Solutions to Exercises from Chapter 03

Contents

1 Exercises on Linear interpolation 1
1.1 Exercise 01 1
1.2 Exercise 02 3
1.3 Exercise 03 7
1.4 Exercise 04 8
1.5 Exercise 05 12

2 Exercises on Lagrangian interpolation and the Neville-Aitken algorithm 12
2.1 Exercise 06 12
2.2 Exercise 07 14
2.3 Exercise 08 18
2.4 Exercise 09 21
2.5 Exercise 10 24
2.6 Exercise 11 25
2.7 Exercise 12 26

3 Exercises on divided differences 30
3.1 Exercise 13 30
3.2 Exercise 14 31
3.3 Exercise 15 33
3.4 Exercise 16 36
3.5 Exercise 17 36

4 Exercises on cubic splines 38
4.1 Exercise 18 38
4.2 Exercise 19 39
4.3 Exercise 20 41

1 Exercises on Linear interpolation

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

```
library(comphy)
```

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

x	$f(x)$

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`.

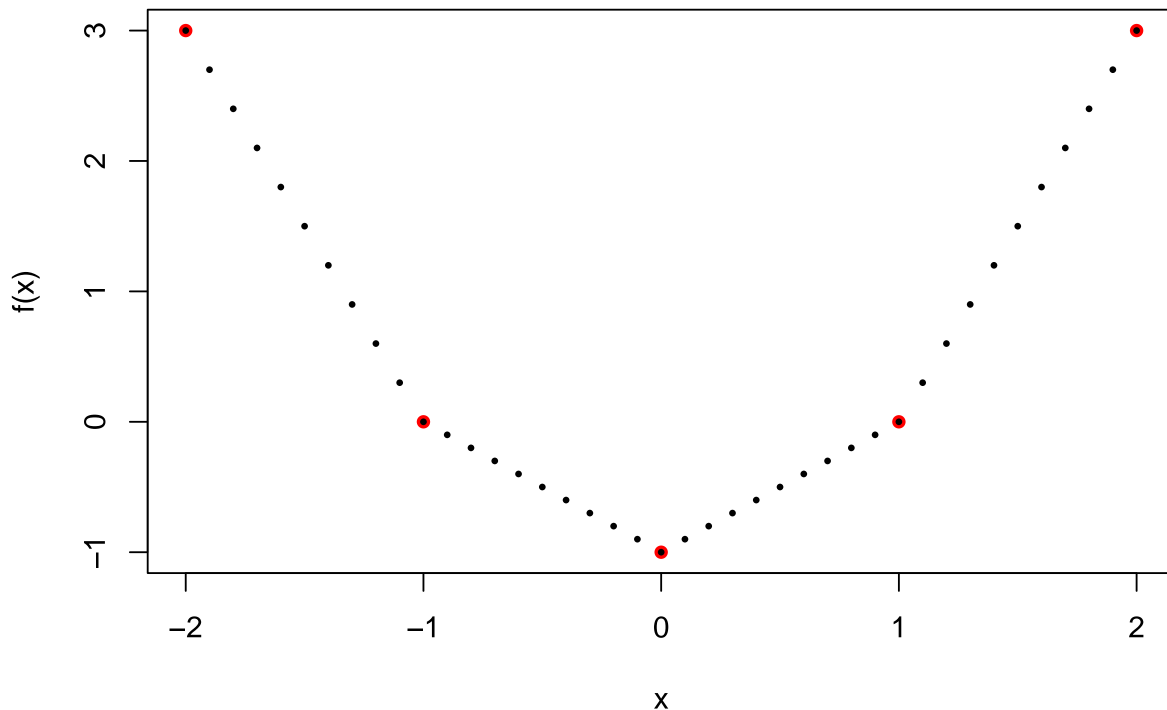
```
# Known values
x <- c(-2,-1,0,1,2)
f <- c(3,0,-1,0,3)

# 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)
```



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).

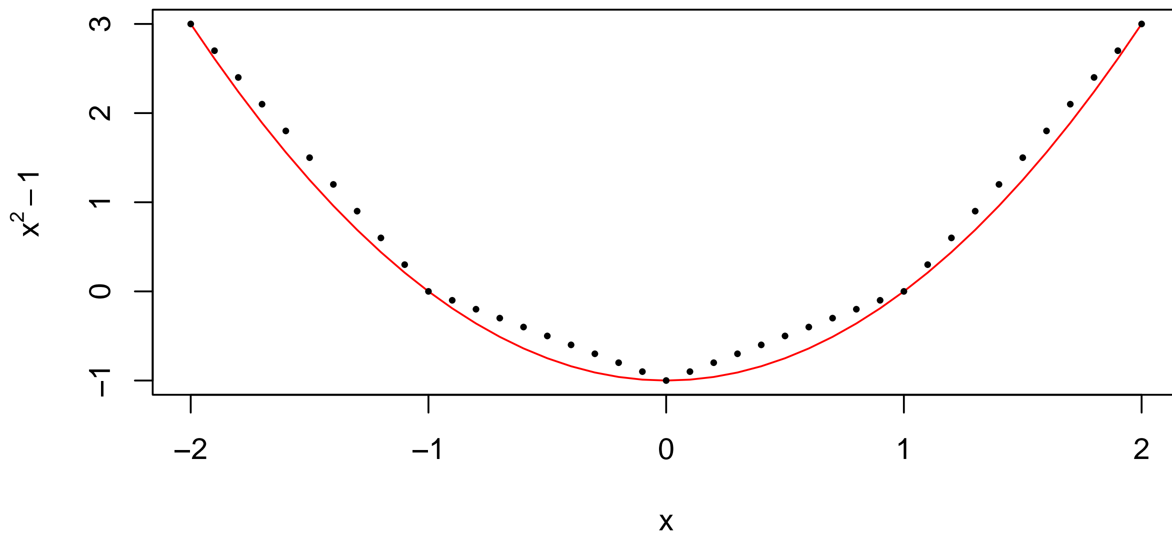
SOLUTION

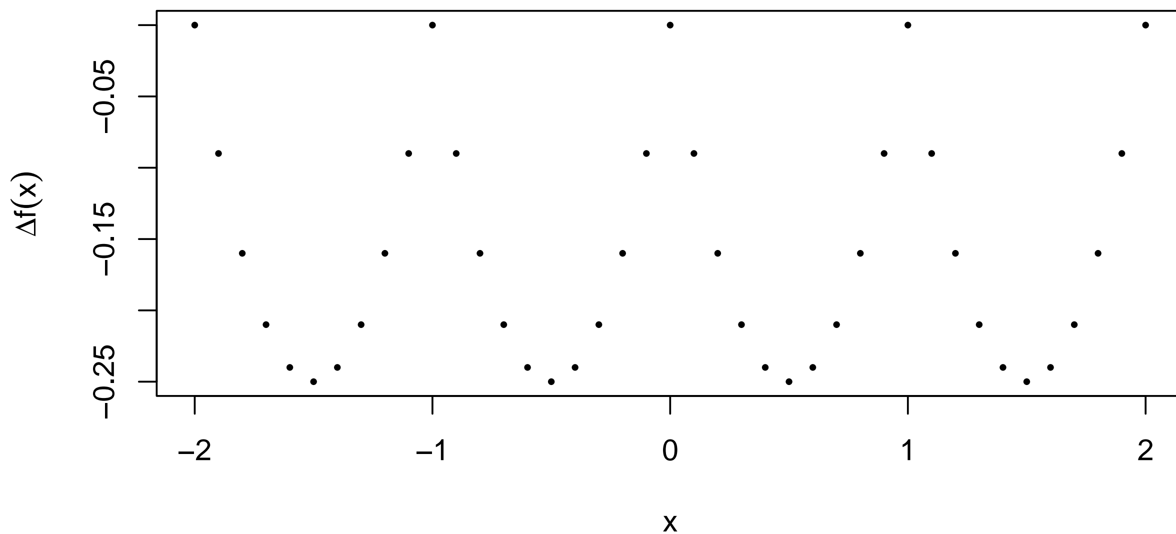
The linear approximation $f_{\text{int}}(x)$ has been already computed in Exercise 01. To calculate the error $\Delta f(x) = f(x) - f_{\text{int}}(x)$ we need to sample $f(x) = x^2 - 1$ at the same points of the interpolating grid `x0`; the vector containing the correct values is called `ftrue`. The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





