Übung 5

Aufgaben 25 bis 31

[redacted]

07.11.2022

Aufgabe 25:

a.

```
hilbert_matrix <- function(n){
 outer(1:n , 1:n, function(x, y) 1/(x+y-1))

}
#example
hilbert_matrix(5)
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 1.0000000 0.5000000 0.3333333 0.2500000 0.2000000
## [2,] 0.5000000 0.3333333 0.2500000 0.2000000 0.1666667
## [3,] 0.3333333 0.2500000 0.2000000 0.1666667 0.1428571
## [4,] 0.2500000 0.2000000 0.1666667 0.1428571 0.1250000
## [5,] 0.2000000 0.1666667 0.1428571 0.1250000 0.1111111
```

b.

`Microbenchmark` ist eine Funktion aus gleichnamigem R-Parket, die als Ersatz zu *system.time* benutzt werden kann. Diese Funktion soll ganz genau die Zeit messen, die benötigt wird einen Ausdruck auszuwerten. In unserem Fall sind auszuwertende Ausdrücke `solve(hilbertmatrix)` und `chol2inv(chol(hilbertmatrix)`. Wenn man der Funktion keien Parameter *times* übergibt, wird jeder Ausdruck 100 mal ausgewertet und von der benötigten Zeit für jede Auswertung wird der Mittelwert berechnet. Reine Ausführung der *microbanchmark*-Funktion liefert eine tabellarische Drastellung von min, median, max… in Nanosekunden, die zur Auswertung benötigt wurden. Man kann die Ergebnisse auch in einem DataFrame speichern. Dieser Data Frame würde dann aus einer Variable *expr* (der auszuwertende Ausdruck) und *time* (benötigte Zeit für Auswertung in Nanosekunden) bestehen. Dataframe würde dann 200 Zeilen beinhalten, da jeder Ausduck 100 mal ausgewertet werden muss.

```
#install.packages("microbenchmark")

library(microbenchmark)
hilbert1 <- hilbert_matrix(3)
hilbert2 <- hilbert_matrix(10)

res1 <- microbenchmark(solve(hilbert1),
                       chol2inv(chol(hilbert1)),
```

```
                        setup = set.seed(120))

res1
```

```
## Unit: microseconds
##                       expr  min   lq   mean median    uq   max neval
##            solve(hilbert1) 14.3 14.8 24.260   16.3 21.10 594.8   100
##   chol2inv(chol(hilbert1))  9.1  9.6 20.667   10.2 11.15 899.0   100
```

Wenn n = 3, ist die durchschnittliche Auswertungszeit von `solve(hilbert1)` höher als die von `chol2inv(chol(hilbert1))`. D.h. bei Berechnung der Inversen ist es deutlich schneller Cholesky -Faktorisierung zu verwenden, als *slove*-Funktion.

Das gleiche Spiel für 10x10 Hilbert-Matrix:

```
res2 <- microbenchmark(solve(hilbert2),
                        chol2inv(chol(hilbert2)),
                        setup = set.seed(120))

res2
```

```
## Unit: microseconds
##                       expr  min   lq   mean median   uq  max neval
##            solve(hilbert2) 16.7 17.4 19.229   17.8 18.5 88.0   100
##   chol2inv(chol(hilbert2))  9.8 10.3 10.911   10.5 10.9 25.2   100
```

Aufgabe 26:

a.

Man berechne $Y = 1.5 \cdot X$*:*

```
Y <- matrix(c(1, 2, 3, 1, 4, 9), ncol = 2)
X <- Y / drop(1.5)
X
```

```
##           [,1]      [,2]
## [1,] 0.6666667 0.6666667
## [2,] 1.3333333 2.6666667
## [3,] 2.0000000 6.0000000
```

Man berechne $Y^T Y$ *and* YY^T*:*

```
t(Y)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    1    4    9
```

```
Y
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    4
## [3,]    3    9
```

```
first_transposed <- crossprod(Y)
second_transposed <- tcrossprod(Y)
microbenchmark(crossprod(Y),
               tcrossprod(Y),
               times = 100L,
               setup = set.seed(120))
```

```
## Unit: nanoseconds
##           expr min  lq mean median  uq  max neval
##   crossprod(Y) 500 600  679    600 600 6800   100
##  tcrossprod(Y) 500 600  650    600 700 2200   100
```

Hier werden Funktionen `crossprod` und `tcrossprod` vewendet. Wenn man *crossprod*-Funktion verwendet, wird transponierte Matrix * zweite Matrix berechnet. *tcrossprod* wird verwendet, wenn man ein Produkt zweier Matrizen berechnen will, wovon die zweite der Funktion übergebene Matrix transponiert sein soll. Es ist in diesem Fall schneller $Y^T Y$ als YY^T zu berechnen, da `tcrossprod` Aufruf von `crossprod(t(x))` evaluert ohne `t(x)` in Wirklichkeit bestimmen zu müssen. **Diese Information habe ich von:** Douglas Bates and Dirk Eddelbuettel, "Fast and Elegant Numerical Linear Algebra Using the RcppEiden Package, p.6 (supplied with RcppEigen version 0.3.1.2)).

