V10 - Funktionale Programmierung

17. Mai 2021

Contents

1 Funktionale Programmierung **1**
- 1.1 Reine Funktionen . . . 3
- 1.2 Funktionen höherer Ordnung . . . 4

2 Funktionale **5**
- 2.1 apply und Co . . . 5
- 2.2 Mathematische Funktionale . . . 13

3 Funktionsfabrik **15**
- 3.1 Counter . . . 17
- 3.2 Maximum Likelihood . . . 18
- 3.3 Funktionsinterpolation . . . 21

4 Funktionsoperatoren **22**
- 4.1 Verhaltensändernde Funktionsoperatoren . . . 22
- 4.2 Ausgabe-Funktionsoperatoren . . . 23
- 4.3 Eingabe-Funktionsoperatoren . . . 24

1 Funktionale Programmierung

Funktionale Programmierung ist eines von vielen **Programmierparadigmen**.

Abstrakte Lösungen von Problemen können auf vielfältige Art und Weise in einer Programmiersprache umgesetzt werden. Ein Programmierparadigma gibt einen fundamentalen Programmierstil an, der bestimmte Umsetzungen bevorzugt.

Die funktionale Programmierung ist durch folgende Eigenschaften gekennzeichnet, die größtenteils in R enthalten / umsetzbar sind:

1. **Funktionen erster Klasse**

Funktionen werden behandelt wie andere Objekte. Sie sind Funktionen erster Klasse oder *first class functions*. Insbesondere können Funktionen Argumente und Rückgabewerte von Funktionen sein.

```
sapply(mtcars, mean) # Funktion mean() als Argument
##        mpg        cyl       disp         hp       drat         wt       qsec
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500

create_fun <- function() {
  f <- function() print("Ich bin eine Funktion")
  return(f) # Funktion als Rückgabewert
}
x <- create_fun()
```

```
x()
## [1] "Ich bin eine Funktion"

str(list(mean, sd, var)) # Funktionen als Elemente einer Liste
## List of 3
##  $ :function (x, ...)
##  $ :function (x, na.rm = FALSE)
##  $ :function (x, y = NULL, na.rm = FALSE, use)
```

2. **Anonyme Funktionen**

Funktionen können ohne Namen existieren und aufgerufen werden. Solche Funktionen heißen anonym.

```
(function(x) x+2)(40)
## [1] 42
```

3. **Funktionskomposition**

Funktionen werden nicht als Abfolge von Anweisungen dargestellt, sondern als ineinander verschachtelte Funktionsaufrufe. Im Tidyverse wird mit dem Pipe-Operator ein Mittelweg bestritten.

```
library(tidyverse)
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.0     v dplyr   1.0.5
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
library(nycflights13)
flights %>%
  mutate(rank_arr_delay = rank(desc(arr_delay))) %>%
  filter(rank_arr_delay <= 3)
## # A tibble: 3 x 20
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     9      641            900      1301     1242           1530
## 2  2013     1    10     1121           1635      1126     1239           1810
## 3  2013     6    15     1432           1935      1137     1607           2120
## # ... with 12 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>, rank_arr_delay <dbl>
```

4. **Funktionsumgebung**

Funktionen können Bezug auf Variablen in der Umgebung ihrer Erstellung nehmen.

```
create_printer <- function(x) {
  function() print(x)
}
hi_printer <- create_printer("hi")
hi_printer()
## [1] "hi"
```

1.1 Reine Funktionen

Eine Funktion heißt **rein** (*pure*), wenn folgende Bedingungen erfüllt sind.

1. Bei gleicher Eingabe (Argumente) erhalten wir die gleiche Ausgabe.
2. Sie hat keine Nebenwirkungen (*side effects*).

Nebenwirkungen sind zB Ändern von globalen Variablen, Ausgabe auf Konsole, Datei erstellen.

Ein Merkmal funktionaler Programmierung ist, dass reine Funktionen gegenüber unreinen stark bevorzugt werden.

ZB ist `sin()` eine reine Funktion.

Der Begriff der reinen Funktion ist streng genommen für R wenig sinnvoll, da selbst `function() 1+2` vom +-Operator abhängig ist.

```
f <- function() 1+2
f()
## [1] 3
`+` <- `-`
f()
## [1] -1
rm(`+`) # undo
```

Wir nennen eine Funktion **reinlich**, falls sie keine Nebenwirkungen außer Warnungen oder Fehlermeldungen) hat und bei gleichen übergebenen Argumenten die gleiche Ausgabe hat, sofern die Funktion nicht durch Name-Masking beeinflusst wird.

Hinweis: **reinlich** ist in der Literatur kein gängiger Begriff.

```
function(x) x+1 # reinlich

x <- 1
function() x+2 # nicht reinlich: abhängig von x in globaler Umgebung

function() x <<- 3 # nicht reinlich: ändert x in globaler Umgebung

x <- 42 # nicht reinlich: ändert x in globaler Umgebung

h <- function(a) {
  x <- a+7
  x
}
h(8) # reinlich, obwohl im Inneren nicht reinliche Funktion benutzt wird

print(x) # nicht reinlich: Nebenwirkung: Ausgabe auf Konsole

plot(x) # nicht reinlich: Nebenwirkung: Modifikation des Graphics Devices

ggplot(tb, aes(x, y)) + geom_point() # reinlich: erzeugt Plot-Objekt, das zurückgegeben wird, print() v

log(-1) # reinlich, trotz Ausgabe von Warnung auf Konsole
```

```
f <- function() 0
call_f <- function() f()
# in der globalen Umgebung: nicht reinlich
# als Code für ein R-Paket: reinlich
```

Ein Vorteil reiner (oder reinlicher) Funktionen ist, dass sie leichter auf Korrektheit zu testen sind, da ihr Ergebnis nur von der Eingabe abhängt.

Außerdem ist die Funktionsweise beim Lesen ihres Quellcodes potentiell leichter zu verstehen, da alle relevanten Elemente direkt angegeben sind.

1.2 Funktionen höherer Ordnung

Funktionen, die Funktionen als Argumente oder als Rückgabewert haben, heißen **Funktionen höherer Ordnung**. Alle anderen Funktionen sind **Funktionen erster Ordnung**.

Funktionen höherer Ordnung lassen sich unterteilen in **Funktionale**, **Funktionsfabriken** und **Funktionsoperatoren**.