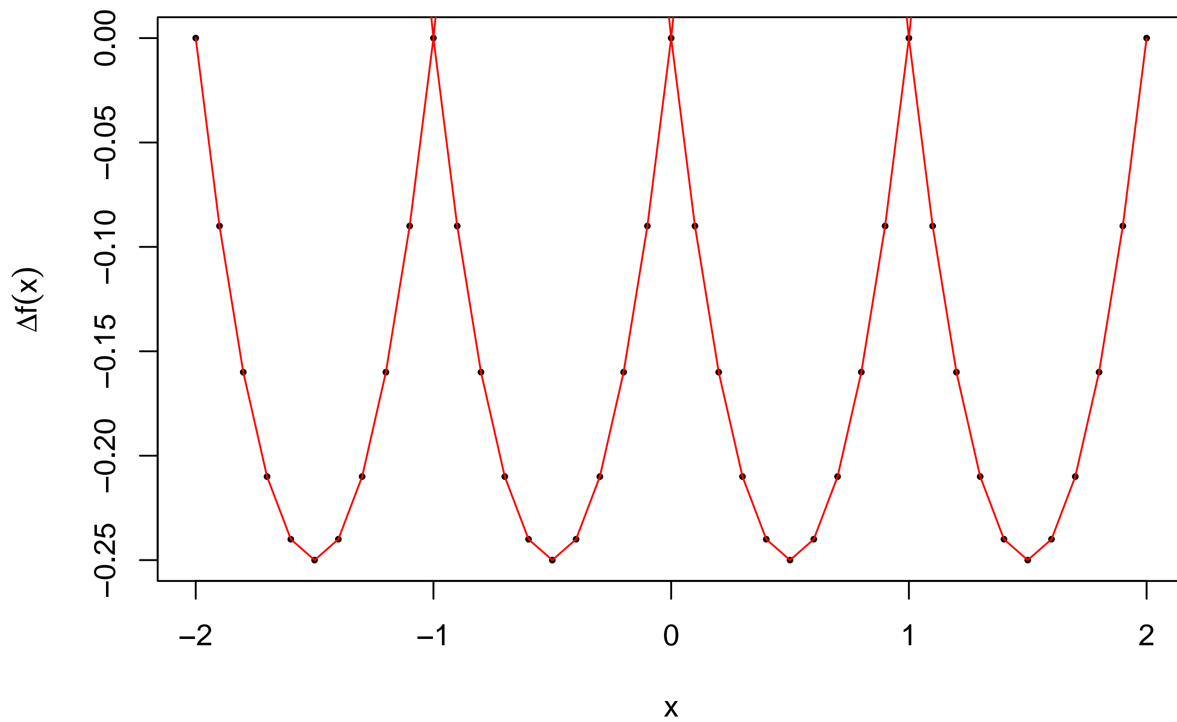
The function is always below its linear interpolation, for this specific example, so that their difference is negative or zero. It is possible, using equation (3.4) to estimate the largest value of the interpolation error. From the picture, this seems to be equal to $1/4 = 0.25$. Formula (3.4) yields

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

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.

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.