b.

```r
#install.packages("expm")
library(expm)
```

```
## Loading required package: Matrix

##
## Attaching package: 'expm'

## The following object is masked from 'package:Matrix':
##
##     expm
```

```r
A <- matrix(rep(1, 1000000), nrow = 1000)
v <- rep(1, 1000)

mat1 <- A%^%2 %*%v

mat2 <- (A%*%A)%*%v

mat3 <- A%*%(A%*%v)

microbenchmark( A%^%2 %*%v,(A%*%A)%*%v, A%*%(A%*%v), times = 10L,
                setup = set.seed(120))
```

```
## Unit: milliseconds
##             expr      min       lq      mean   median       uq      max neval
##    A %^% 2 %*% v 662.4605 681.8500 727.97672 745.3165 757.0305 797.4238    10
##  (A %*% A) %*% v 669.8313 677.0925 723.43426 718.1531 750.6084 802.2966    10
##  A %*% (A %*% v)   2.7203   2.7948   4.11303   3.6656   5.5461   6.8632    10
```

```r
#microbenchmark(mat1, mat2, mat3, times = 10L,
 #              setup = set.seed(120))
```

Der dritte Ausdruck hat die kürzeste Laufzeit, da in den ersten beiden Ausdrücken zuerst Quadrat der Matrix berechnet wird und erst dann mit dem Vektor multipliziert. Beim dritten Ausdruck wird zuerst Produkt von Vektor und Matrix ausgerechnet und dann nochmal mit der Matrix A multipliziert. Es nimmt mehr Zeit in Anspruch zuerst zwei ganz große Matrizen untereinander und dann noch mit einem Vektor zu multiplizieren, als zuerst das Produkt eines Vektors und einer Matrix auszurechnen. Es wird dann für jede Matrixzeile ein Wert ausgerechnet, da ein inneres Produkt des Vektors und der entsprechenden Matrixzeile berchnet werden muss. Dann bleibt noch übrig, diesen neu erzeugten Vektor mit einer Matrix zu multiplizieren, was als Ergebnis wiederum einen vektor zurückgibt. Es ist sogar zeitaufwändiger nur AA bzw. A^2 auszurechnen, als den gesamten Ausdruck $A(Av)$.

c.

```r
#B ist identitätsmatrix, d.h. Diagonale besteht nur aus Einsen
B <- diag(1000)

microbenchmark(A%*%v, A%*%B%*%v, (A%*%B)%*%v,
               A%*%(B%*%v), times = 10L,
               setup = set.seed(120))
```

```
## Unit: milliseconds
##             expr      min       lq      mean    median       uq      max neval
##          A %*% v   1.2571   1.5075   1.68829   1.63675   1.7633   2.3002    10
##    A %*% B %*% v 623.1065 686.9406 725.46666 724.16645 760.4796 825.2478    10
##  (A %*% B) %*% v 671.3593 686.6487 717.79964 711.30870 752.8656 780.2054    10
##  A %*% (B %*% v)   2.9374   3.9024   4.65391   4.41625   5.6723   6.0746    10
```

Wie wir anhand obiger Beobachtugen erwartet haben können, wird Ausdruck Av am schnellsten ausgewertet, da es lediglich um Berechnung des inneren Produkts jeweiliger Matrixzeile und eines Vektors geht. `A%*%(B%*%v)` wird am schnellsten ausgefürhrt, da wie oben angedeutet, es viel effizienter ist eine Matrix mit einem Vektor zu multiplizieren, als erstmal zwei Matrizen untereinander zu multiplizieren und dann noch mit einem Vektor, da Matrizenmultiplikation mehr Rechenaufwand kostet, als zuerst Matrix-Vektor Produkt zu bestimmen, was ebenfalls ein Vektor ist und diesen neuen Vektor dann nochmal mit einer Matrix zu multiplizieren.

Aufgabe 27:

a. *hat-Matrix von Hiblert-Matrix mit n=6:*

```
X <- hilbert_matrix(6)
H <- X%*% solve(t(X) %*% X) %*% t(X)
#H
```

b. *Eigenvektoren und Eigenwerte*

```
my_eigen <- eigen(H, symmetric = TRUE)
#Hilbert Matrix are symmetric and positive definite

#str(my_eigen)

eg_values <- my_eigen$values
eg_values
```

```
## [1] 1.0004314 0.9999964 0.9999936 0.9999677 0.9995693 0.9984126
```

`str(my_eigen)` gibt eine Liste bestehend aus 2 Elementen zurück. Das erste Listenelement ist ein Vektor bestehend aus Eigenwerten und das zweite Vektor bestehend aus Eigenvektoren. Auf Eigenwerte kann man ganz einfach mittels Operator $ zugreifen. `my_eigen$values` gibt uns alle Eigenwerte zurück.

c.

```
trace <- function(x) sum(diag(x))

trace(H)
```

```
## [1] 5.998371
```

```
sum(eg_values)
```

```
## [1] 5.998371
```

Summen aller Eigenwerte und Elemente an der Diagonale von H sind gleich. Das ist übrigens ein Satz, der leicht bewiesen werden kann.

d.

```
det(H)
```

```
## [1] 0.998371
```

```
prod(eg_values)
```

```
## [1] 0.9983709
```

Hier sieht man das geliche Verhalten wie oben. Zwei ausgrechnete Werte sind gleich, aber das Ergebnis der Funktion *prod* wurde nicht auf 6 sondern auf 7 Nachkommastellen gerundet.