In \ Out	Vector	Function
Vector	Regular function	Function factory
Function	Functional	Function operator

Funktionale haben mindestens eine Argument der Klasse `function` und geben einen Vektor-Typen (Liste, atomarer Vektor, Matrix, Array, Data-Frame, Tibble, ...) aus.

```
randomize <- function(f) f(runif(1e3)) # Funktional
randomize(mean)
## [1] 0.5060361
randomize(mean)
## [1] 0.5007658
randomize(sum)
## [1] 488.9722
```

Funktionsfabriken bilden Vektor-Typen auf Funktionen ab.

```
add_x <- function(x) { # Funktionsfabrik
  function(a) a+x
}
add_3 <- add_x(3)
add_7 <- add_x(7)
add_3(2)
## [1] 5
add_7(2)
## [1] 9
```

Funktionsoperatoren bilden Funktionen auf Funktionen ab.

```
chatty <- function(f) { # Funktionsoperator
  function(x, ...) {
    res <- f(x, ...)
    cat("Processing ", x, "\n", sep = "")
    res
  }
}
sqr <- function(x) x ^ 2
chatty_sqr <- chatty(sqr)
```

```r
chatty_sqr(1)
## Processing 1
## [1] 1
s <- c(3, 2, 1)
sapply(s, sqr)
## [1] 9 4 1
sapply(s, chatty_sqr)
## Processing 3
## Processing 2
## Processing 1
## [1] 9 4 1
```

2 Funktionale

2.1 apply und Co

Die Funktionen der `apply`-Familie zählen zu den Funktionalen. Sie nehmen einen Vektor (oder davon abgeleiteten Datentyp) und eine Funktion als Argumente entgegen und geben einen Vektor (oder Array) aus.

Sie ersetzen häufige `for`-Schleifen-Muster durch eine kompaktere und aussagekräftigere Darstellung.

2.1.1 lapply()

Das einfachste Funktional dieser Familie ist `lapply()`. `lapply(x, f)` wendet die Funktion `f` auf jedes Element des Vektors `x` an. Das Resultat wird als Liste ausgegeben. Mit `lapply(x, f, arg1, arg2)` können der Funktion `f` weitere Argumente übergeben werden.

`lapply()` ist mittels einer `for`-Schleife implementiert.

```r
# eigene Implementierung von lapply()
lapply2 <- function(x, f, ...) {
  out <- rep(list(NULL), length(x))
  for (i in seq_along(x)) out[[i]] <- f(x[[i]], ...)
  out
}
```

Der Vorteil von `lapply(f, x)` gegenüber einer expliziten `for`-Schleife ist die kompakte Form und dadurch bessere Lesbarkeit des Codes.

Viele `for`-Schleifen-Muster können durch `lapply` ersetzt werden. Dies funktioniert immer dann, wenn der `i`-te Durchgang der `for`-Schleife nur vom `i`-ten Eintrag des Vektors und keinem der vorangegangen Ergebnisse abhängt.

```r
for (x in xs) {...}
lapply(xs, function(x) {...})

for (i in seq_along(xs)) {...} # besser als for (i in 1:length(xs)) {<...>}
lapply(seq_along(xs), function(i) {<...>})

for (nm in names(xs)) {...}
lapply(names(xs), function(nm) {...})
```

2.1.2 sapply()

Die Funktion `sapply()` führt eine Vereinfachung des Rückgabewertes durch. Dazu wird intern die Funktion `simplify2array()` genutzt.

```r
simplify2array(list(1:2, 11:12, 21:22))
##      [,1] [,2] [,3]
## [1,]    1   11   21
## [2,]    2   12   22
simplify2array(list(matrix(1:4, nrow=2), matrix(11:14, nrow=2)))
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]   11   13
## [2,]   12   14
sapply(1:3, function(x, y) c(x, y), y=0)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    0    0    0
```

```r
# eigene Implementierung von sapply()
sapply2 <- function(x, f, ...) {
  res <- lapply2(x, f, ...)
  simplify2array(res)
}
```

2.1.3 replicate()

`replicate(n, expr)` führt den Ausdruck `expr` n-mal aus und gibt das Ergebnis als Array zurück. `replicate(n, expr, simplify=FALSE)` gibt eine Liste aus. Typischerweise ist das Ergebnis von `expr` von Zufallszahlen abhängig (`rDISTRI()`-Funktionen).

```r
str(replicate(5, runif(10)))
##  num [1:10, 1:5] 0.2815 0.5305 0.4329 0.9172 0.0275 ...
str(replicate(5, runif(sample(1:10, 1)), simplify = FALSE))
## List of 5
##  $ : num [1:10] 0.621 0.357 0.536 0.449 0.175 ...
##  $ : num [1:5] 0.843 0.64 0.993 0.709 0.753
##  $ : num [1:3] 0.446 0.147 0.439
##  $ : num [1:10] 0.57 0.476 0.87 0.343 0.668 ...
##  $ : num [1:7] 0.631 0.637 0.294 0.52 0.846 ...
```

2.1.4 mapply()

`mapply()` verallgemeinert `sapply()` für mehrere Inputs. `mapply(f, x1, x2)` führt `f(x1[[i]], x2[[i]])` für jedes `i` aus. Dabei müssen die Längen der Vektoren `x1` und `x2` übereinstimmen. Weitere (konstante) Argumente können `f` mit dem Argument `MoreAgs` in einer Liste übergeben werden. Wird `SIMPLIFY=FALSE` gesetzt, wird eine Liste statt einem atomaren Typ zurückgegeben.

```r
mapply(function (x,y,z) c(x-y,y-z,z-x), 1:5, 11:15, 21:25)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  -10  -10  -10  -10  -10
## [2,]  -10  -10  -10  -10  -10
## [3,]   20   20   20   20   20
```

```r
str(mapply(function (x,y,z) c(x,y,z), 1:3, 11:13, SIMPLIFY=FALSE, MoreArgs=list(z=100)))
## List of 3
##  $ : num [1:3] 1 11 100
##  $ : num [1:3] 2 12 100
##  $ : num [1:3] 3 13 100
```

Achtung: Bei `mapply()` ist die Funktion das erste Argument, nicht das zweite wie bei `lapply()` und `sapply()`.

Achtung: Aus unerfindlichen Gründen heißt bei `mapply()` das Argument `SIMPLIFY` und bei `replicate()` und `sapply()` heißt es `simplify`

2.1.5 rollapply()

Existiert ein bestimmtes `for`-Schleifen-Muster nicht bereits in R, kann man es oft selbst implementieren.

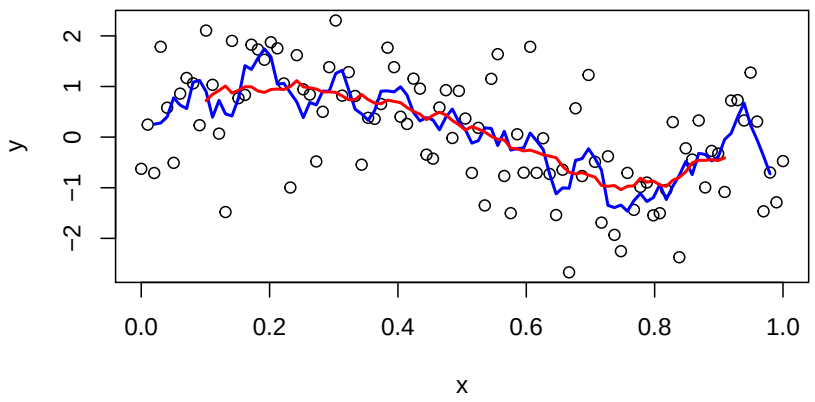
Wir implementieren die Funktion `rollmean()` um den gleitenden Mittelwert zu berechnen. Für $k \in \mathbb{N}$ ist der gleitende Mittelwert $m \in \mathbb{R}^{n-k+1}$ von $x \in \mathbb{R}^n$ definiert durch