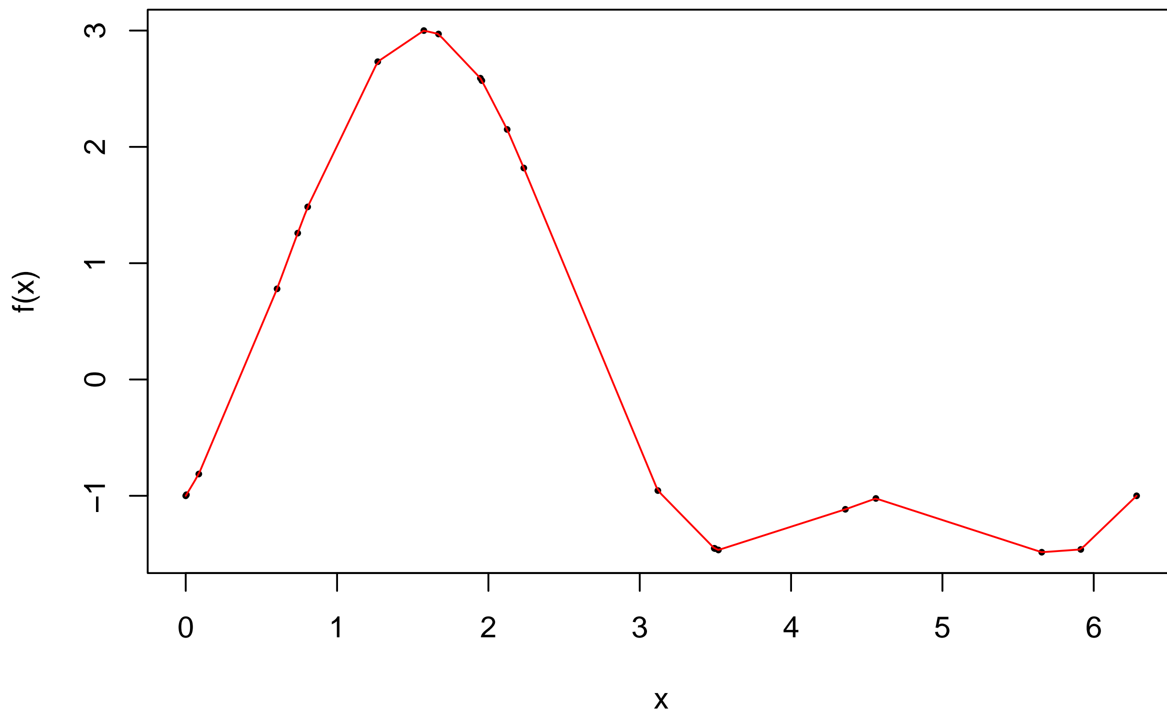
SOLUTION

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")
```



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

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

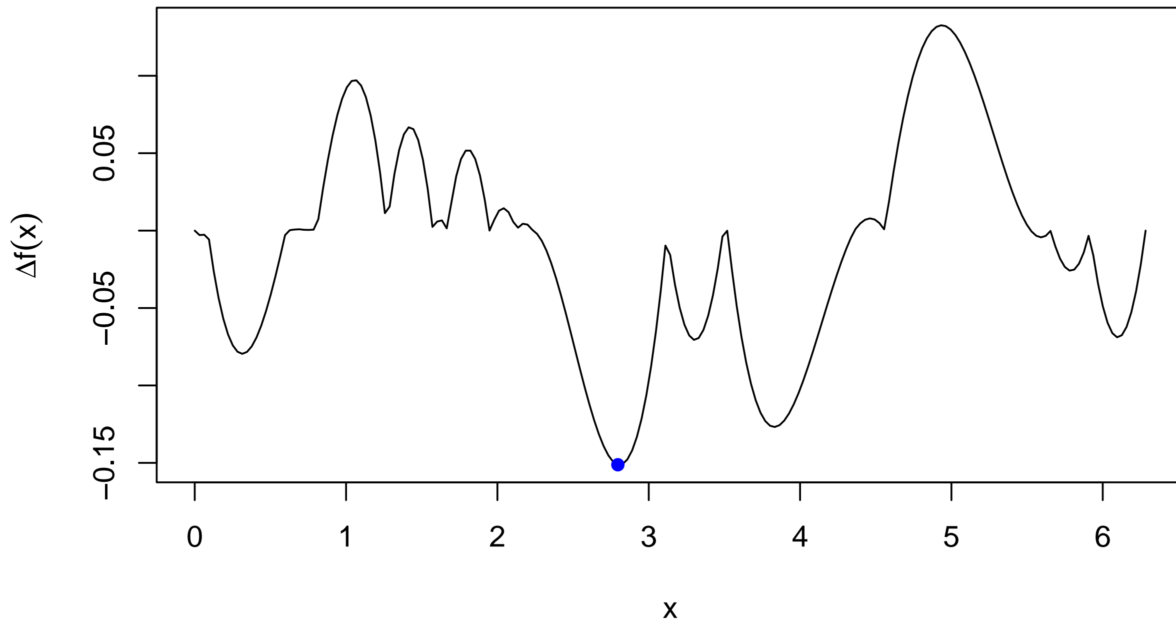
is the largest. Can you justify the value found, using formula (3.4)?

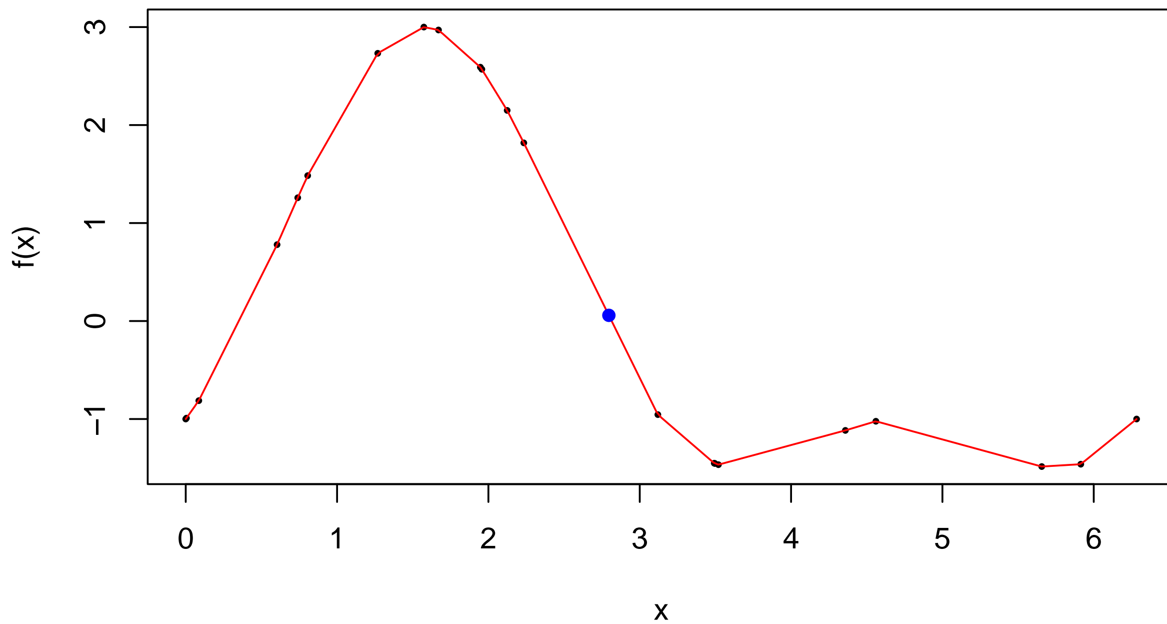
SOLUTION

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)
```

```r
}

# 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.

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 $|\Delta f(x)| \equiv |f(x) - f_{\text{int}}(x)|$ 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 $2\sqrt{\epsilon}$.

2 Exercises on Lagrangian interpolation and the Neville-Aitken algorithm

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:

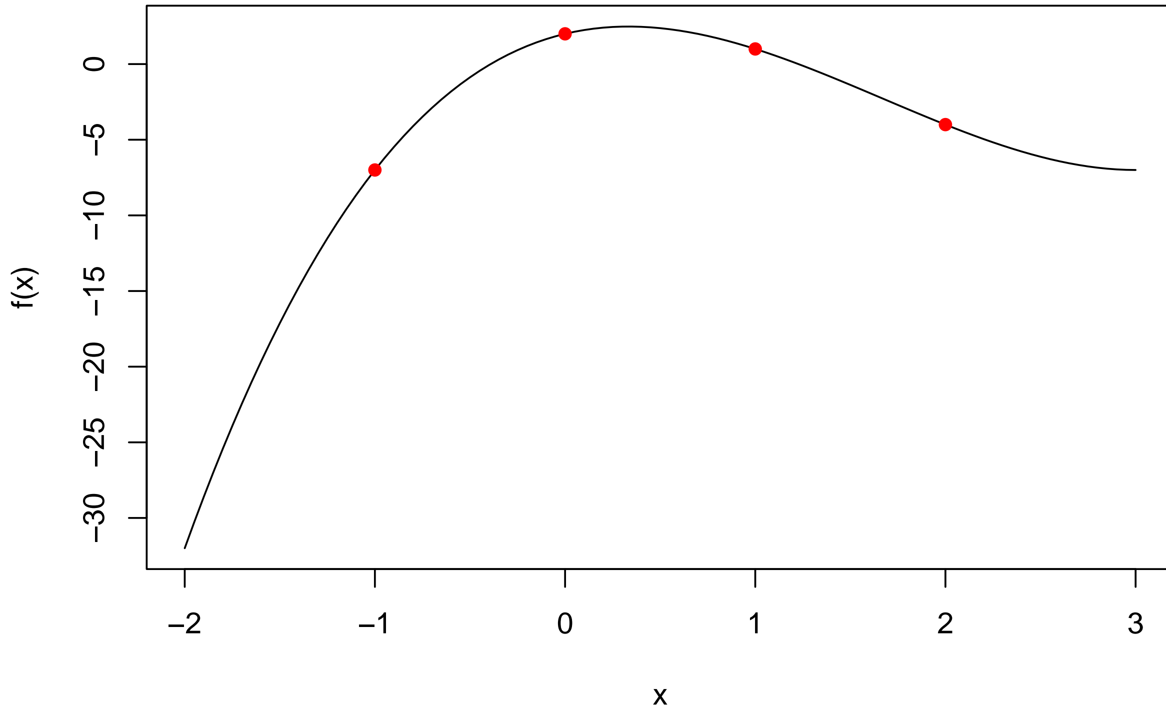
$$
f(x) = \sum_{i=1}^{4} f_i L_{3,i}(x)
$$

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")
```



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))
}
```

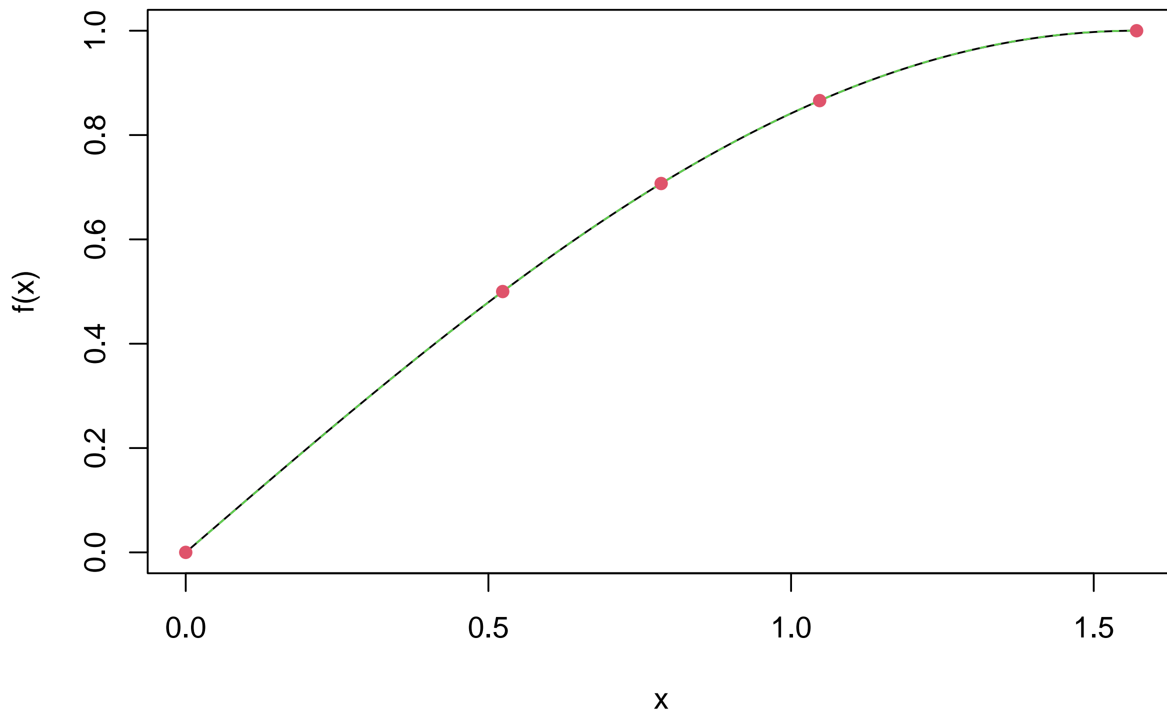
Once created, a function should always be tested to check that it returns the expeccted numerical values. In our case it is worth trying with the 5 interpolation points.

```
# 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.