Satz: *Produkt der Eigenwerte ist gleich der Determinante der Matrix.*

e.

```
x_inverse <- solve(X)
xinv_eigen <- eigen(x_inverse)
xinv_eigenvalues <- xinv_eigen$values
xinv_eigenvectors <- xinv_eigen$vectors

xinv_eigenvalues
```

```
## [1] 9.235320e+06 7.954970e+04 1.624040e+03 6.126880e+01 4.126079e+00
## [6] 6.177034e-01
```

Es besteht eine ganz wichtige Beziehung zwischen Eigenwerten der Inversen und der Originalmatrix und **zwar sind Eigenwerte der Inversen gleich dem Kehrwert von Eigenwerten der Originalmatrix**. Das bestätigt der folgende Code:

```
x_eigen <- eigen(X)
x_val <- x_eigen$values

to_check <- 1/x_val
#to_check

#ich werde die zwei Vektoren sortieren, damit man
#gleich sieht dass die gleich sind

sort(xinv_eigenvalues)
```

```
## [1] 6.177034e-01 4.126079e+00 6.126880e+01 1.624040e+03 7.954970e+04
## [6] 9.235320e+06
```

```
sort(to_check)
```

```
## [1] 6.177034e-01 4.126079e+00 6.126880e+01 1.624040e+03 7.954970e+04
## [6] 9.235320e+06
```

Aufgabe 28:

a.

```
hilb20 <-hilbert_matrix(20)
#Aufruf von chol(hilb20) muss auskommentiertwerden, sonst
#kann keine PDF-Datei erstellt werden, da diese Funktion bricht
#chol(hilb20)
```

Funktion `chol` kann nur auf positiv definite Matrizen angewendet werden. In unserem Fall ist eine Hilbert-Matrix gegeben, deren Einträge durch $(i, j) = 1/(i + j - 1)$ bestimmt werden. Wenn n=20 gegeben ist, heißt es, dass die jewieilige Einträge in der Matrix immer kleiner und kleiner werden. An dieser Stelle ist es wichtig anzumerken, dass eine **symmetrische** Matrix (wie Hilbert-Matrix) positiv definit ist, wenn alle ihre Eigenwerte positiv sind. Wenn man sich entsprechende Eigenwerte der Hilbert-Matrix `hilb20` näher anschaut, merkt man leicht, dass einige von denen kleiner als `.Machine$double.eps` sind. Das ist der Grund, warum R diese Zahlen dann als negativ betrachtet und keine Cholesky Faktorisierung bestimmen kann (Hilbert-Matrix `hilb20` **kann nicht als positiv definit gesehen werden**).

```
hilb20eig <- eigen(hilb20)
hilb20eigvalues <- hilb20eig$values

hilb20eigvalues <= .Machine$double.eps
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

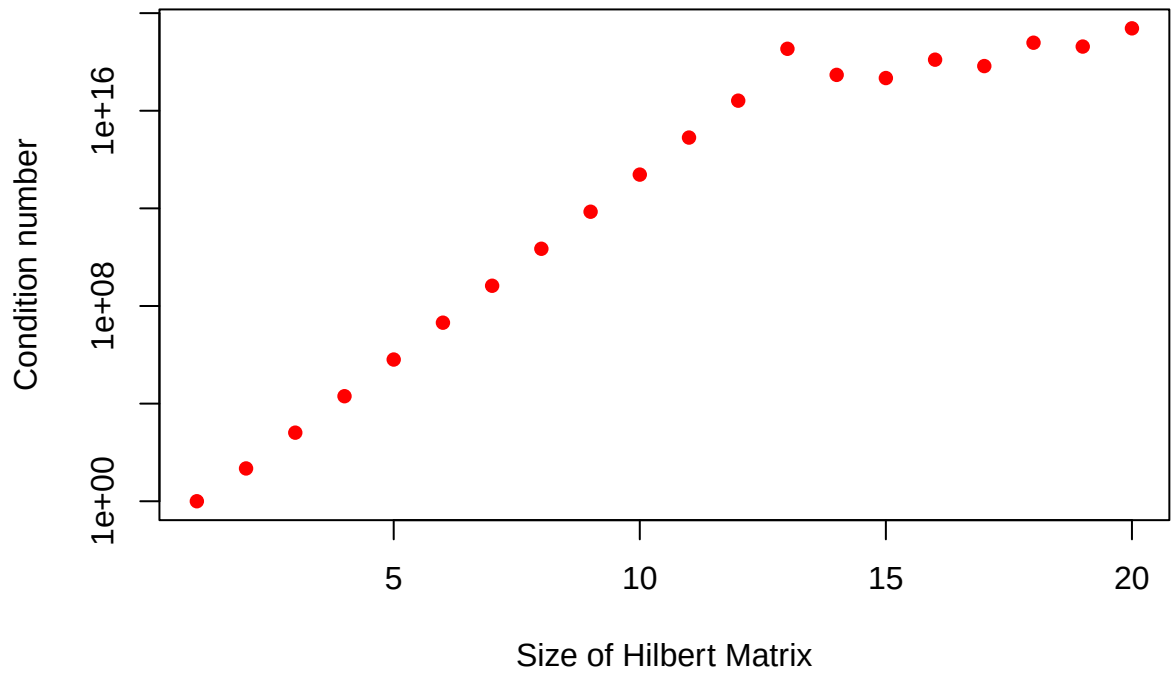
Dieser "Fehler" kommt bei allen Hilbert-Matrizen mit n >12 vor und es kann keine Cholesky Faktorisierung in R durchgeführt werden.

b.

Die **Kondition** ist ein Begriff der numerischen Mathematik, welcher die Abhängigkeit der Lösung eines problems von Änderungen in den Eingabedaten beschreibt . Dementsprechend ist die **Konditionszahl** ein Maß für diese Abhängigkeit. Gibt Funktion `kappa()` einen extrem großen Wert zurück, weist dies auf schlechte numerische Eigenschaften hin.

Im Folgenden, werde ich einen Vektor konstruieren, in dem ich Konditionszahlen für Hilbert-Matrizen verschiedener Größe abspeichern werde und diese werde ich dann in Abhängigkeit von der Matrixgöße plotten. Es würde keinen Sinn machen einen Verhältnisplot mit nur 4 verschiedenen Hilbert-Matrizen zu machen, deswegen werde ich zu meinem Plot alle Hilbert-Matrizen mit Größen von 1×1 bis 20×20 hinzufügen.

