P06 – Funktionale Programmierung

17. Mai 2021

Contents

Hinweise zur Abgabe:

Erstelle pro Aufgabe eine R-Code-Datei (in diesem Fall 3 Dateien) und benenne diese nach dem Schema `P<Woche>-<Aufgabe>.R` also hier `P06-1.R`, `P06-2.R` und `P06-3.R`. Schreibe den Code zur Lösung einer Aufgabe in die jeweilige Datei.

Es ist erlaubt (aber nicht verpflichtend) zu zweit abzugeben. Abgaben in Gruppen von drei oder mehr Personen sind nicht erlaubt. Diese Gruppierung gilt nur für die Abgabe der Programmierprobleme, nicht für die Live-Übungen.

Bei Abgaben zu zweit gibt nur eine der beiden Person ab. Dabei müssen in **jeder** abgegebenen Datei in der **ersten Zeile** als Kommentar **beide** Namen stehen also zB

```r
# Ada Lovelace, Charles Babbage

1+1
# ...
```

Die Abgabe der einzelnen Dateien (kein Archiv wie `.zip`) erfolgt über Moodle im Element namens `P06`. Die Abgabe muss bis spätestens Sonntag, 23. Mai 2021, 23:59 erfolgen.

Rekursiv definierte Folgen (10 Punkte)

Schreibe ein Funktional `rec_series(start, f, n)`, das eine `double`-Vektor $(x_1, \dots, x_n)$ ausgibt mit `start` gleich $(x_1, \dots, x_m)$ und $x_i = f(x_{i-1}, \dots, x_{i-m})$ für $i = m+1, \dots, n$. Hierbei ist m implizit durch `length(start)` gegeben.

```r
rec_series(rep(1, 2), sum, 16) # Fibonacci
##  [1]   1   1   2   3   5   8  13  21  34  55  89 144 233 377 610 987
rec_series(rep(1, 3), sum, 15)
##  [1]    1    1    1    3    5    9   17   31   57  105  193  355  653 1201 2209
rec_series(c(1,1), function(x) 2*x[1]+x[2], 12)
##  [1]    1    1    3    7   17   41   99  239  577 1393 3363 8119
rec_series(c(1,1), function(x) x[2]-x[1], 15)
##  [1]    1    1    0    1   -1    2   -3    5   -8   13  -21   34  -55   89 -144
```

Store (10 Punkte)

Schreibe eine Funktionsfabrik `make_store(n)`, die für eine positive ganze Zahl `n` eine Funktion `store(item, reset)` produziert, welche intern eine Liste mit `n` Elementen hält, die zu Beginn mit `NULL` gefüllt ist. Bei einem Aufruf `store(x)` soll `x` an die erste Stelle der Liste gesetzt werden. Beim nächsten Aufruf soll das übergebenen Element an Position 2 gesetzt werden, usw. Das $n+1$-te Element überschreibt Element 1, Das $n+2$-te Element überschreibt Element 2, usw. Das $kn+i$-te Element überschreibt Element i für $k, n, i \in \mathbb{N}$.

Gib die Liste bei jedem dieser Aufrufe `invisible()` zurück. Wird kein Element übergeben wird die Liste nicht geändert und sichtbar zurückgegeben.

`reset` steht per Default auf `FALSE`. Wird `TRUE` übergeben wird der Ausgangszustand direkt nach der Erzeugung mit `make_store()` wieder hergestellt, bevor das neue Element (an Stelle 1) eingefügt wird.

```r
store3 <- make_store(3)
store2 <- make_store(2)
store3(1)
store3("asdf")
str(store3())
## List of 3
##  $ : num 1
##  $ : chr "asdf"
##  $ : NULL
store3(list())
store3(TRUE)
store3(1:10)
str(store3())
## List of 3
##  $ : logi TRUE
##  $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ : list()
store3(NA, reset=TRUE)
str(store3())
## List of 3
##  $ : logi NA
##  $ : NULL
##  $ : NULL
store3(1:10)
str(store3())
## List of 3
##  $ : logi NA
##  $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ : NULL
store2(1)
store2(2)
store2(3)
str(store2())
## List of 2
##  $ : num 3
##  $ : num 2
str(store3())
## List of 3
##  $ : logi NA
##  $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ : NULL
```

Numerische Integration (40 Punkte)

Im Folgenden gehen wir davon aus, dass Argumente mit dem Namen `f` Funktionsobjekte sind, die eine vektorisierte Implementierung einer Funktion $f : \mathbb{R} \to \mathbb{R}$ darstellen. Dh `f(x)` für einen `double`-Vektor `x` der Länge n ergibt eine `double`-Vektor der Länge n mit den Werten $(f(x_1), \ldots, f(x_n))$.

a)

Schreibe eine Funktion `midpoint(f, a, b)`, die für $a, b \in \mathbb{R}, a < b$ die Fläche des Rechtecks mit den Ecken $(a, 0), (b, 0), (b, f(\frac{a+b}{2})), (a, f(\frac{a+b}{2}))$ ausgibt. Sind `a` und `b` `double`-Vektoren der selben Länge n, so ist die Ausgabe eine `double`-Vektor der Länge n mit den n Flächeninhalten der entsprechenden Rechtecke.