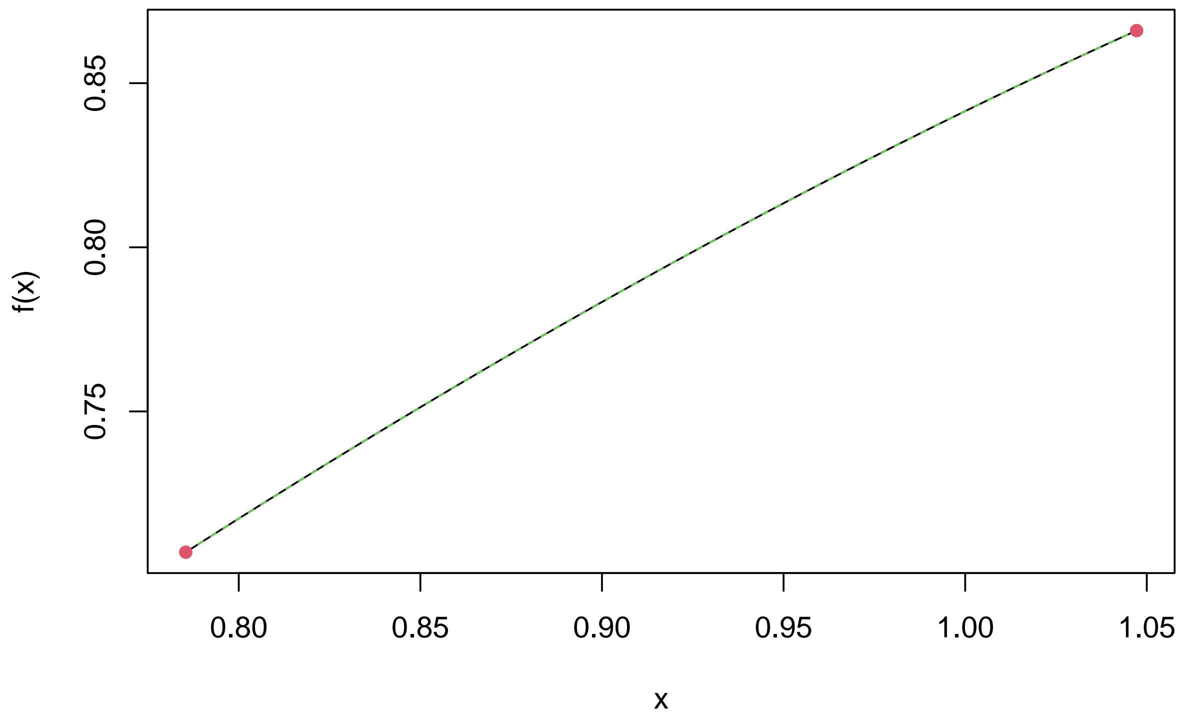
```
x <- seq(0,pi/2,length=1000)
f <- sin(x)

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

```r
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)
```



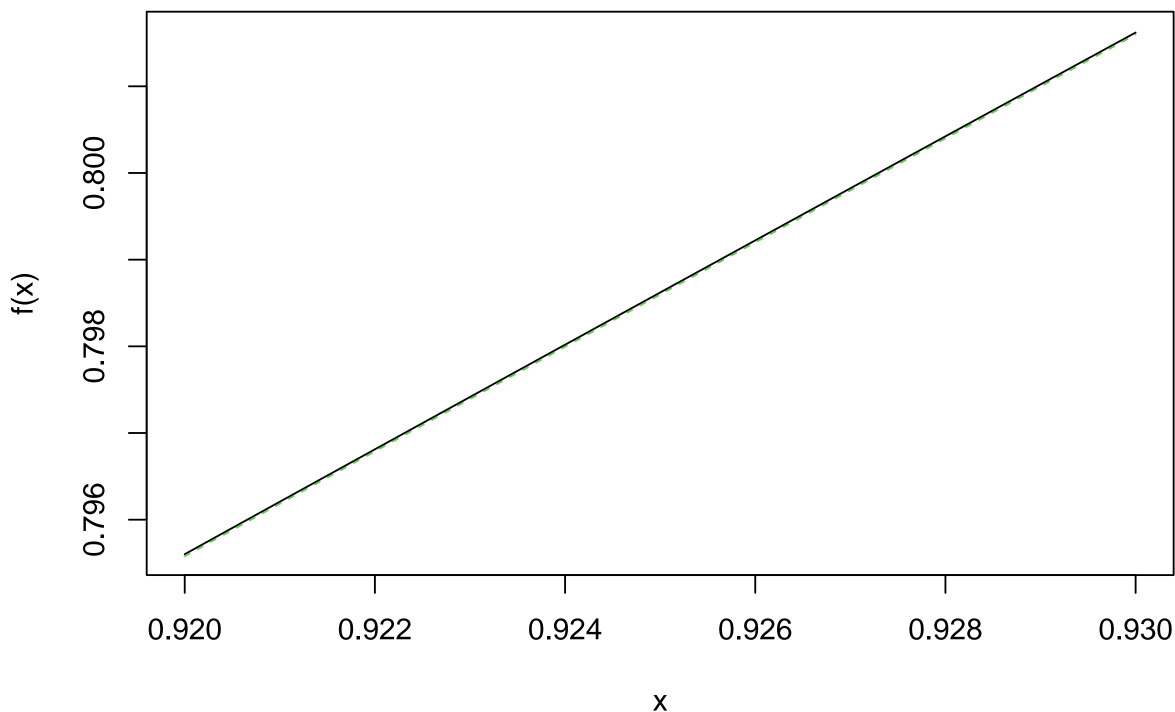
The overlap is a good one, which means the approximation of $f(x)$ with $P_4(x)$ is quite accurate. To appreciate how close the interpolating curve is to $\sin(x)$ we could zoom around a couple of interpolation points, say x_3 and x_4; the visual result is produced using the following code.

```r
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.

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:

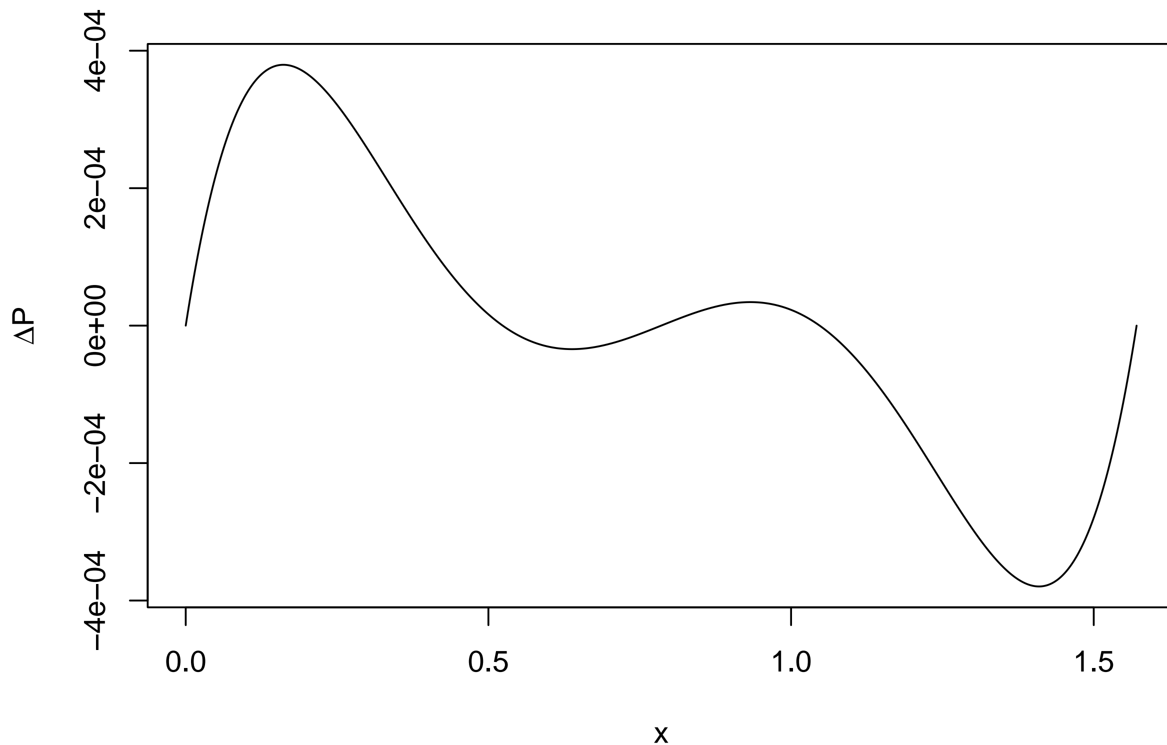
$$\Delta P(x) \equiv \frac{1}{120} x \left(x - \frac{\pi}{6}\right)(x - \pi/4)(x - \pi/3)(x - \pi/2)$$