```
n <- 20
conditionHilbert <- rep(NA, n)

for(i in 1:n){
  conditionHilbert[i] <- kappa(hilbert_matrix(i))
}

plot(1:n, conditionHilbert, pch = 16, col = "red",
     xlab ="Size of Hilbert Matrix",
     ylab = "Condition number",
     log ="y")
```



In meinem Plot habe ich y-Achse logaritmisch skaliert, sprich es werden Logarithmen von den Konditionszahlen im Plot angezeigt. Es lässt sich anhand des Plots feststellen, das **logaritmisierte Konditionszahlen linear wachsen**, bzw. dass **Konditionszahlen (wären sie nicht logaritmisiert gewesen) exponentiell wachsen**. Das heißt je größer eine Hilbert-Matrix wird, desto größer ihre Konditionszahl. Es ist aber jedenfalls eine Merkwürdigkeit anzumerken: Nach n=13 werden Konditionszahlen fast konstant und die schweben immer in Nähe voneinander.

Wenn man sich nur Konditionzahlen für Hilbert-Matrizen der Größen $3 \times 3, 5 \times 5, 7 \times 7, 20 \times 20$ ansehen will, kann man diese ganz einfach abrufen:

```
interessting_values <- c(conditionHilbert[3], conditionHilbert[5],
                         conditionHilbert[7], conditionHilbert[20])
interessting_values
```

```
## [1] 6.462247e+02 6.403564e+05 6.742592e+08 2.378519e+19
```

Es handelt sich hier um ganz große Zahlen, die exponeziell wachsen. Konditionszahl erreicht ihren höchsten Wert bei einer 20×20 Hilbert-Matrix. Dies bedeutet folgendes: Wenn man bei kleiner Hilbert-Matrizen n ändert, wird dass bei folgender Hilbert-Matrix eine ganz große Abhängigkeit von dem übergebenen Parameter n aufweisen. Jedoch nach n=13 wird das nicht so sein. Kondizionszahlen von n=15 und n=17 werden sich nicht viel unterscheiden, wie zum Beispiel bei n=5 und n=7, wo sich Konditionzahlen um Exponent 10^3 unterscheiden.

```
conditionHilbert[15]
```

```
## [1] 2.186473e+17
```

```r
conditionHilbert[17]
```

```
## [1] 6.768158e+17
```

Aufgabe 29:

a.

```r
LmNaive <- function(y, X){
  return(solve(t(X)%*%X)%*% t(X) %*%y)
}
```

b.

```r
LmCP <- function(y, X){
  return (solve(crossprod(X), t(X)%*%y))
}
```

c.

$\hat{\beta} = (X^T X)^{-1} X^T y$ kann auch geschrieben werden als $\hat{\beta} = \dfrac{X^T y}{X^T X} \leftrightarrow X^T X \hat{\beta} = X^T y$

Eine Matrix A kann als $X^T X = LL^T$ faktorisiert werden, mit $L \in \mathbb{R}^{p \times p}$ untere Dreiecksmatrix: $l_{ij} = 0$ wenn$i < j$. Dies heißt Cholensky Faktorisierung von A und L ist invertierbar.

1. In z lösen wir $Lz = X^T y$ durch forward Substitution auf(`forwardsolve`), wo $z = L^T \hat{\beta}$.
2. Dann können wir $\hat{\beta}$ in $L^T \hat{\beta}$ brechnen, indem wir *backward* Substitution anwenden (`backsolve`).

Folgende Schritte werden durchgeführt:

1. Cholensky Faktorisierung von $X^T X = LL^T$: Wir verwenden `chol` Funktion, die uns eine Matrix L^T zurückgibt. Die Speichern wir in U.
2. Dann durch `forwardsolve` Ausdruck $U^T z = X^T y$ auflösen. Speichern wir in Z.
3. Am ende bleibt nur noch übrig durch `backwardsolve` den Ausdruck $U\hat{\beta} = z$ auszuwerten und wir haben unser $\hat{\beta}$.

```r
LmChol <- function(y, X){
  U <- chol(crossprod(X))
  Z <- forwardsolve(U, crossprod(X, y))
  beta_hat <- backsolve(U, Z)
  return(beta_hat)
}
```

d.

```r
LmSvd <- function(y, X){
  svd_output <- svd(X)
  U <- svd_output[["u"]]
  V <- svd_output[["v"]]
  sigma <- svd_output[["d"]]
  beta <- V %*% diag(1 / sigma) %*% t(U) %*% y
  return(beta)
}
```

e.

$\hat{\beta} = R^{-1}Q^T y$

```
LmQR <- function(y, X){
  QR <- qr(X)
  Q <- qr.Q(QR)
  R <- qr.R(QR)
  beta_hat <- backsolve(R, crossprod(Q, y))
  return(beta_hat)

}
```

f.