Schreibe eine analoge Funktion `trapezoid(f, a, b)`, den den Flächeninhalt des Trapez $(a, 0), (b, 0), (b, f(b)), (a, f(a)))$ berechnet.

```r
midpoint(function(x) x, (0:4)*2, (1:5)*2)
## [1]  2  6 10 14 18
midpoint(sin, (0:4)/2*pi, (1:5)/2*pi)
## [1]  1.110721  1.110721 -1.110721 -1.110721  1.110721
trapezoid(function(x) x, (0:4)*2, (1:5)*2)
## [1]  2  6 10 14 18
trapezoid(sin, (0:4)/2*pi, (1:5)/2*pi)
## [1]  0.7853982  0.7853982 -0.7853982 -0.7853982  0.7853982
```

b)

Schreibe eine Funktion `nc_integrate(f, lower, upper, n, rule)`. Wir gehen davon aus, dass jedes Argument die Länge 1 hat. `nc_integrate()` teilt das Intervall [`lower`, `upper`] in n Teilintervalle $[a_i, b_i]$ gleicher Länge und berechnet – zunächst auf den Teilintervallen – die Fläche unter dem Funktionsgraph von f mit `rule()`. Dabei übergeben wir als `rule` entweder `midpoint` oder `trapezoid`. Diese Flächen werden dann zu einer Approximation der Fläche zwischen `lower` und `upper` summiert.

```r
nc_integrate(function(x) 3*x^2, 0, 2, n=4, rule = midpoint)
## [1] 7.875
nc_integrate(function(x) 3*x^2, 0, 2, n=4, rule = trapezoid)
## [1] 8.25
nc_integrate(sin, 0, pi, n=4, rule = midpoint)
## [1] 2.052344
nc_integrate(sin, 0, pi, n=4, rule = trapezoid)
## [1] 1.896119
```

c)

Schreibe eine Funktionsfabrik `newton_cotes(coef, closed=TRUE)`, die eine `rule` zur Approximation von Integralen für `nc_integrate()` im Sinne der **Newton-Cotes-Formeln** (siehe auch https://de.wikipedia.org/wiki/Newton-Cotes-Formeln) ausgibt:

Sei `rule <- newton_cotes(coef, closed)`. Dann berechnet `rule(f, a, b)` den Wert

$$(b - a) \sum_{j=1}^{m} w_j f(t_j), \quad \text{wobei} \quad w_j = \frac{\texttt{coef}_j}{\sum_{k=1}^{m} \texttt{coef}_k}$$

und $(t_j)_{j=1,\ldots,m}$ mit $t_1 < \cdots < t_m$ das Intervall $[a, b]$ in Intervalle gleicher Länge unterteilt. Bei `closed=TRUE` ist $t_1 = a$ und $t_m = n$. Bei `closed=FALSE` unterteilen $a < t_1 < \cdots < t_m < b$ das Intervall $[a, b]$ in Teilintervaller gleicher Länge.

Die `rule()` ist wie `midpoint()` und `trapezoid()` in den Argumenten `a` und `b` vektorisiert und ruft `f()` nur einmal (auf einem entsprechenden `double`-Vektor oder -Matrix) auf. *Hinweis:* Um eine Matrix aller Stützstellen für alle Einträge der Vektoren `a`, `b` zu berechnen, nutze `outer()`.

```r
nc_integrate(function(x) 3*x^2, 0, 2, n=4, rule = midpoint)
## [1] 7.875
nc_integrate(function(x) 3*x^2, 0, 2, n=4, rule = newton_cotes(1, FALSE))
## [1] 7.875
nc_integrate(function(x) 3*x^2, 0, 2, n=4, rule = trapezoid)
## [1] 8.25
nc_integrate(function(x) 3*x^2, 0, 2, n=4, rule = newton_cotes(c(1,1)))
## [1] 8.25
simpson <- newton_cotes(c(1, 4, 1))
boole <- newton_cotes(c(7, 32, 12, 32, 7))
nc_integrate(sin, 0, pi*11, n=4, rule = simpson)
## [1] -5.041282
nc_integrate(sin, 0, pi*11, n=4, rule = boole)
## [1] 3.004235
nc_integrate(sin, 0, pi*11, n=4, rule = newton_cotes(c(2,-1,2), FALSE))
## [1] 10.04406
```

d)

Wir wollen verschiedene Newton-Cotes-Formeln auf verschieden Integale mit unterschiedlicher Anzahl an erlaubten Funktionsevaluationen testen.