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)))
```

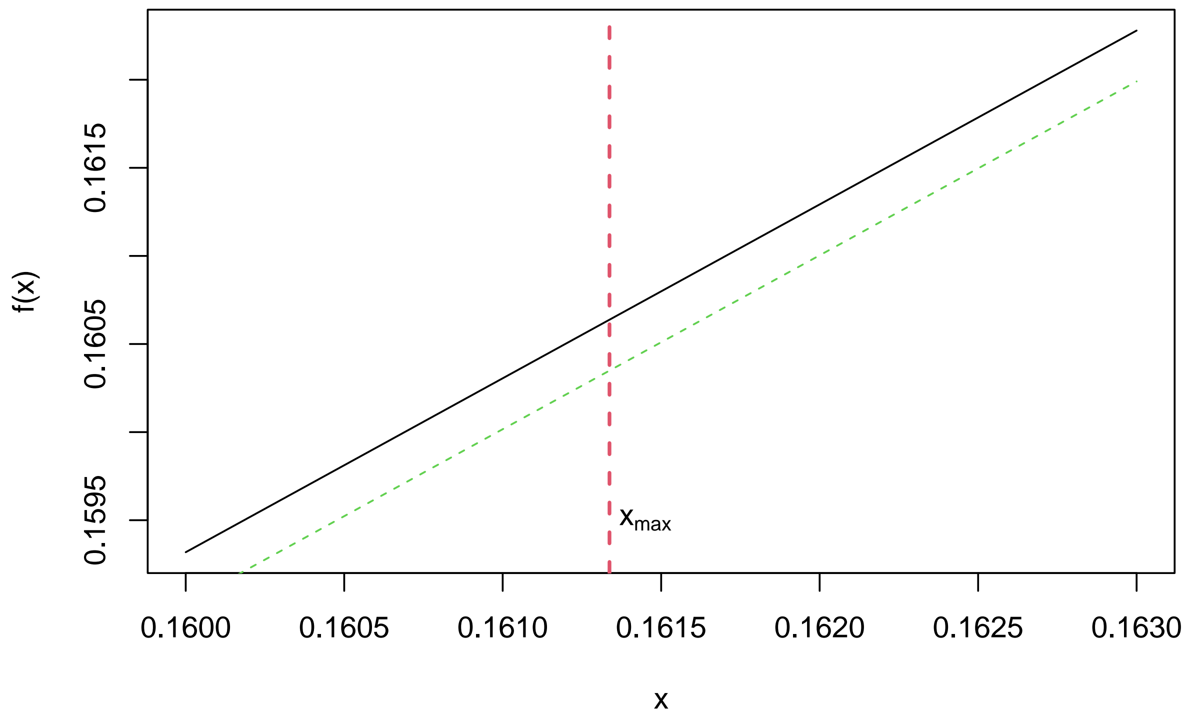


From the plot there appear to be a highest and lowest point, close to 0 and $\pi/2$ respectively. We have indeed,

```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.

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
```

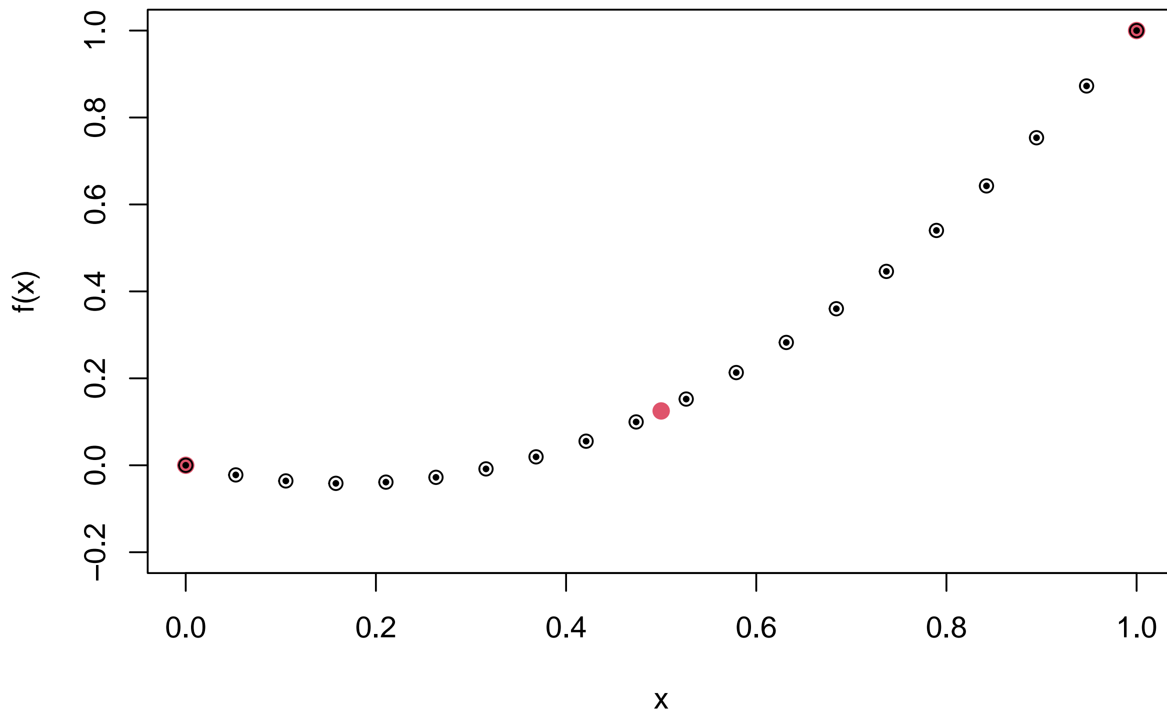
The application of `nevaitpol`, needed to obtain the interpolating point with Neville-Aitken, is more complicated. What is complicated is not so much the use of `sapply`, but how to access the interpolating point afterwards. Let's start with `sapply`. As `x0` has 20 elements and as `nevaitpol` creates 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.

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.

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()
```

```r
  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.

```r
# 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.

```r
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.

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

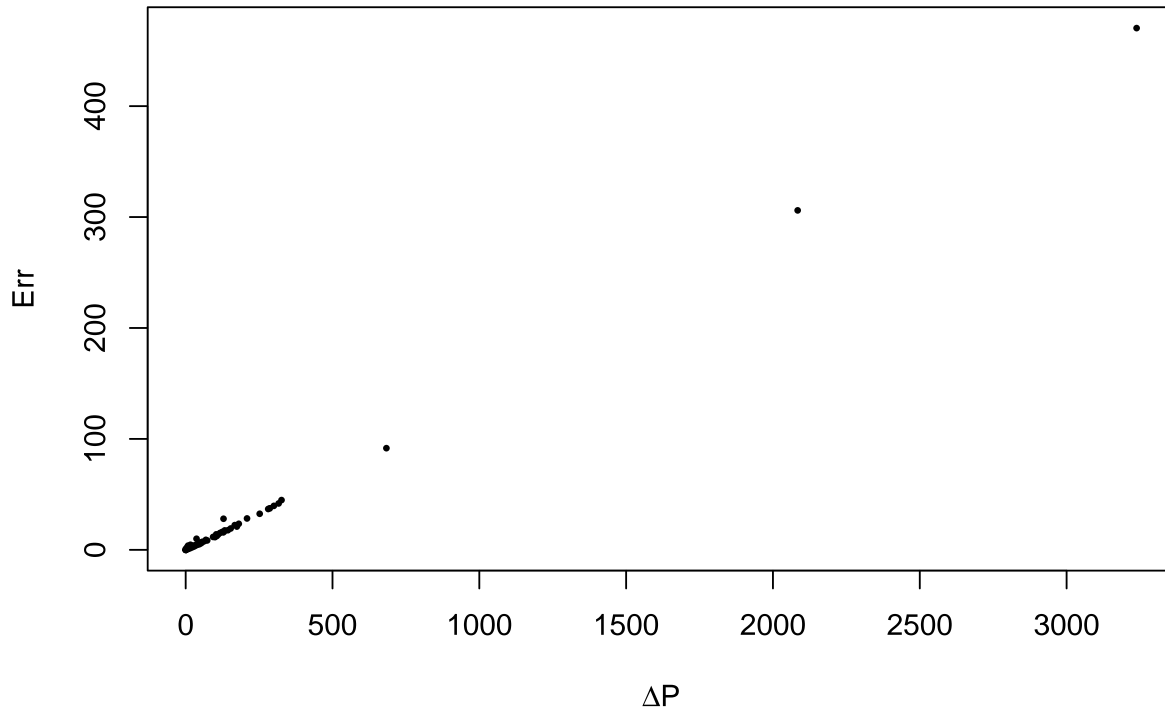
The script that produce the two sets of values is coded as follows.