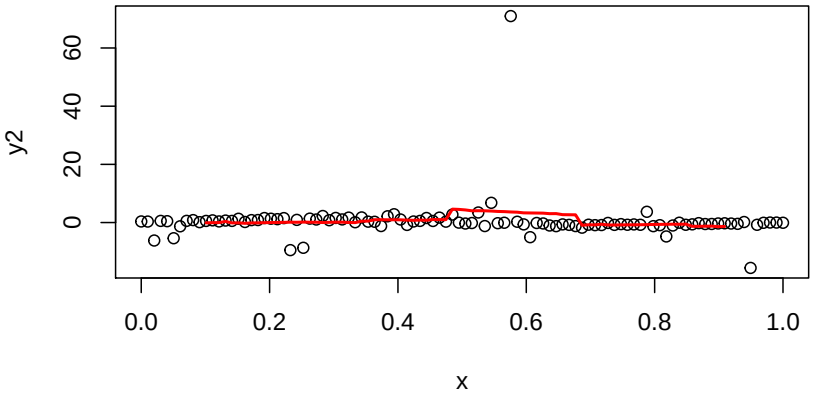
$$m_j := \frac{1}{k} \sum_{i=-\lfloor k/2 \rfloor}^{\lceil k/2 \rceil} x_{j+i}$$

```r
rollmean <- function(x, k) {
  out <- rep(NA_real_, length(x))
  neg_offset <- floor(k / 2)
  pos_offset <- ceiling(k / 2) - 1
  for (i in (1 + neg_offset):(length(x) - pos_offset)) {
    out[[i]] <- mean(x[(i - neg_offset):(i + pos_offset)])
  }
  out
}
x <- seq(0, 1, length = 1e2)
fx <- sin(2 * pi * x)

noise1 <- rnorm(1e2)
y <- fx + noise1
plot(x, y)
lines(x, rollmean(y, 5), col = "blue", lwd = 2)
lines(x, rollmean(y, 20), col = "red", lwd = 2)

noise2 <- rcauchy(1e2) / 3
y2 <- fx + noise2
plot(x, y2)
lines(x, rollmean(y2, 20), col = "red", lwd = 2)
```

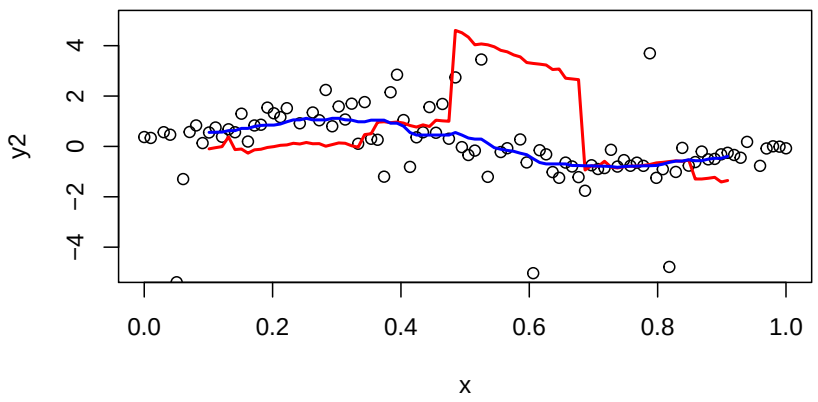




Für unseren zweiten Datensatz scheint `rollmean` unangebracht. Wir würden stattdessen gerne eine entsprechende Funktion `rollmedian()` anwenden. Da dafür das gleiche `for`-Schleifen-Muster benötigt wird, bietet es sich an, eine eigene `apply`-Funktion dafür zu implementieren.

```
rollapply <- function(x, k, f, ...) {
  out <- rep(NA_real_, length(x))
  neg_offset <- floor(k / 2)
  pos_offset <- ceiling(k / 2) - 1
  for (i in (1 + neg_offset):(length(x) - pos_offset)) {
    out[[i]] <- f(x[(i - neg_offset):(i + pos_offset)], ...)
  }
  out
}

plot(x, y2, ylim=c(-5,5))
lines(x, rollapply(y2, 20, mean), col = "red", lwd = 2)
lines(x, rollapply(y2, 20, median), col = "blue", lwd = 2)
```



2.1.6 Schleifen vs Funktionale

Benutzung von `apply`-Funktionalen führt zu übersichtlicheren Programmen und besser lesbarem Code.

Effizienz ist nicht der Grund, warum `lapply()` statt einer `for`-Schleife genutzt werden sollte. Im Hintergrund einer `apply`-Funktion wird eine Schleife durchlaufen.

```
N <- 1e5
x <- runif(N)
out <- double(N)

# vergleiche:
system.time(for (i in 1:N) out[[i]] <- x[[i]]^2)
##    user  system elapsed
##    0.02    0.00    0.02
# mit:
system.time(sapply(x, `^`, 2))
##    user  system elapsed
##    0.04    0.00    0.05

# eigentlich würde man natürlich so vorgehen:
system.time(x^2)
##    user  system elapsed
##       0       0       0
```

Explizite Schleifen lassen sich in bestimmten Situationen nur schwer vermeiden. Es ist nicht immer möglich, `while`-Schleifen in eine `apply`-Form zu bringen.

2.1.7 apply()

`apply(A, d, f, ...)` führt die Funktion `f` auf den Dimensionen `d` des Arrays `A` aus. `d` kann ein numerischer oder, falls Dimensionen benannt sind, ein `character` Vektor sein.

`d` gibt diejenigen Dimensionen an, über die iteriert wird. Das Komplement sind diejenigen Dimensionen, die an `f` übergeben werden.

Die Ausgabe von `f` wird als Vektor interpretiert (Dimensionen werden ignoriert) und entlang der ersten Dimension geschrieben.

```r
# Matrix (2D)
m <- matrix(1:12, nrow=3)
dimnames(m) <- list(zeilen=character(0), spalten=character(0))
m
##       spalten
## zeilen [,1] [,2] [,3] [,4]
##   [1,]    1    4    7   10
##   [2,]    2    5    8   11
##   [3,]    3    6    9   12
apply(m, 1, range) # iteriere über alle Zeilen
##        zeilen
## spalten [,1] [,2] [,3]
##    [1,]    1    2    3
##    [2,]   10   11   12
# dimnames der Ausgabe von apply ist etwas verwirrend. Besser wäre
fix_nm <- function(arr, name) {
  names(dimnames(arr))[[1]] <- name
  return(arr)
}
fix_nm(apply(m, 1, range), "range")  # iteriere über alle Zeilen
##       zeilen
## range  [,1] [,2] [,3]
##   [1,]    1    2    3
##   [2,]   10   11   12
fix_nm(apply(m, "spalten", range), "range") # iteriere über alle Spalten
##       spalten
## range  [,1] [,2] [,3] [,4]
##   [1,]    1    4    7   10
##   [2,]    3    6    9   12
```

Sei $\mathsf{Array}(n_1, \ldots, n_N)$ die Menge der Arrays mit N Dimensionen der Länge $n_1, \ldots, n_N$, dh $\mathsf{Array}(n_1, \ldots, n_N) \subset \mathbb{R}^{n_1 \times \cdots \times n_N}$ für `double`-Arrays.