Dazu erzeugen wir die Vektoren `param_fun`, `param_rule` und `param_n`, die verschiedene Paramter enthalten.

```r
# some objects for param_fun-list
sin1x <- function(x) {
  y <- suppressWarnings(sin(1/x))
  y[is.na(y)] <- 0
  y
}
set.seed(0)
x <- c(0:1, runif(5))
y <- runif(7)

# list of functions with integral interval (f, lower, upper)
param_fun <- list(
  poly = list(f = function(x) x^4 - x^3 - 3*x^2 + x + 2, lower = 0, upper = 2),
  sin1x = list(f = sin1x, lower = 0, upper = 1),
  lin = list(f = approxfun(x, y), lower = 0, upper = 1),
  spline = list(f = splinefun(x, y), lower = 0, upper = 1)
)

# options for creating integral-rules (coef, closed)
param_rule <- list(
  midpoint = list(coef = 1, closed=FALSE),
  trapezoid = list(coef = c(1,1), closed=TRUE),
  simpson = list(coef = c(1,4,1), closed=TRUE),
  boole = list(coef = c(7,32,12,32,7), closed=TRUE),
  open5 = list(coef = c(611, -453, 562, 562, -453, 611), closed=FALSE)
)

# number of allowd function evaluations
param_n <- 5*(2:20)
```

Erzeuge mit `expand.grid()` (siehe `?expand.grid`) ein Tibble aller Kombinationen der Namen von `param_fun`, `param_rule` und der Werte von `param_n`.

```
library(tidyverse)

param <- as_tibble(
  expand.grid(
    # TODO
    stringsAsFactors=FALSE))
```

```
dim(param)
## [1] 380   3
param[30*(1:10),]
## # A tibble: 10 x 3
##        n fun_name rule_name
##    <dbl> <chr>    <chr>
##  1    60 sin1x    midpoint
##  2    20 spline   midpoint
##  3    75 poly     trapezoid
##  4    35 lin      trapezoid
##  5    90 spline   trapezoid
##  6    50 sin1x    simpson
##  7    10 spline   simpson
##  8    65 poly     boole
##  9    25 lin      boole
## 10    80 spline   boole
```

Erzeuge aus `param_rule` mit `newton_cotes()` (und `do.call()`) eine Liste von `rule()`-Funktionen.

```
nc_rules <- # TODO
```

```
sapply(nc_rules, typeof)
##  midpoint trapezoid   simpson     boole     open5
## "closure" "closure" "closure" "closure" "closure"
```

Die folgende Funktion `call_nc()` nimmt die Werte einer Zeile von `param` entgegen und gibt den Werte der entsprechenden Integralapproximation aus

```
# run nc_integrate with given options
call_nc <- function(n, fun_name, rule_name) {
  opts <- param_fun[[fun_name]]
  opts$rule = nc_rules[[rule_name]]
  # to make things fair:
  # a rule which evaluates f at m points may only be called n/m times
  opts$n = round(n / length(param_rule[[rule_name]]$coef))
  do.call(nc_integrate, opts)
}
```

Nutze `call_nc()`, `mutate()` und `mapply()` (oder `rowwise()` in der neuen `dplyr`-Version), um `param` Spalten `true`, `value`, `error` hinzuzufügen. `true` ist der wahre Wert des Integrals:

```
# true  values of integrals
true <- c(
  poly = 0.4,
  sin1x = 0.504067061906928,
  lin = 0.472878602164825,
  spline = 0.97236924451286)
```

`value` ist der Wert von `nc_integrate()` aufgerufen auf das Integral beschrieben in `param$fun_name`, die

Newton-Cotes-Regel beschrieben in `param$rule_name` und die Anzahl an erlaubten Funktionsaufrufen `param$n`.

`error` ist |`value` − `true`|.

```
# TODO:
# param %>% ... -> res
```

```
dim(res)
## [1] 380   6
res[30*(1:10),]
## # A tibble: 10 x 6
##        n fun_name rule_name value  true    error
##    <dbl> <chr>    <chr>     <dbl> <dbl>    <dbl>
##  1    60 sin1x    midpoint  0.500 0.504 4.52e- 3
##  2    20 spline   midpoint  0.962 0.972 1.01e- 2
##  3    75 poly     trapezoid 0.402 0.4   1.85e- 3
##  4    35 lin      trapezoid 0.486 0.473 1.31e- 2
##  5    90 spline   trapezoid 0.976 0.972 3.75e- 3
##  6    50 sin1x    simpson   0.481 0.504 2.29e- 2
##  7    10 spline   simpson   1.22  0.972 2.51e- 1
##  8    65 poly     boole     0.4   0.4   5.55e-17
##  9    25 lin      boole     0.476 0.473 3.62e- 3
## 10    80 spline   boole     0.972 0.972 1.05e- 4
```

Zuletzt plotten wir die Ergebnisse.

```
# plot results
plots <- lapply(names(param_fun), function(nm)
  res %>%
    filter(fun_name == nm) %>%
    ggplot(aes(x = n, y = error, color = rule_name)) +
    scale_y_log10() +
    geom_line() + geom_point() + labs(title = nm)
)
gridExtra::grid.arrange(grobs = plots, nrow=2)
```