```
# 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")
```

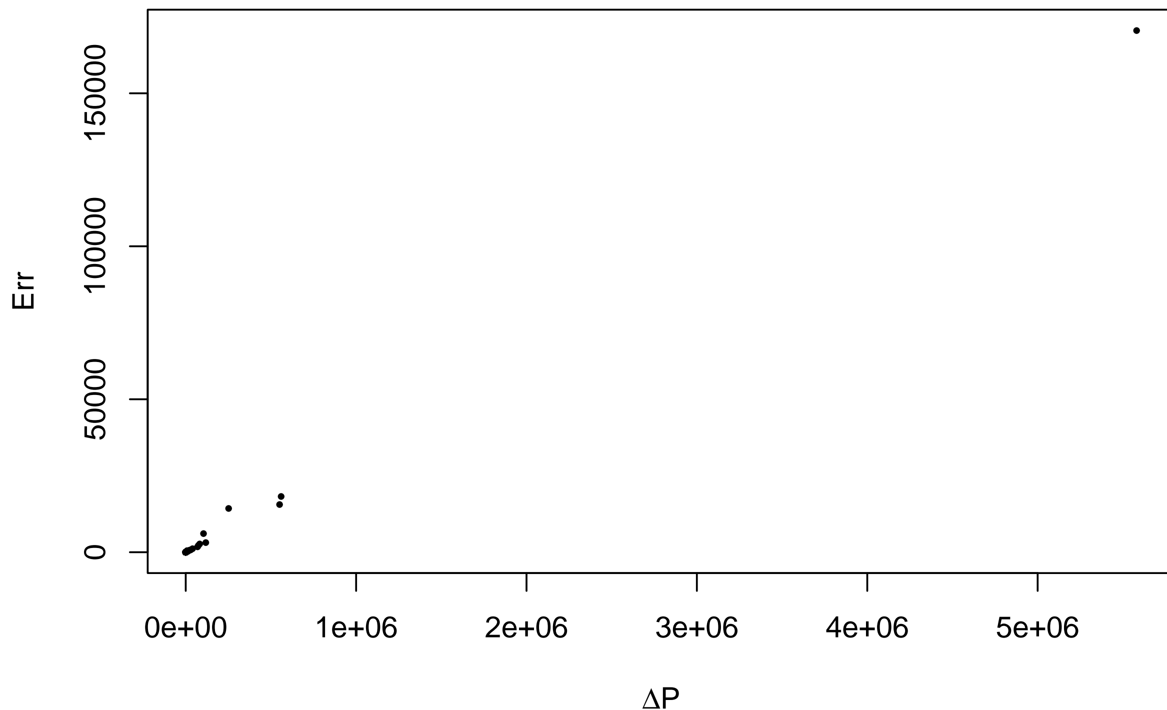


There is a clear direct proportional relation between the two quantities. The difference of the two values in the one-before-last column of the P matrix can therefore give an indication on how the interpolating value is close to the correct one. Further investigations and empirical trials could give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

```r
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 Exercises on divided differences

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
```

3.2 Exercise 14

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

$$
\begin{gathered}
f(x) = a_1 + a_2(x+1) + a_3(x+1)(x-1) + a_4(x+1)(x-1)(x-2)+ \\
a_5(x+1)(x-1)(x-2)(x-4) + a_6(x+1)(x-1)(x-2)(x-4)(x-5),
\end{gathered}
$$

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,

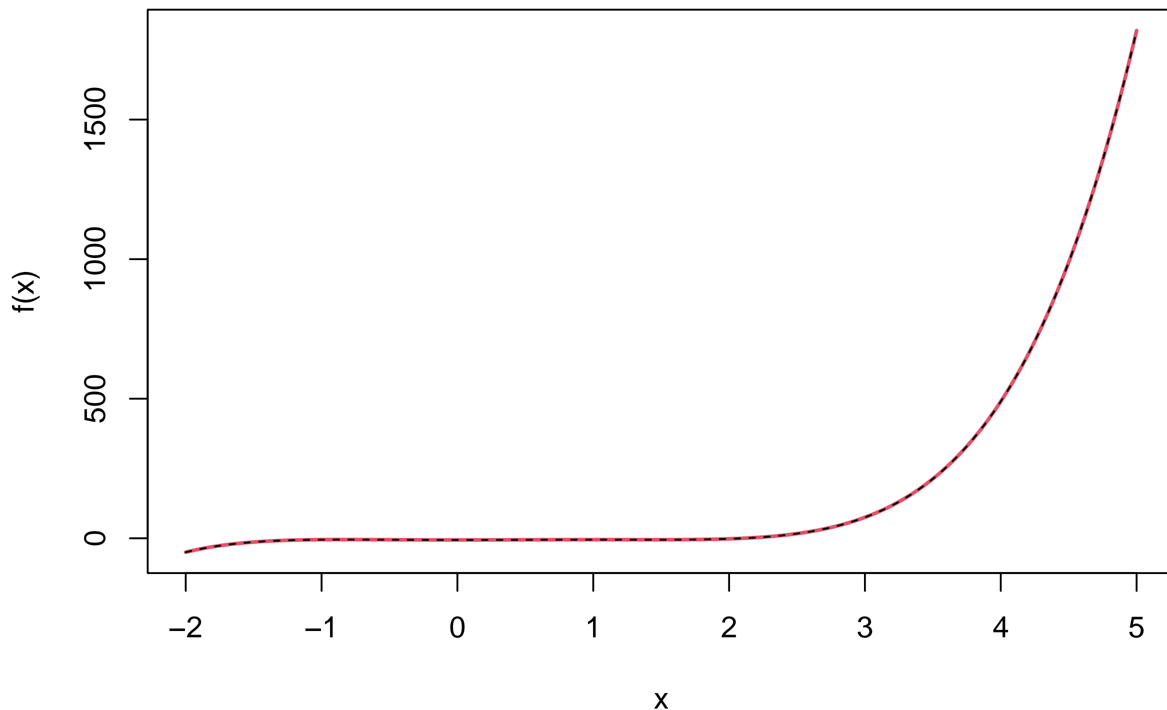
$$
x_1 = -2\ ,\ x_2 = -1\ ,\ x_3 = 1\ ,\ x_4 = 2\ ,\ x_5 = 4\ ,\ x_6 = 5
$$

The calculation can be done without effort in R.

```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 $x = 5$.

```r
# 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 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.

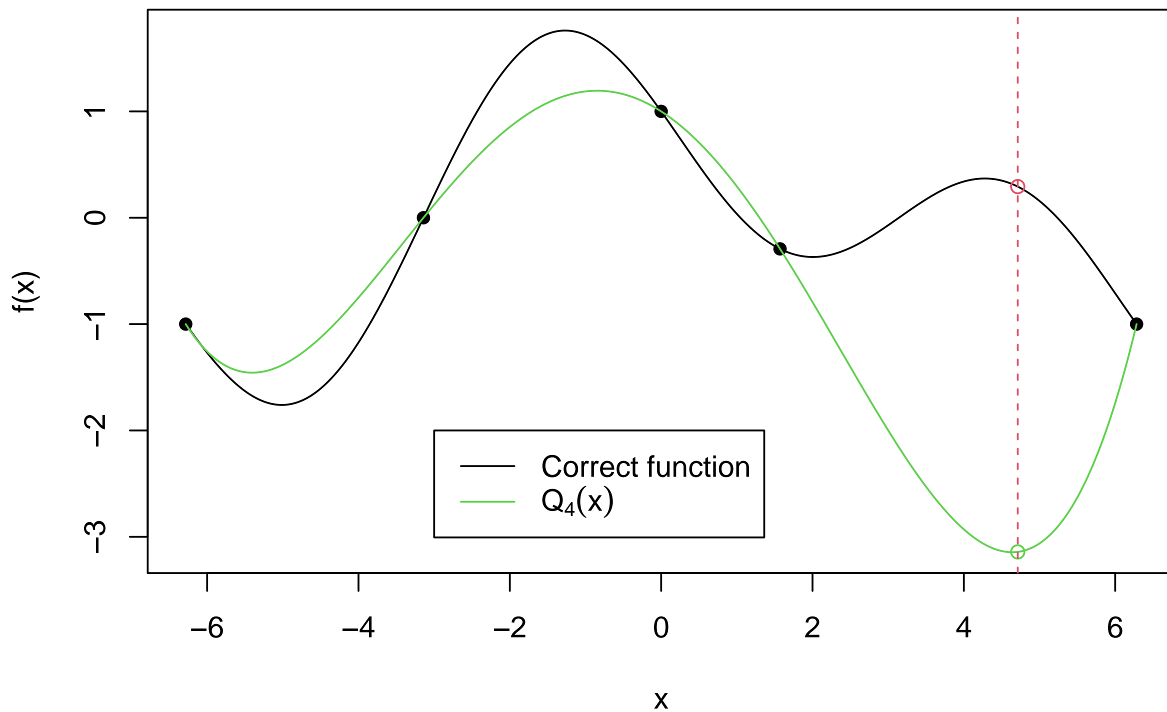
SOLUTION In this exercise we are required to interpolate functions, polynomials and errors at the specific point, $(3/2)\pi$. So, in order to use function `polydivdif`, we need to provide a grid which includes it. First we carry out the calculations for $Q_4(x)$.