Sei $\mathsf{Vector}(m)$ die Menge der Vektoren mit Länge m.

Allgemein gilt:

- Für ein Array $A \in \mathsf{Array}(n_1, \ldots, n_N)$ mit N Dimensionen,
- einen Dimensions-Index-Vektor $d = (d_1, \ldots, d_D) \in \{1, \ldots, N\}$ mit $d_i \neq d_j$ für $i \neq j$
- und eine Funktion $f : \mathsf{Array}(n_{\bar{d}_1}, \ldots, n_{\bar{d}_{\bar{D}}}) \to \mathsf{Vektor}(m)$, wobei $\bar{D} = N - D$ und $\{\bar{d}_1, \ldots, \bar{d}_{\bar{D}}\} = \{1, \ldots, N\} \setminus \{d_1, \ldots, d_D\}$,
- gilt $Y := \texttt{apply}(A, d, f) \in \mathsf{Array}(m, n_{d_1}, \ldots, n_{d_D})$.
- Dabei gilt $Y[\cdot, i_1, \ldots, i_D] = f(A[\ldots, i_1, , \ldots, , i_2, , \ldots, , i_D, , \ldots])$, wobei auf der rechten Seite i_j an der Stelle d_j steht, für $i_j \in \{1, \ldots, n_{d_j}\}$.

Beispiel mit $N = 5$, $D = 2$, $m = 20$:

- $A \in \mathsf{Array}(11, 12, 13, 14, 15)$
- $d = (3, 4)$
- $f : \mathsf{Array}(11, 12, 15) \to \mathsf{Vektor}(20)$
- $Y := \texttt{apply}(A, d, f) \in \mathsf{Array}(20, 13, 14)$
- $Y[\cdot, i_1, i_2] = f(A[\cdot, \cdot, i_1, i_2, \cdot])$ für $i_1 \in \{1, \ldots, 13\}$, $i_2 \in \{1, \ldots, 14\}$

```r
# Beispiel mit N=3
a <- array(1:24, dim = 2:4, dimnames = list(x1=character(0), x2=character(0), x3=character(0)))
a
## , , 1
```

```
##
##       x2
## x1     [,1] [,2] [,3]
##   [1,]    1    3    5
##   [2,]    2    4    6
##
## , , 2
##
##       x2
## x1     [,1] [,2] [,3]
##   [1,]    7    9   11
##   [2,]    8   10   12
##
## , , 3
##
##       x2
## x1     [,1] [,2] [,3]
##   [1,]   13   15   17
##   [2,]   14   16   18
##
## , , 4
##
##       x2
## x1     [,1] [,2] [,3]
##   [1,]   19   21   23
##   [2,]   20   22   24
# D=1, m=1
fix_nm(apply(a, 1, colSums), "colSums")
##        x1
## colSums [,1] [,2]
##    [1,]    9   12
##    [2,]   27   30
##    [3,]   45   48
##    [4,]   63   66
fix_nm(apply(a, "x2", colSums), "colSums")
##        x2
## colSums [,1] [,2] [,3]
##    [1,]    3    7   11
##    [2,]   15   19   23
##    [3,]   27   31   35
##    [4,]   39   43   47
# D=2, m=1
apply(a, c(1,2), sum)
##       x2
## x1     [,1] [,2] [,3]
##   [1,]   40   48   56
##   [2,]   44   52   60
# D=2, m=2
apply(a, c("x3","x1"), range)
## , , 1
##
##       x3
## x2     [,1] [,2] [,3] [,4]
```

```
##    [1,]    1    7   13   19
##    [2,]    5   11   17   23
##
## , , 2
##
##       x3
## x2      [,1] [,2] [,3] [,4]
##    [1,]    2    8   14   20
##    [2,]    6   12   18   24
```

2.1.8 outer()

`outer()` berechnet das dyadische (äußere) Produkt zweier Arrays.

Für Vektoren a, b gilt $Y = \texttt{outer}(a, b, f)$ mit $Y[i, j] = f(a[i], b[j])$.

`f` muss dabei vektorisiert sein, da `f` nur einmal auf entsprechend *recycelten* Arrays `A`, `B` aufgerufen wird, sodass $f(A, B) = Y = \texttt{outer}(a, b, f)$.

```
outer(1:3, 1:5, `*`) # Multiplikationstabelle
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    2    4    6    8   10
## [3,]    3    6    9   12   15
outer(1:3, 1:5, `+`) # Summantionstabelle
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    3    4    5    6
## [2,]    3    4    5    6    7
## [3,]    4    5    6    7    8
outer(1:3, 1:5, pmin)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    1    1    1
## [2,]    1    2    2    2    2
## [3,]    1    2    3    3    3
```

Allgemein gilt:

- Für $a \in \mathsf{Array}(n_1, \ldots, n_N)$, $b \in \mathsf{Array}(m_1, \ldots, m_M)$
- und $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ vektorisiert
- ist $\texttt{outer}(a, b, f) = f(A, B) \in \mathsf{Array}(n_1, \ldots, n_N, m_1, \ldots, m_M)$
- mit $A, B \in \mathsf{Array}(n_1, \ldots, n_N, m_1, \ldots, m_M)$, sodass $A[i_1, \ldots, i_N, j_1, \ldots, j_M] = a[i_1, \ldots, i_N]$ und $B[i_1, \ldots, i_N, j_1, \ldots, j_M] = b[j_1, \ldots, j_M]$ für alle $i.$, $j.$.

Angenommen wir haben eine Liste von Funktionen und eine Liste von Datensätzen und möchten jede Funktion auf jeden Datensatz anwenden.

`outer()` kann auch auf Listen angewendet werden, wenn die Funktion `f` für Listen vektorisiert ist.

```
funs <- list(mean=mean, median=median, mid=function(x)(max(x)-min(x))/2)
X <- replicate(5, runif(sample(20,1)), simplify=FALSE)
str(X)
## List of 5
##  $ : num [1:14] 0.8744 0.0861 0.9984 0.7357 0.1859 ...
##  $ : num [1:11] 0.8918 0.6811 0.6772 0.2367 0.0857 ...
##  $ : num [1:14] 0.887 0.621 0.91 0.222 0.159 ...
##  $ : num [1:19] 0.484 0.513 0.746 0.95 0.53 ...
##  $ : num [1:14] 0.583 0.0304 0.4273 0.2546 0.9599 ...
```

```r
call <- function(x_list, f_list) mapply(function(f,x) f(x), f_list, x_list)
outer(X, funs, call)
##           mean    median       mid
## [1,] 0.4722397 0.4776795 0.4561486
## [2,] 0.5943356 0.6656311 0.4411423
## [3,] 0.5489152 0.6404765 0.4185863
## [4,] 0.5186691 0.5095012 0.4229157
## [5,] 0.5359626 0.5687091 0.4647491
```

2.1.9 Weitere

- `vapply()`: wie `sapply()` mit Angabe von Ausgabe-Prototyp.
- `rapply()`: rekursiv über geschachtelte Listen

2.2 Mathematische Funktionale

Das Paket `stats`, das beim Start von R automatisch geladen wird, enthält unter anderem die Funktionale:

- `integrate()`: integriere eine Funktion
- `optimize()` (bzw `optim()` oder `constrOptim()`): minimiere eine Funktion

2.2.1 Integrieren

Für die vektorisierte Implementierung `f()` einer Funktion $f : [a, b] \to \mathbb{R}$ nähert `integrate(f, lower=a, upper=b)` den Wert $\int_a^b f(x)\mathrm{d}x$ durch numerische Integration an.