```
?lm.fit()
```

```
## starting httpd help server ... done
```

`lm.fit()` steht für *Fitter Functions for Linear Models*. Diese Funktion wird von der Funktion `lm` aufgerufen und sie bestimmt ob es sich bei den übergebenen Parameter um passende Parameter für Regression handelt.

g.

```
set.seed(1)
n <- 50
x <- rt(n, 2)
eps <- rnorm(n,0, 0.1)
y <- 3 - 2 * x + eps
X <- cbind(1, x)
colnames(X) <- c("Intercept","x")
```

```
#LmNaive
lmNaive_OLS <- LmNaive(y, X)
#LmCP
lmCP_OLS <- LmCP(y, X)
#LmChol
lmChol_OLS <- LmChol(y, X)
#LmSVD
lmSVD_OLS <- LmSvd(y, X)
#LmQR
lmQR_OLS <- LmQR(y, X)

default_qr_OLS <- qr.solve(X, y)
my_fit <- lm.fit(X, y)

microbenchmark(lmNaive_OLS, lmCP_OLS, lmChol_OLS,
               lmSVD_OLS, lmQR_OLS, default_qr_OLS,
               my_fit,
               times = 10L)
```

```
## Warning in microbenchmark(lmNaive_OLS, lmCP_OLS, lmChol_OLS, lmSVD_OLS, : Could
## not measure a positive execution time for 10 evaluations.
```

```
## Unit: nanoseconds
##            expr min lq mean median uq  max neval
##     lmNaive_OLS   0  0   70      0  0  700    10
##        lmCP_OLS   0  0  500      0  0 5000    10
##      lmChol_OLS   0  0  220      0  0 2200    10
##       lmSVD_OLS   0  0   60      0  0  600    10
##        lmQR_OLS   0  0   80      0  0  800    10
##  default_qr_OLS   0  0   90      0  0  900    10
##          my_fit   0  0  140      0  0 1400    10
```

Alle Implemetierunge bis auf die Naive, liefern gleiche Schätzungen von $\hat{\beta}$ mit kleinen Abweichungen die durch Rundungsfehler oder numerische Fehler entstanden sind. Nur die Naive Implementierung liefert bemerkenswert unterschiedliche Resultate und braucht am längsten um ausgewertet zu werden. Das war aber zu erwarten, da wir bereits wissen, dass es empfholen ist, wo möglich Matrizen statt mit `%*%` mit Hilfe von `crossprod` bzw. `tcrossprod` zu multiplizieren und der Funktion `solve` zwei Parameter zu übergeben.

Aufgabe 30:

a.

X ist als eine sparse-Matrix definiert und Elemente ungleich Null befinden sich nur an der Diagonale. Wenn wir diese Eigenschaft von X nicht in Betracht ziehen, aber eine neue Matrix $y = X\beta$ berechnen wollen, dann wird die Matrix X zeilenweise durchgemustert. Man muss sich n Komponenten von y ausrechnen, die eigentlich inneres Produkt der jeweiligen Zeile in Matrix und des Vektors sind. Es gibt insgesamt n Zeilen in der Matrix, was im Endeffekt mit $2 \cdot n \cdot n = 2n^2$ Flops resultiert. (2, da wir inneres Produkt von jeder Matrixzeile und dem Vektor berechnen und das wird so in R implementiert, dass wir bei jeder Iteration für diese bestimmte Zeile einen `counter` $g_i \leftarrow g_i + x_{ij}\beta_j$, ausrechnen, was zwei Flops sind).

Da wir mit einer sparse-Matrix arbeiten, können wir uns das ersparen, da Elemente ungleich Null nur an der Diagonale liegen und die gibt es insgesamt n. Dann brauchen wir nur noch $2n$ Flops, um $X\beta$ auszurechnen.

Pseudocode für eine sparse-Matrix, die mit einem Vektor multipliziert wird:

$g \leftarrow 0 \wedge g \in \mathbb{R}^n$

$for\ \ i = 1, n\ \ do$

$for\ \ j = 1, n\ \ do$

$\ \ if(i == j)$

$\ \ \ \ g_i \leftarrow g_i + x_{ij}\beta_j$

$end\ \ for$

$end\ \ for$

$return\ \ g$

b.

```
make_sparse_c1 <- function(M){
  indices <- cbind(which(M!= 0, arr.ind = TRUE),
                   M[M!=0])
  colnames(indices)<- c("i", "j", "x")
  return (indices)
}

set.seed(1234)
n <- 30
M <- matrix(0, nrow = n, ncol = n)
M[sample(1:(n^2), floor(n^2* 0.1))] <- rnorm(floor(n^2 * 0.1))

indices_of_sparse <- make_sparse_c1(M)
head(indices_of_sparse)
```

```
##       i j           x
## [1,]  4 1  1.60590963
## [2,] 10 1 -0.50125806
## [3,] 12 1 -1.06864272
## [4,] 22 1 -0.36652393
## [5,] 30 1 -0.70944004
## [6,]  9 2 -0.03476039
```

c.