```
# 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
```

3.4 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.

3.5 Exercise 17

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

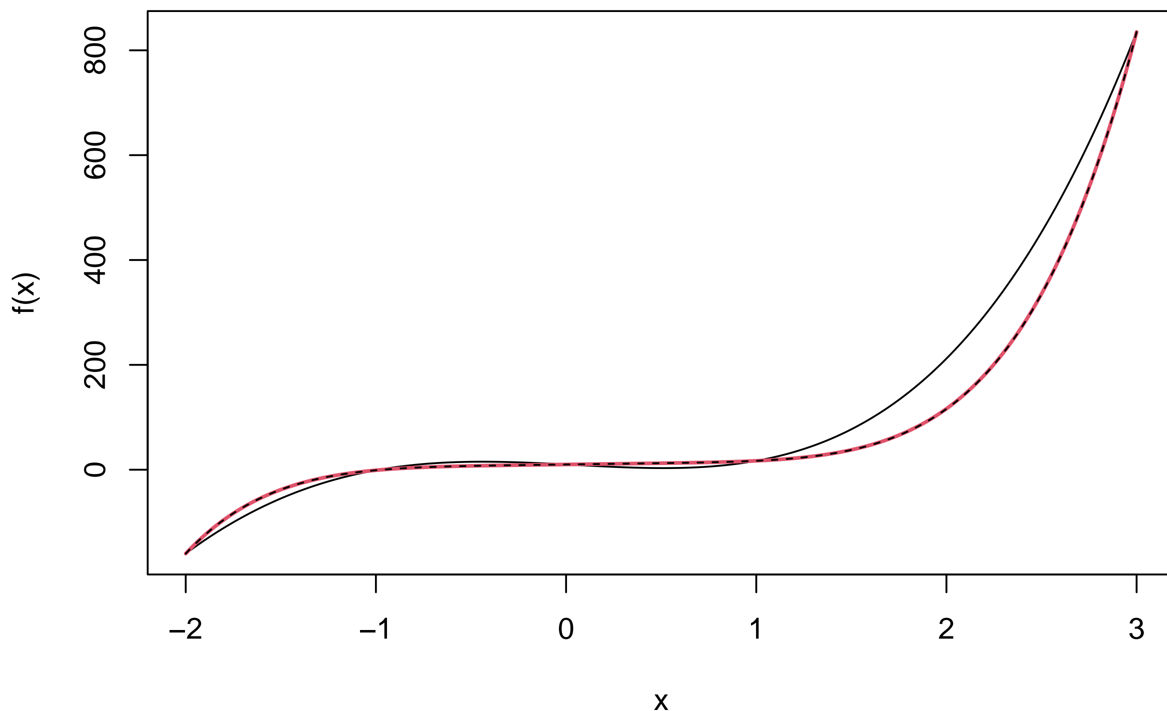
x	$f(x)$
-2	-160
-1	-1
0	10
1	17
3	835
2	116

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)
```

```
# 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
```

4 Exercises on cubic splines

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,

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

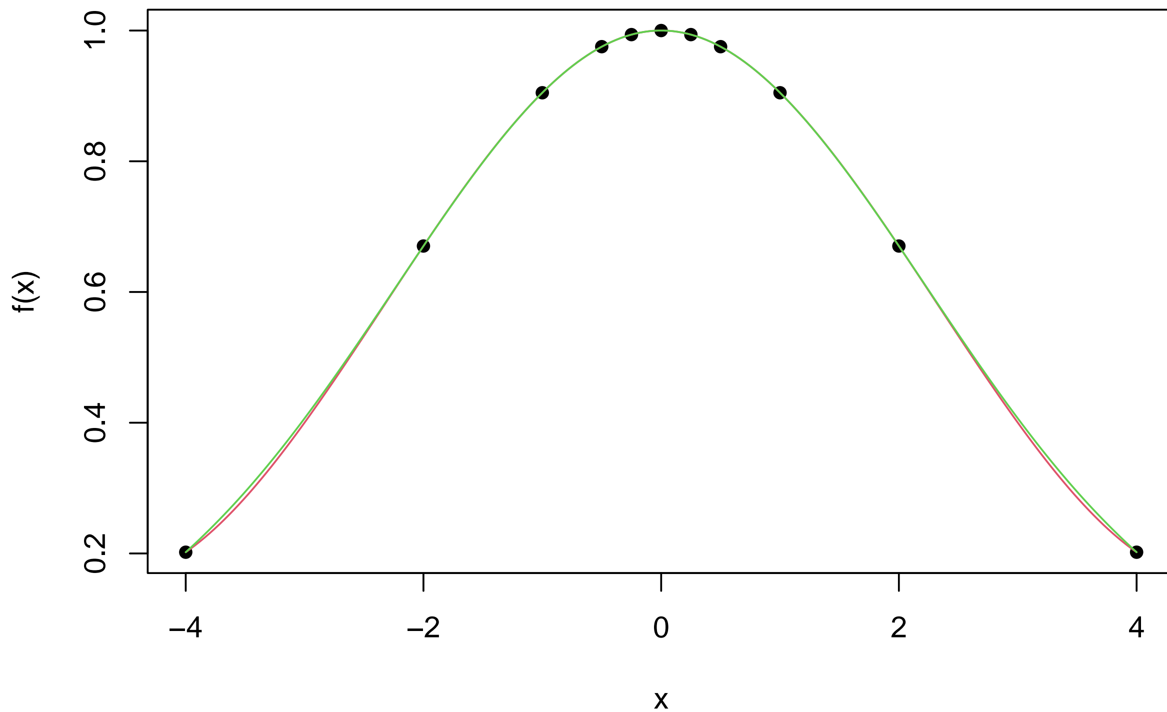
using cubic splines with the Forsythe, Malcolm and Moler method for end segments. Compare graphically the cubic splines with a polynomial fit using divided differences.

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.

4.2 Exercise 19

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

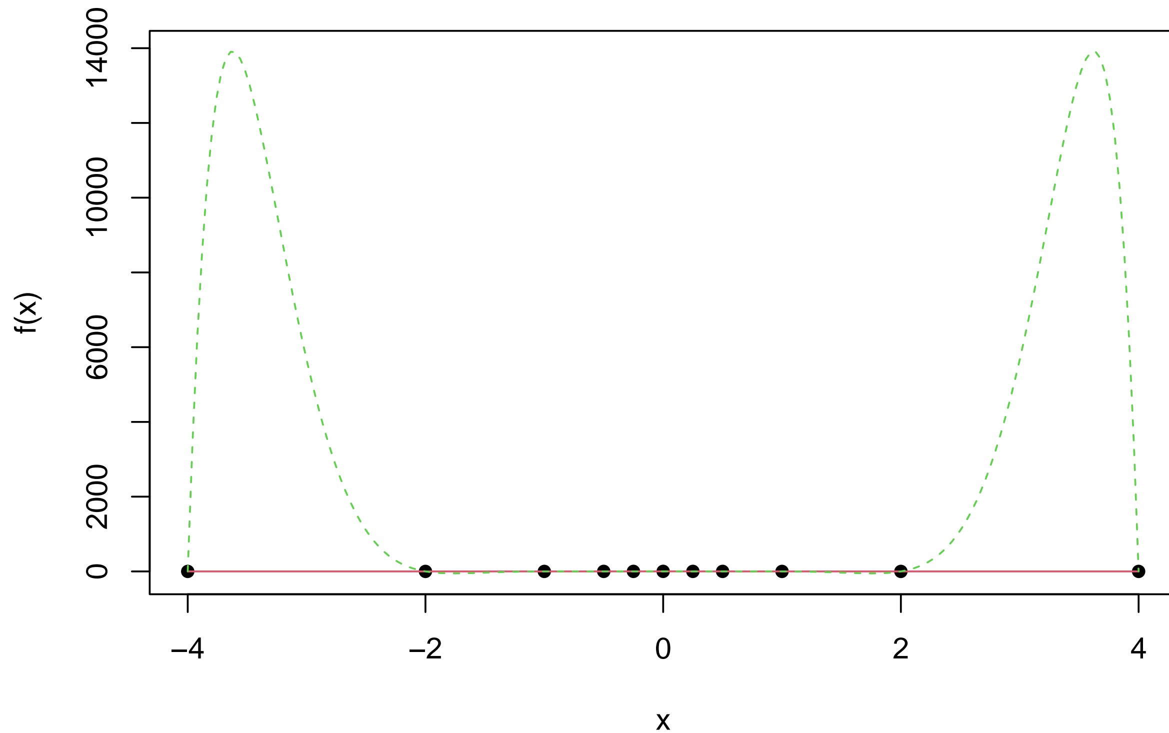
$$f(x) = exp(-x^2/0.1)$$

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.