`f()` muss vektorisiert sein, dh für einen `double`-Vektor `x` soll `f(x)` gleich `sapply(x,f)` sein.

```r
int <- integrate(sin, 0, pi)
print(int)
## 2 with absolute error < 2.2e-14
str(int)
## List of 5
##  $ value       : num 2
##  $ abs.error   : num 2.22e-14
##  $ subdivisions: int 1
##  $ message     : chr "OK"
##  $ call        : language integrate(f = sin, lower = 0, upper = pi)
##  - attr(*, "class")= chr "integrate"
int$value
## [1] 2
```

`integrate()` wählt automatisch die Unterteilung in Unterintervalle. Die genaue Steuerung des benutzten Algorithmus kann durch weitere Argumente der Funktion übergeben werden. Siehe `?integrate`.

Wir implementieren unsere eigene Version von `integrate()` mittels einer Riemann-Summe.

```r
integrate2 <- function(f, lower, upper, subintervals) {
  x <- seq(lower, upper, length.out = subintervals+1)
  sum(f(x[-1])) / subintervals * (upper-lower)
}
integrate2(sin, 0, pi, 10)
## [1] 1.983524
integrate2(sin, 0, pi, 1000)
## [1] 1.999998
```

Achtung: Numerische Integration ist nicht trivial. Der für `integrate()` implementierte Algorithmus ist zwar deutlich effizienter als die Riemann-Summe. Allerdings wird eine Funktion, die in weiten Teilen nahezu konstant ist, ggf als konstant interpretiert.

```
density <- function(q) dnorm(q, sd=0.01, mean=1) # integral is 1
integrate(density, lower=-10, upper=10)
## 0 with absolute error < 0
integrate2(density, lower=-10, upper=10, 1000)
## [1] 1.014384
```

2.2.2 Optimieren

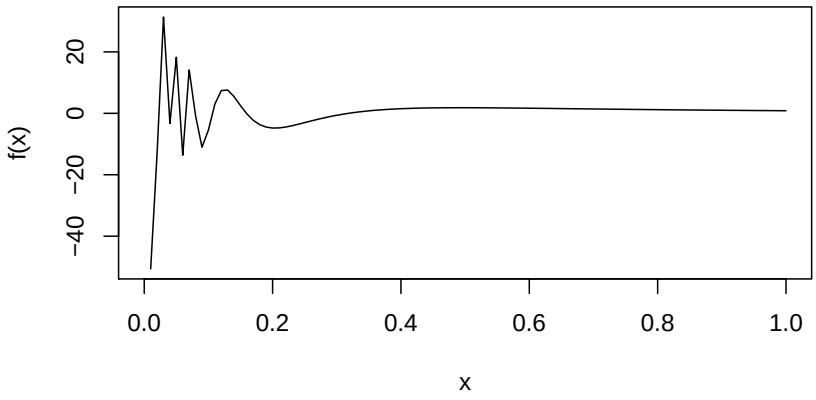
Die Funktion `optimize()` minimiert eine gegebene eindimensionale Funktion innerhalb eines gegebenen Intervalls.

```
opti <- optimize(sin, c(0, 2 * pi))
str(opti)
## List of 2
##  $ minimum  : num 4.71
##  $ objective: num -1
opti$minimum / pi
## [1] 1.500001
```

`optim()` kann Funktionen mit höher-dimensionalen Inputs minimieren.

Beachte, dass numerische Minimierung allgemeiner Funktionen eine schwierige Aufgabe ist, die viel Rechenzeit beanspruchen kann und ggf nicht zufriedenstellende Ergebnisse liefert (zB lokale Minima).

```
f <- function(x) 1/x*sin(1/x)
str(optimize(f, c(0, 1))) # lokales Minimum
## List of 2
##  $ minimum  : num 0.204
##  $ objective: num -4.81
curve(f)
## Warning in sin(1/x): NaNs produced
```



In vielen Fällen sind die Ergebnisse nicht zufriedenstellend. Es empfiehlt sich einen simplen (nicht notwendigerweise effizienten) Algorithmus zum Vergleich zu implementieren: Grid-Search!

```
grid_optim <- function(f, lower, upper, grid_cnt = 5)  {
  grid_vectors <- lapply(seq_along(lower), function(i) seq(lower[[i]], upper[[i]], length.out = grid_cn
  grid <- as.matrix(expand.grid(grid_vectors))
  v <- apply(grid, 1, f)
  list(par = grid[which.min(v), ], value = min(v))
}

compare_optim <- function(f, lower, upper, grid_cnt = 5) {
  res_grid <- grid_optim(f, lower, upper, grid_cnt)
  res_optim <- optim((lower+upper)/2, f, method="L-BFGS-B", lower=lower, upper=upper)
  cat("grid: val=",res_grid$value," par: ",res_grid$par,"\n")
  cat("optim: val=",res_optim$value," par: ",res_optim$par,"\n")
}

f <- function(x) sum(x^2)
compare_optim(f, c(-3,-1), c(10,20))
## grid: val= 1.0625  par:  0.25 -1
## optim: val= 7.108619e-27  par:  7.019422e-14 -4.670537e-14

g <- function(x) { # has areas of constant value
  z <- sum(x^2)
  if (z < 1) return(0)
  if (z < 2) return(1)
  return(-1)
}
compare_optim(g, c(-10,-10), c(10,10))
## grid: val= -1  par:  -10 -10
## optim: val= 0  par:  0 0
```

3 Funktionsfabrik

Eine Funktionsfabrik erstellt eine Funktion aus einem Vektor-Typ.

```
power1 <- function(exp) { # eine einfache Funktionsfabrik
  force(exp) # Lazy-Evaluation
  function(x) {
    x ^ exp
  }
}

square <- power1(2)
cube <- power1(3)
square(3)
## [1] 9
cube(3)
## [1] 27
```

Wir erinnern uns an den Begriff der Funktionsumgebung.

```
library(rlang)
```

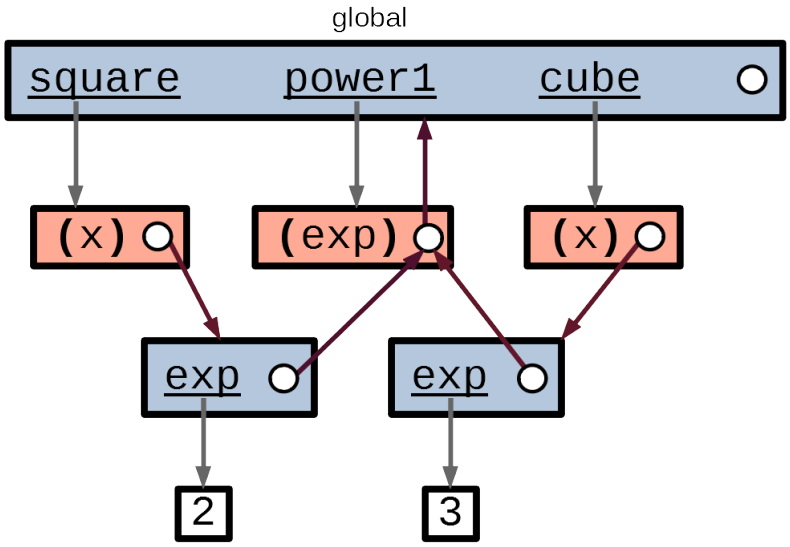
```
env_print(fn_env(square))
## <environment: 000000001C7B60A8>
## parent: <environment: global>
```

```
## bindings:
##  * exp: <dbl>
env_print(fn_env(cube))
## <environment: 000000001C752760>
## parent: <environment: global>
## bindings:
##  * exp: <dbl>
```

Mit `env_get()` können wir den Wert einer Namensbindung in einer Umgebung abfragen.

