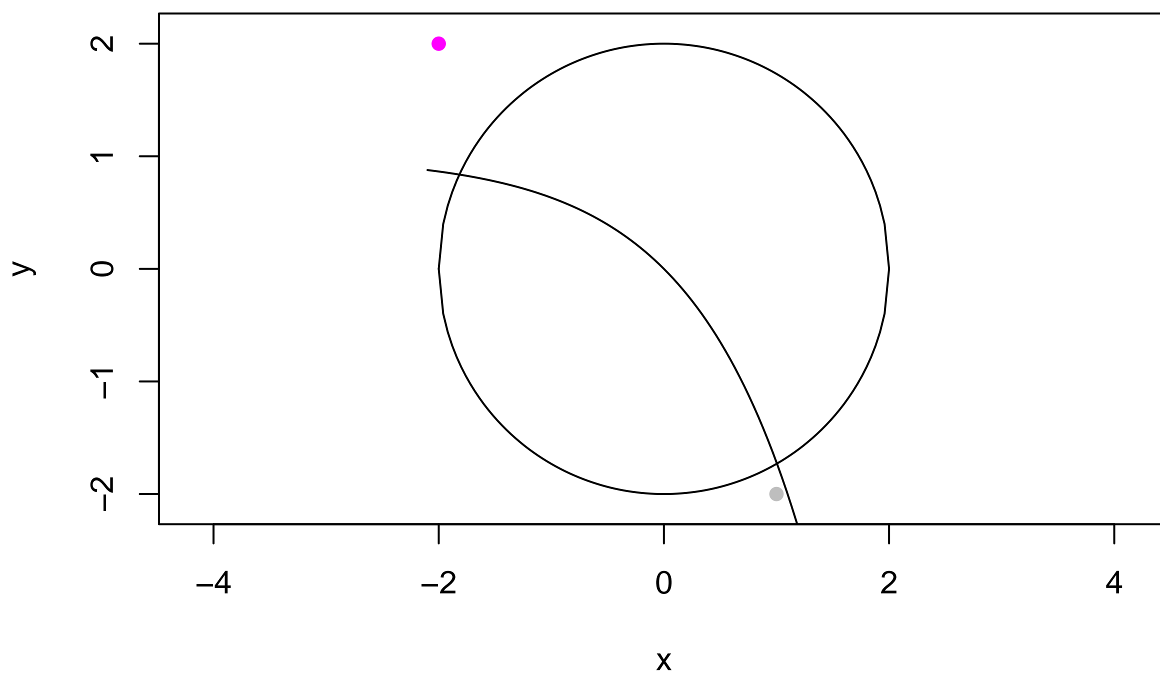
A plot of the two curves

$$x^2 + y^2 = 4, \qquad y = 1 - e^x,$$

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")
```



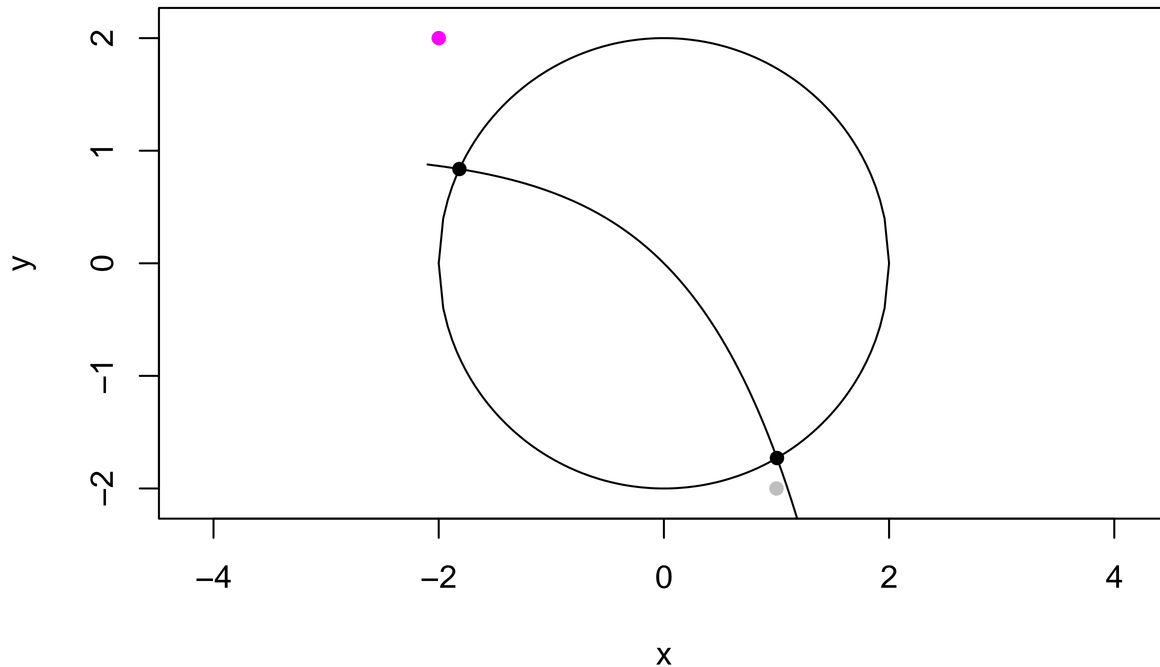
Two possible points to start the search for the intersections are $(-2, 1)$, coloured in magenta, and $(1, -2)$, coloured in grey. The search with both R functions is done here.