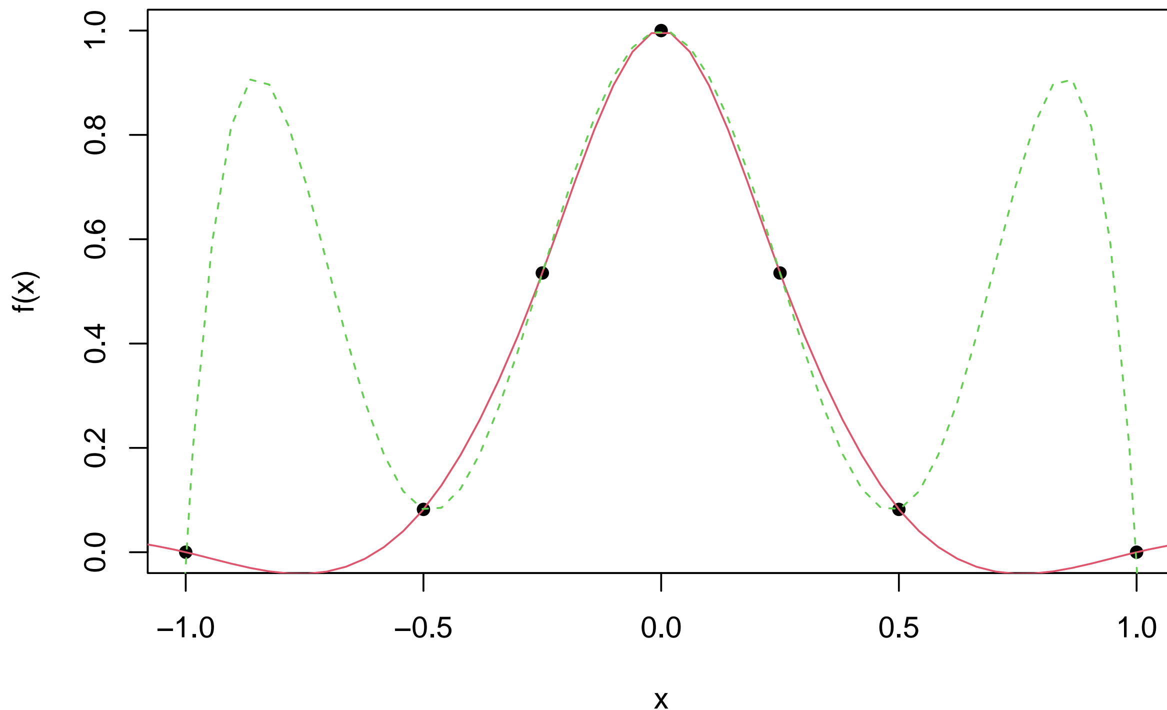
```
x <- c(-4,-2,-1,-1/2,-1/4,0,1/4,1/2,1,2,4)
f <- exp(-x^2/0.1)

# 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
```

```r
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.

```r
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)
```



4.3 Exercise 20

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

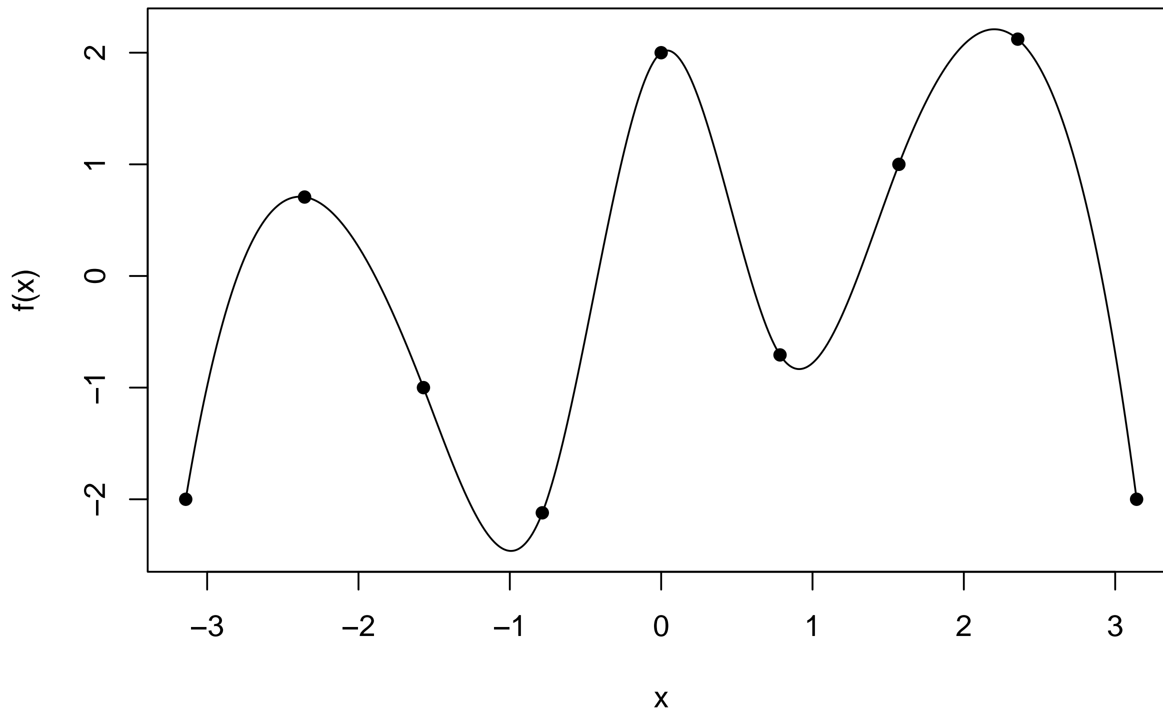
$$x_i = -\pi + (i-1)\pi/4 \ , \ \ i = 1, \ldots, 9$$
$$f_i = \sin(x_i) + 2\cos(3x_i) \ , \ \ i = 1, \ldots, 9$$

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.

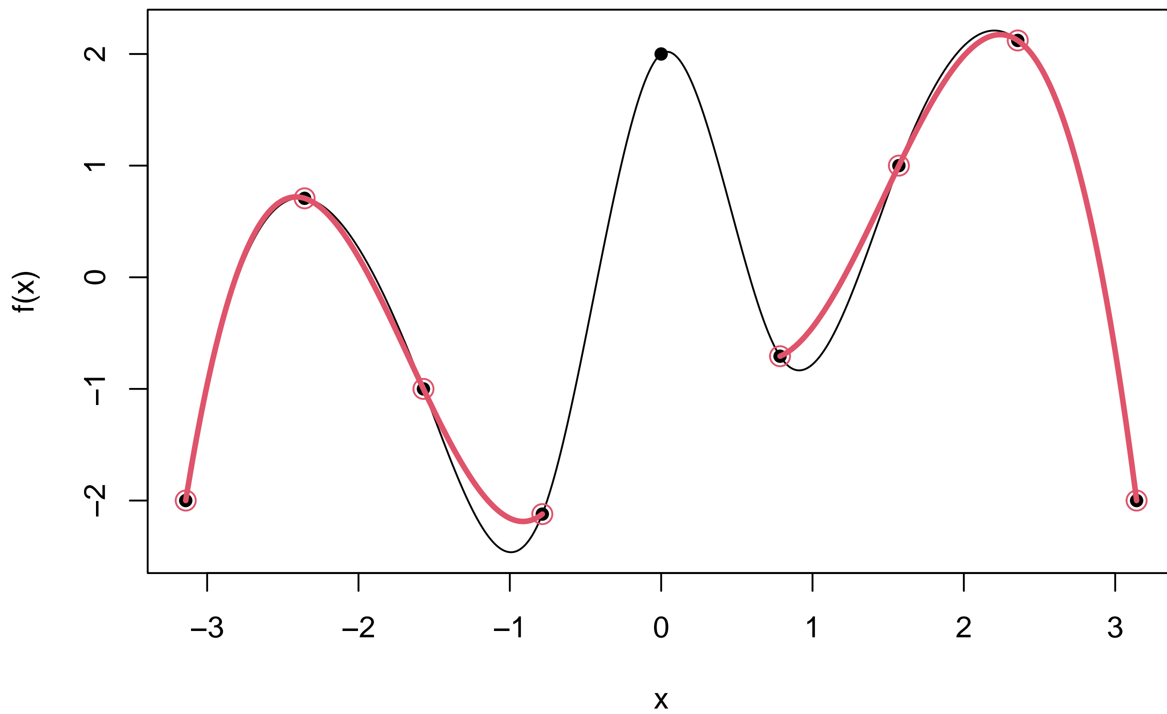
```
# Known points
x <- seq(-pi,pi,length.out=9)
print(x)
#> [1] -3.1415927 -2.3561945 -1.5707963 -0.7853982  0.0000000  0.7853982  1.5707963
#> [8]  2.3561945  3.1415927
f <- sin(x)+2*cos(3*x)

# 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.