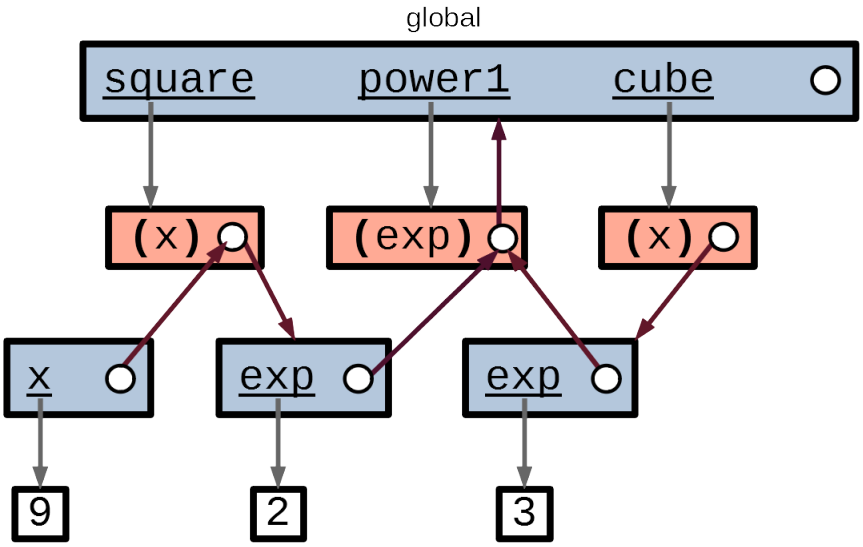
```
env_get(fn_env(square), "exp")
## [1] 2
env_get(fn_env(cube), "exp")
## [1] 3
```

Folgendes Diagramm veranschaulicht die beiden produzierten Funktionen.



Beim Aufruf von `square(9)` wird eine Ausführungsumgebung mit der Namensbindung `x --> 9` erzeugt.

```
square(9)
## [1] 81
```



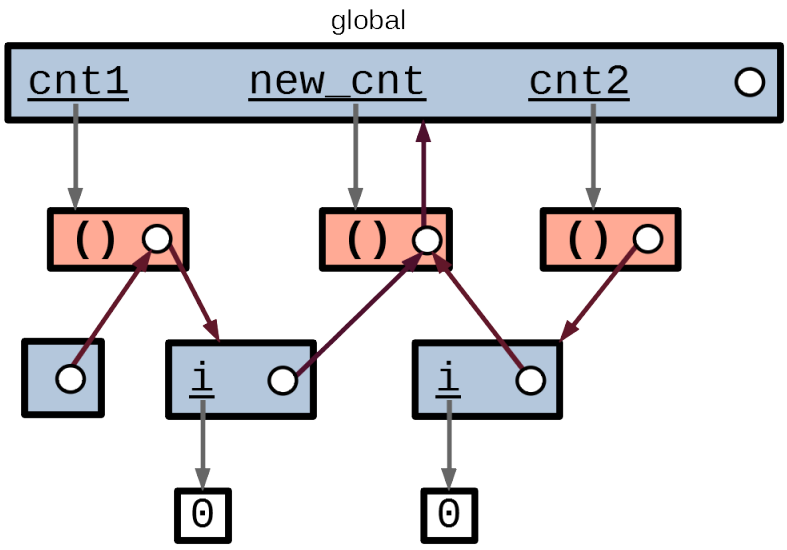
3.1 Counter

Unter Benutzung des Zuweisungsoperators `<<-` bauen wir eine Funktionsfabrik, die Zählfunktionen erstellt.

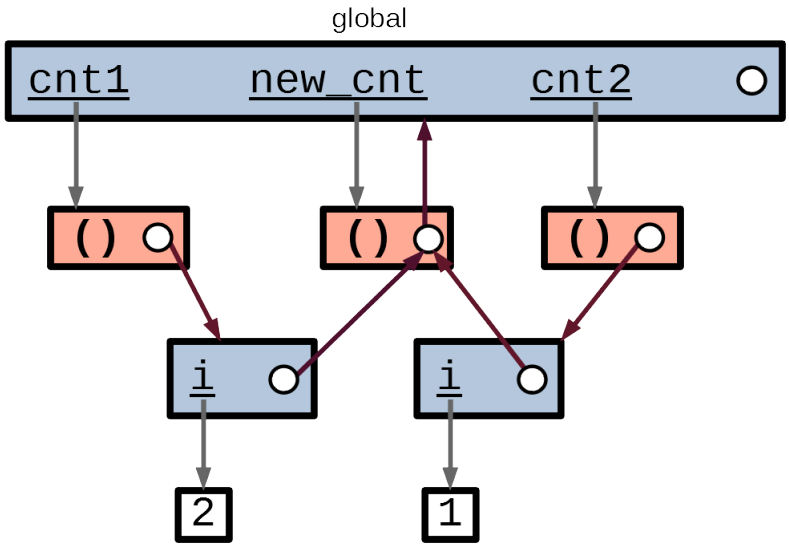
```
new_cnt <- function() {
  i <- 0
  function() {
    i <<- i + 1
    i
  }
}
cnt1 <- new_cnt()
cnt2 <- new_cnt()
```

Erinnerung: Die Ausführungsumgebung von `new_cnt()` wird bei jedem Aufruf von `new_cnt()` neu erstellt und übernimmt dann Rolle der Funktionsumgebung der produzierten Funktion `cnt1()` bzw. `cnt2()`.

Beim Aufruf von `cnt1()` entsteht auch eine Ausführungsumgebung. Diese enthält allerdings keine Namensbindungen.




```
cnt1()
## [1] 1
cnt1()
## [1] 2
cnt2()
## [1] 1
```



3.2 Maximum Likelihood

Wir haben $n \in \mathbb{N}$ Beobachtungen $X \in \mathbb{N}_0^n$ bzw `X` aus den natürliche Zahlen mit 0 und gehen davon aus, dass diese unabhängig aus einer Poisson–Verteilung gezogen wurden.

```
X <- c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)
```

Wir wissen jedoch nicht welchen Parameter λ die zugrunde liegende Verteilung hat und möchten diesen aus den Daten schätzen.