```r
library(Matrix)
str(as(M, "sparseMatrix"))
```

```
## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
##   ..@ i       : int [1:90] 3 9 11 21 29 8 12 27 28 2 ...
##   ..@ p       : int [1:31] 0 5 9 14 15 16 18 22 25 27 ...
##   ..@ Dim     : int [1:2] 30 30
##   ..@ Dimnames:List of 2
##   .. ..$ : NULL
##   .. ..$ : NULL
##   ..@ x       : num [1:90] 1.606 -0.501 -1.069 -0.367 -0.709 ...
##   ..@ factors : list()
```

`@i` - speichert Zeilen in denen eine Zahl ungleich Null gespeichert ist. Indexierung beginnt hier aber bei 0 und nicht bei 1, wie es in R üblich ist.

`@p` - kumulative Anzahl der Datenpunkte ungleich Null, wenn man Dataset spaltenweise durchmustert (von links nach rechts). Erster Eintrag ist immer 0. Wird berechnet durch:`c(0, cumsum(colSums(M!= 0)))`

`@Dim` - Dimension der Marix: 30x30

`@x` - non zero Werte, aber sortiert (von oben nach unten, von links nach rechts).

d.

```r
sparse_add <- function(a, b){
  newsp <- merge(a, b, all = TRUE)
  newsp <- aggregate(x~i+j, data = newsp, sum)
  return (newsp)
}

a<- matrix(0, nrow = n, ncol = n)
a[sample(1:(n^2), floor(n^2* 0.1))] <- rnorm(floor(n^2 * 0.1))
a<- make_sparse_c1(a)
sumup <- sparse_add(a,indices_of_sparse)
```

e.

```r
#habe ich leider nicht geschafft
compute_lambda1 <- function (y, X){
  #z<- matrix()
  for(j in ncol(X)){
    z[,j] <- (X[, j]- mean(X[, j]))/sd(X[, j])
  }

}
```

Aufgabe 31:

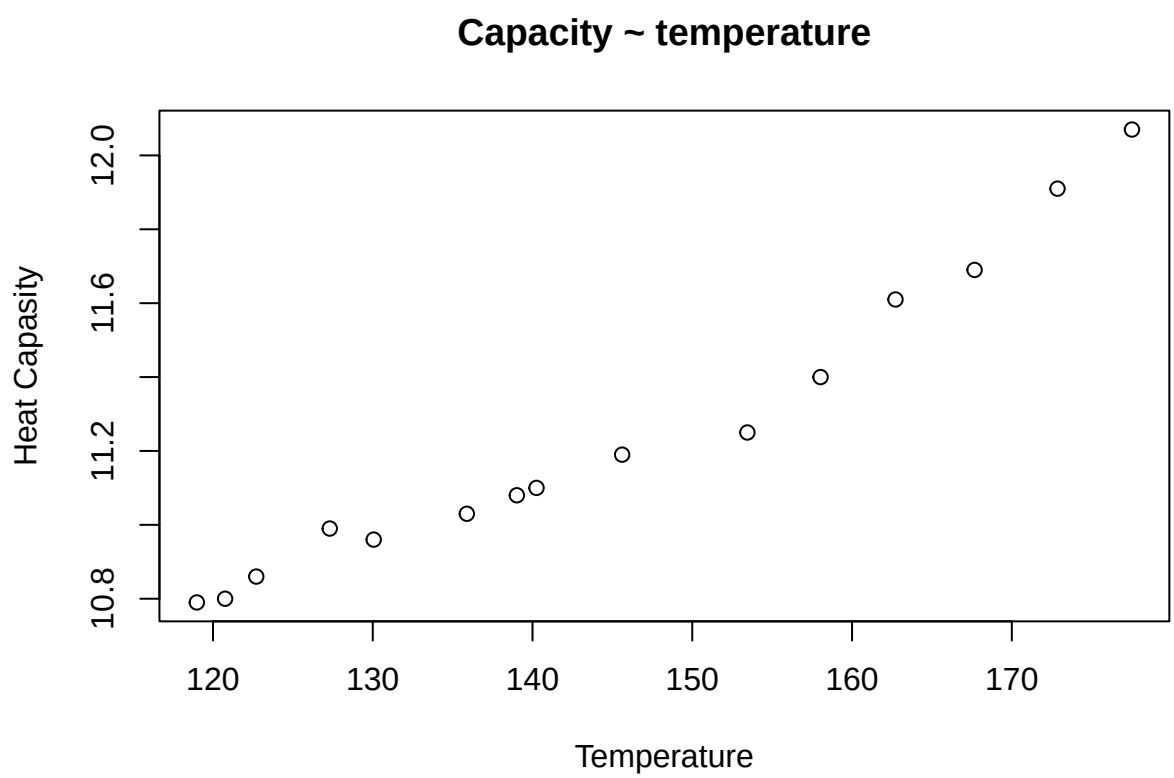
a.

```
data("heatcap", package = "GLMsData")
set.seed(1)
sample <- sample(c(TRUE, FALSE), nrow(heatcap),
                 replace = TRUE, prob = c(0.9, 0.1))
train <- heatcap[sample, ]
test <- heatcap[!sample, ]
```

b.

Zuerst können wir den TrainSet plotten, damit wir ein Gefühl dafür bekommen, wie der Zusammmenhang zwischen `Cp` und `Temp` aussieht.