```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)
```

```r
#> 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]  1.004168738452203 -1.729637286984234
```

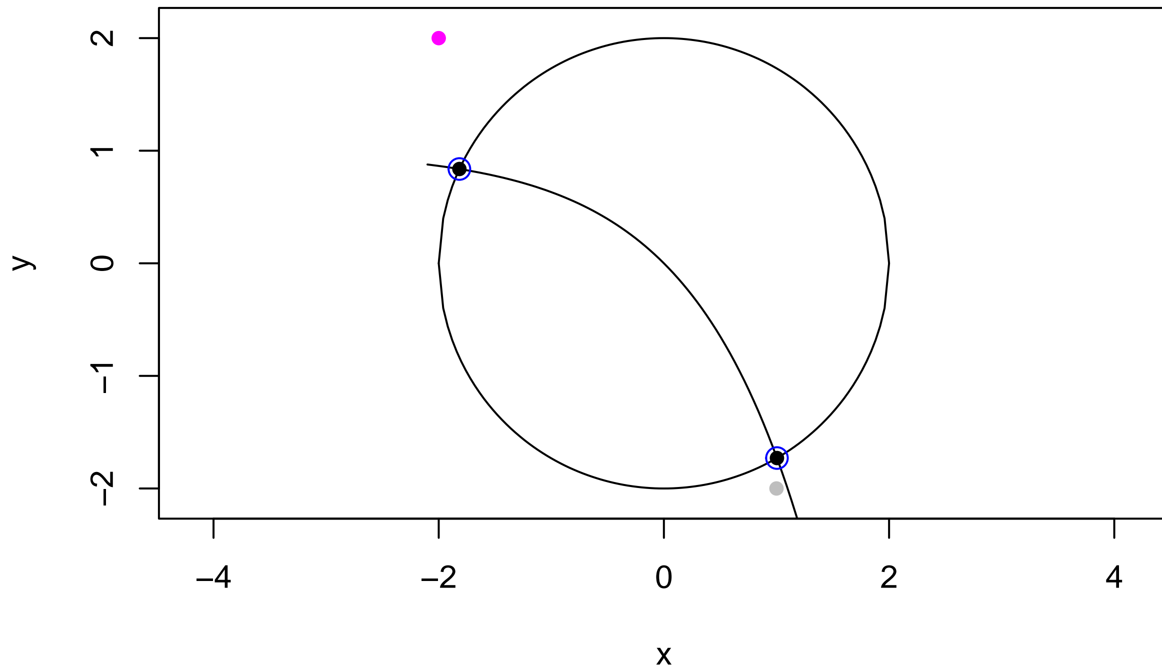
We can also plot the points found on the curves' plot.

```r
# 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]  1.004168739422263 -1.729637287780070

# 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.

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)
```

```r
# 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 0.242745878757069 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?

```r
# 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] 0.242745878757383 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.

```r
# 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.

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

$$-0.5 \le x \le 0.5, \quad -1 \le y \le 1, \quad -1 \le z \le 1.$$

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
```

```
#>  0.468799768818373 -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
```

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.

```r
# 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`.

```r
# Run with fixed ftol
dtmp <- data.frame()
```

```r
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`.

```r
# 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.