Eine Möglichkeit ist der Maximum-Likelihood-Schätzer: Wir betrachten die Zähldichte $p_{n,\lambda}\colon \mathbb{N}_0^n \to [0,\infty)$ der multivariaten Poisson-Verteilung zum Parameter λ, setzen unsere Beobachtungen X ein und maximieren $\lambda \mapsto p_{n,\lambda}(X)$ bzgl $\lambda \in (0,\infty)$.

Für $k \in \mathbb{N}_0$ gilt

$$p_{1,\lambda}(k) = \exp(-\lambda)\frac{\lambda^k}{k!}$$

und für $x \in \mathbb{N}_0^n$ wegen Unabhängigkeit

$$p_{n,\lambda}(x) = \prod_{i=1}^{n} p_{1,\lambda}(x_i) = \exp(-\lambda n)\prod_{i=1}^{n}\frac{\lambda^{x_i}}{x_i!}\,.$$

Häufig wird statt der Dichte selbst ihr Logarithmus maximiert, was das gleiche Ergebnis liefert, da $x \mapsto \log(x)$ streng monoton wachsend ist. Es gilt

$$\log(p_{n,\lambda}(x)) = -\lambda n + \sum_{i=1}^{n}(x_i\log(\lambda) - \log(x_i!)) = -\lambda n + \log(\lambda)\sum_{i=1}^{n} x_i - \sum_{i=1}^{n}\log(x_i!)\,.$$

```
log_n_dpois <- function(lambda, x) {
  n <- length(x)
  -n*lambda + log(lambda)*sum(x) - sum(lfactorial(x))
}
```

Wir setzen verschiedene Werte für λ ein. Höhere Werte deuten auf ein λ hin, das besser zu den Daten passt.

```
log_n_dpois(10, X)
## [1] -183.6405
log_n_dpois(30, X)
## [1] -30.98598
log_n_dpois(50, X)
## [1] -67.01095
```

Da die Daten fix sind, können wir eine Funktion $\lambda \mapsto \log(p_{n,\lambda}(x))$ implementieren, der wir das `x` nicht übergeben müssen. An dieser Stelle ist eine Funktionsfabrik sinnvoll.

```
make_log_n_dpois_x <- function(x) {
  n <- length(x)
  function(lambda) {
    -n*lambda + log(lambda)*sum(x) - sum(lfactorial(x))
  }
}
log_n_dpois_x <- make_log_n_dpois_x(X)
log_n_dpois_x(10)
## [1] -183.6405
log_n_dpois_x(30)
## [1] -30.98598
log_n_dpois_x(50)
## [1] -67.01095
```

Wir bemerken, dass einige Terme der log-Dichte nicht von λ sondern nur von x abhängen. Es genügt diese einmal zu berechnen.

```
make_log_n_dpois_x <- function(x) {
  n <- length(x)
  sum_x <- sum(x)
  sum_fac_x <- sum(lfactorial(x))
```

```
  function(lambda) {
    -n*lambda + log(lambda)*sum_x - sum_fac_x
  }
}
log_n_dpois_x <- make_log_n_dpois_x(X)
log_n_dpois_x(10)
## [1] -183.6405
log_n_dpois_x(30)
## [1] -30.98598
log_n_dpois_x(50)
## [1] -67.01095
```

Wir wollen natürlich nicht von Hand verschiedene Werte einsetzen, sondern automatisch das Maximum finden. Hierzu dient die Funktion `optimize()`.

```
str(optimize(log_n_dpois_x, c(0, 100), maximum = TRUE))
## List of 2
##  $ maximum  : num 32.1
##  $ objective: num -30.3
# alternativ:
str(optimize(log_n_dpois, c(0, 100), x = X, maximum = TRUE))
## List of 2
##  $ maximum  : num 32.1
##  $ objective: num -30.3
```

Um eine gute Näherung des Maximums zu finden, muss `optimize()` die ihr übergebene Funktion ggf sehr häufig aufrufen. Eine effiziente Implementierung dieser Funktion ist daher von Nutzen.

```
n <- 1e6
Y <- rpois(n, 42)
log_n_dpois_y <- make_log_n_dpois_x(Y)

t <- proc.time()
str(optimize(log_n_dpois_y, c(0, 100), maximum = TRUE))
## List of 2
##  $ maximum  : num 42
##  $ objective: num -3285785
print(proc.time()-t)
##    user  system elapsed
##       0       0       0

t <- proc.time()
str(optimize(log_n_dpois, c(0, 100), x = Y, maximum = TRUE))
## List of 2
##  $ maximum  : num 42
##  $ objective: num -3285785
print(proc.time()-t)
##    user  system elapsed
##    0.75    0.01    0.77
```

Funktionsfabriken haben hier zwei positive Effekte:

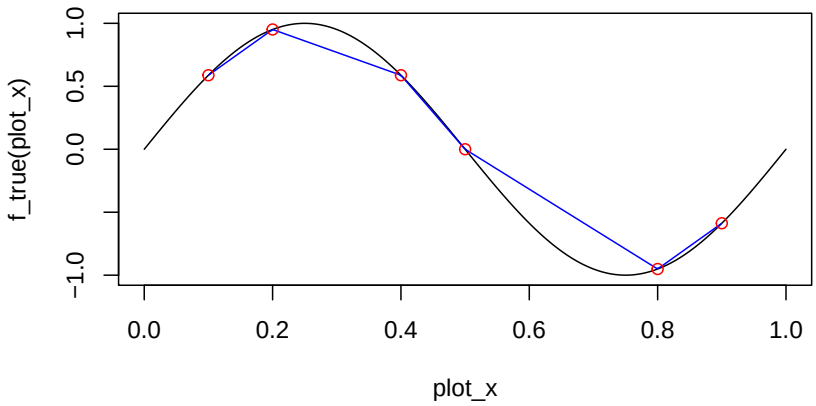
- Wir können einige Werte in der Funktionsfabrik selbst vorherberechnen und damit etwas CPU-Zeit sparen.
- Das zweistufige Design reflektiert die mathematische Struktur des zugrunde liegenden Problems.

Diese Vorteile werden größer bei komplexeren Maximum-Likelihood-Problemen mit mehreren Parametern und Datenvektoren.