```
plot(train$Cp ~ train$Temp,
     xlab ="Temperature",
     ylab = "Heat Capasity",
     main ="Capacity ~ temperature")
```




```
degree <- 5

  fit.train = lm(Cp ~
                   Temp + I(Temp^2) + I(Temp^3) + I(Temp^4) + I(Temp^5),
```

```r
                   data = train)

summary(fit.train)
```

```
##
## Call:
## lm(formula = Cp ~ Temp + I(Temp^2) + I(Temp^3) + I(Temp^4) +
##     I(Temp^5), data = train)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.045752 -0.020238 -0.000042  0.010783  0.059248
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  7.104e+01  6.099e+02   0.116    0.910
## Temp        -3.172e+00  2.097e+01  -0.151    0.883
## I(Temp^2)    5.781e-02  2.870e-01   0.201    0.845
## I(Temp^3)   -4.872e-04  1.956e-03  -0.249    0.809
## I(Temp^4)    1.951e-06  6.634e-06   0.294    0.775
## I(Temp^5)   -3.004e-09  8.963e-09  -0.335    0.745
##
## Residual standard error: 0.03545 on 9 degrees of freedom
## Multiple R-squared:  0.995,  Adjusted R-squared:  0.9923
## F-statistic: 360.2 on 5 and 9 DF,  p-value: 4.433e-10
```

c.

```r
fit.train1 <- lm(Cp ~ poly(Temp, 5), data = train)

summary(fit.train1)
```

```
##
## Call:
## lm(formula = Cp ~ poly(Temp, 5), data = train)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.045752 -0.020238 -0.000042  0.010783  0.059248
##
## Coefficients:
##                 Estimate Std. Error  t value Pr(>|t|)
## (Intercept)    11.248667   0.009154 1228.867  < 2e-16 ***
## poly(Temp, 5)1  1.471645   0.035452   41.511 1.36e-11 ***
## poly(Temp, 5)2  0.284146   0.035452    8.015 2.18e-05 ***
## poly(Temp, 5)3  0.111381   0.035452    3.142   0.0119 *
## poly(Temp, 5)4 -0.068087   0.035452   -1.921   0.0870 .
## poly(Temp, 5)5 -0.011883   0.035452   -0.335   0.7452
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.03545 on 9 degrees of freedom
```

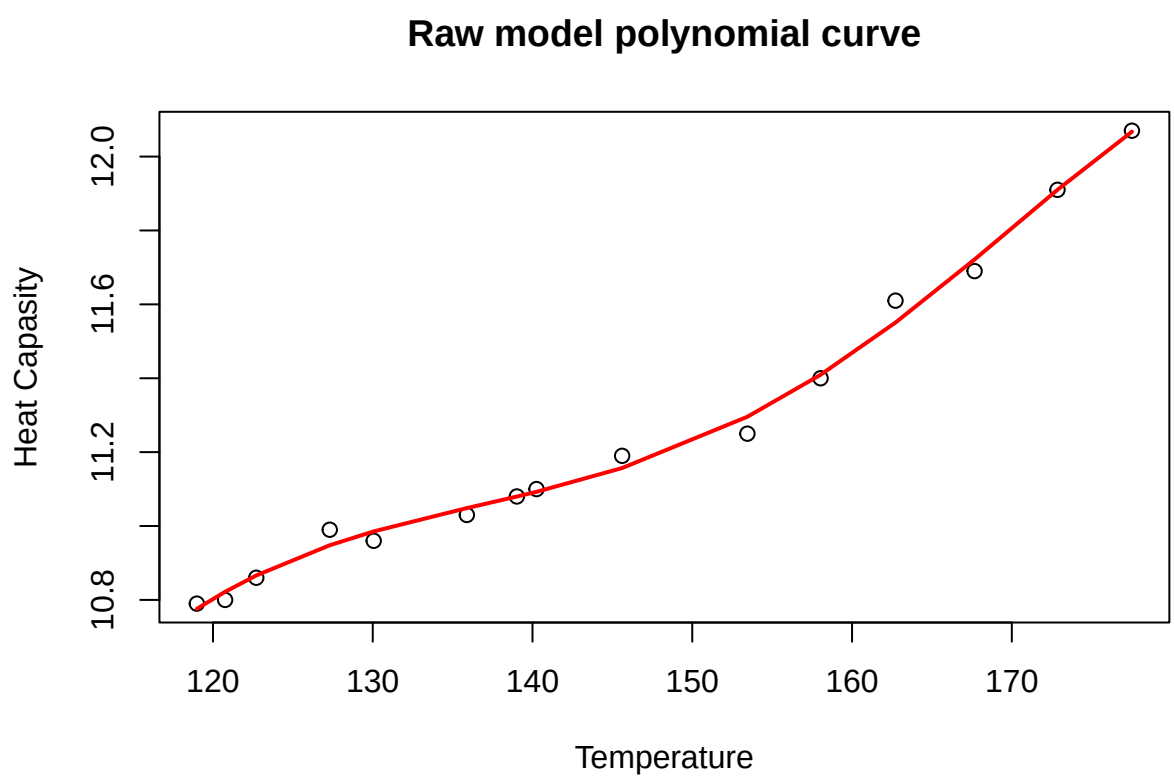
```
## Multiple R-squared:  0.995,  Adjusted R-squared:  0.9923
## F-statistic: 360.2 on 5 and 9 DF,  p-value: 4.433e-10
```

Wenn man sich `summary` von `fit.train` und `fitr.train1` ansieht, merkt man leicht, dass die zwei linearen Modelle verschiedene Intercepts und Koeffizienten ausgerechnet haben. Das liegt daran, dass `poly` Funktion orthonormale Polynome erzeugt und die Koeffiziente von beispielsweise `I(Temp^2)` nicht die Gleichen wie von `poly(Temp, 2.` sind Falls wir uns gleiche Koeffizienten bei beiden Modellen wünschen, können wir in der `poly`-FUnktion den Parameter `raw = TRUE` setzen. Dann werden beide Modelle gleiche Koeffizienten haben.

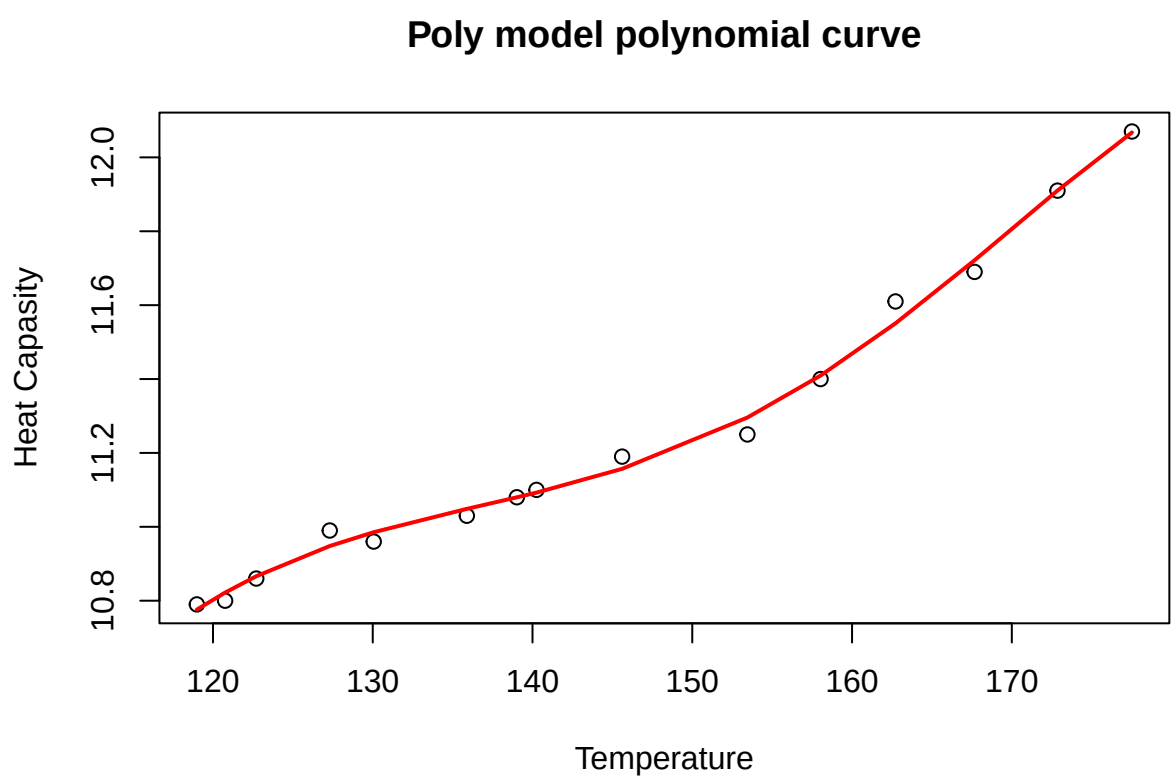
Dabei kann man noch anmerken, dass `poly`-Funktion von einem Vektor `x` (in unserem Fall von der Variablen `Temp`), äquivalent zur QR-Faktorisierung der Matrix ist, deren Spalten Potenzen von `x` sind.

Die F-Statistiken für beide Modelle sind gleich, was darauf hinweist, dass beide Modelle gleich gut für Regression geeignet sind. Wir können das jedoch etwas besser veranschaulichen, indem wir Regressionskurven der beiden Modelle `fit.train` und `fit.train1` plotten:

```r
#regression curve for raw model
plot(train$Cp ~ train$Temp,
     xlab ="Temperature",
     ylab = "Heat Capasity",
     main ="Raw model polynomial curve")

pred <- predict(fit.train)
ix <- sort(train$Temp, index.return=T)$ix
lines(train$Temp[ix], pred[ix], col='red', lwd=2)
```



```r
#regression curve for poly model
plot(train$Cp ~ train$Temp,
     xlab ="Temperature",
     ylab = "Heat Capasity",
     main ="Poly model polynomial curve")

pred <- predict(fit.train1)
ix <- sort(train$Temp, index.return=T)$ix

lines(train$Temp[ix], pred[ix], col='red', lwd=2)
```



d.

```r
fit.test <- predict(fit.train, newdata= test)
rmse <- mean((fit.test - test$Cp)^2)
rmse <- sqrt(rmse)
rmse
```

```
## [1] 0.0803141
```

e.

RMSE für $K = 1, ..., 7$

```r
rmse <- rep(NA, 7)

for(i in 1:7){
  fit.train <- lm(Cp ~ poly(Temp, i), data = train)
  fit.test <- predict(fit.train, newdata = test)
  rmse[i] <- mean((fit.test - test$Cp)^2)
  rmse[i] <- sqrt(rmse[i])
}

rmse
```

```
## [1] 0.19049058 0.06920347 0.02107097 0.06181531 0.08031410 0.04592737 0.03696224
```

K, welches das kleinste $RMSE$ liefert wäre $K = 3$ mit $RMSE = 0.021$.