3.3 Funktionsinterpolation

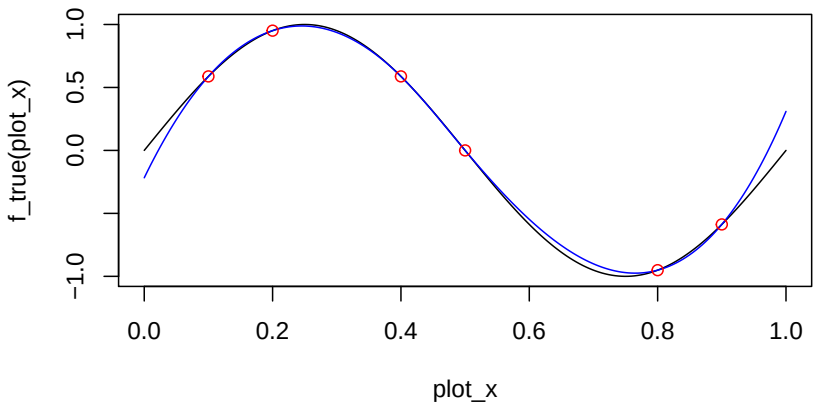
Die Funktionsfabrik `approxfun()` nimmt Werte $(x_i, y_i)_{i=1,\ldots,n}$ entgegen und gibt eine Funktion f aus, sodass $f(x_i) = y_i$. Für Zwischenwerte $x \notin \{x_1, \ldots, x_n\}$ wird ein linear interpolierter Wert zurückgegeben.

```
# erzeuge Daten
f_true <- function(x) sin(x*pi*2)
x <- c(0.1, 0.2, 0.4, 0.5, 0.8, 0.9)
y <- f_true(x)
# berechne (lineare) Interpolationsfunktion
f_lin <- approxfun(x, y)
# plotte Resultate
plot_x <- seq(0, 1, length.out = 200)
plot(plot_x, f_true(plot_x), type = "l")
lines(plot_x, f_lin(plot_x), col="blue")
points(x, y, col="red")
```



Analog zur linearen Interpolation durch `approxfun()` führt `splinefun()` eine Interpolation mit kubischen Splines (Polynome dritten Grades) durch.

```
# berechne (kubische) Interpolationsfunktion
f_cub <- splinefun(x, y)
# plotte Resultate
plot_x <- seq(0, 1, length.out = 200)
plot(plot_x, f_true(plot_x), type = "l")
lines(plot_x, f_cub(plot_x), col="blue")
points(x, y, col="red")
```



4 Funktionsoperatoren

Funktionsoperatoren bilden Funktionen auf Funktionen ab. Aus der Mathematik ist etwa der Ableitungsoperator $D : \mathcal{C}^1(\mathbb{R}) \to \mathcal{C}^0(\mathbb{R})$ bekannt. In R werden Funktionsoperatoren meist nicht für rein mathematische Zwecke eingesetzt. Einige Beispiele behandeln wir im Folgenden.

4.1 Verhaltensändernde Funktionsoperatoren

Zähle Funktionsaufrufe und schreibe `.` auf die Konsole, um eine Fortschrittsanzeige zu erzeugen.

```
dot_every <- function(n, f) {
  i <- 1
  function(...) {
    if (i %% n == 0) cat(".")
    i <<- i + 1
    f(...)
  }
}
x <- lapply(1:100, runif)
x <- lapply(1:100, dot_every(10, runif))
## ..........
```

Um den Effekt sichtbar zu machen (in der live-Ausführung), wollen wir die Ausführung der Funktion `runif()` künstlich verzögern. Wir nutzen dazu `Sys.sleep()` und schreiben einen entsprechenden Funktionsoperator.

```
delay_by <- function(delay, f) {
  function(...) {
    Sys.sleep(delay)
    f(...)
  }
}
```

```
new_runif <- dot_every(10, delay_by(0.05, runif))
x <- lapply(1:100, new_runif)
## ..........
```

4.1.1 Memoisation

Um die Ausführung einer häufig genutzten Funktion zu beschleunigen, speichern wir alle berechneten Input-Output-Paare in eine Liste. Wird die Funktion mit einem bestimmten Input ein zweites Mal aufgerufen, wird ohne weitere Berechnung der passende Output-Wert nachgeschlagen und ausgegeben.

`memoise::memoise()` ist ein Funktionsoperator, der dieses Verhalten automatisch einer Funktion beibringt.

```
library(memoise)

fib <- function(n) {
  if (n < 2) return(1)
  fib(n - 2) + fib(n - 1)
}
system.time(fib(25))
##    user  system elapsed
##    0.10    0.00    0.09
system.time(fib(26))
##    user  system elapsed
##    0.16    0.00    0.16
fib <- memoise(fib)
system.time(fib(25))
##    user  system elapsed
##    0.03    0.00    0.03
system.time(fib(26))
##    user  system elapsed
##       0       0       0
```

4.2 Ausgabe-Funktionsoperatoren

Anstatt einer Ausgabe auf der Konsole möchten wir den Text als String-Objekt (`character`-Vektor) zurückgeben. Dies ermöglicht die Funktion `capture.output()`.

```
capture_it <- function(f) { # Funktionsoperator
  force(f)
  function(...) {
    capture.output(f(...))
  }
}
str_out <- capture_it(str)
str(1:10)
##  int [1:10] 1 2 3 4 5 6 7 8 9 10
str_out(1:10)
## [1] " int [1:10] 1 2 3 4 5 6 7 8 9 10"
```

Um die CPU-Zeit der Funktion auszugeben, benutze `system.time()`.

```
time_it <- function(f) { # Funktionsoperator
  force(f)
  function(...) {
    system.time(f(...))
  }
}
compute_sum <- list(
  vec = function(x) sum(x),
  loop = function(x) {
    s <- 0
```

```
    for (a in x) s <- s+a
    s
  }
)
x <- runif(1e7)
call_fun <- function(f, ...) f(...)
lapply(compute_sum, time_it(call_fun), x)
## $vec
##    user  system elapsed
##       0       0       0
##
## $loop
##    user  system elapsed
##    0.19    0.00    0.19
```

4.3 Eingabe-Funktionsoperatoren

`pryr::partial()` nimmt eine Funktion mit mehreren Parametern entgegen und gibt die gleiche Funktion zurück, sodass bestimmte Parameter auf einen konstanten Wert gesetzt sind. `f <- partial(g, b = 1)` entspricht `f <- function(a) g(a, b = 1)`.

```
library(pryr)
rowapply <- partial(apply, MARGIN=1)
A <- matrix(1:9, nrow=3)
rowapply(A, sum)
## [1] 12 15 18
rowSums(A)
## [1] 12 15 18
```

Eine nicht-vektorisierte Funktion kann mittels `Vectorize()` vektorisiert werden. Dies führt nicht zu einer verbesserten Performance, sondern dient der Vereinfachung des Codes.

```
# vektorisiere das Attribut "size" der Funktion sample(),
# sodass eine Liste zurückgegeben wird (SIMPLYFY = FALSE)
sample2 <- Vectorize(sample, "size", SIMPLIFY = FALSE)
str(sample2(1:5, c(1, 1, 3)))
## List of 3
##  $ : int 2
##  $ : int 5
##  $ : int [1:3] 5 4 1
str(sample2(1:5, 5:3))
## List of 3
##  $ : int [1:5] 1 2 4 5 3
##  $ : int [1:4] 2 1 4 3
##  $ : int [1:3] 2 5 1
```

Um einer Funktion ihre Argumente als Liste zu übergeben, nutze `do.call()`. Mit dem entsprechenden Funktionsoperator wandelt man Funktionen so um, dass sie nur noch ein Argument nehmen: Die Liste der ursprünglichen Argumente.

```
args_as_list <- function(f) {
  force(f)
  function(args) {
    do.call(f, args)
  }
}
```

```r
x <- c(NA, runif(100), 1000)
args <- list(
  list(x),
  list(x, na.rm = TRUE),
  list(x, na.rm = TRUE, trim = 0.1)
)
lapply(args, args_as_list(mean))
## [[1]]
## [1] NA
##
## [[2]]
## [1] 10.37333
##
## [[3]]
## [1] 0.4783179
```