Einführung zu S-Plus[1]

1 S-Plus allgemein

S-Plus ist ein sehr mächtiges Statistikprogramm. Im Grunde ist es eine Programmierumgebung, in der sich sehr viel realisieren lässt. Die UNIX- und WINDOWS-Versionen unterscheiden sich nur unwesentlich.

- Start von S-Plus: z.B. **thales$ Splus** ↩
- Beenden von S-Plus: q() ↩
- Befehlszeile: >
- Bestätigung einer Eingabe: ↩
- "+" zu Beginn der Befehlszeile: Eingabe kann/muss fortgesetzt werden
- Mehrere Befehle in einer Eingabeziele durch "**;**" trennen
- **Hilfe:** > help(*name*) oder > ?*name*
- **Kommentare!!!** mit # einleiten

Variablen

> objekt1 <− 1.043 # d.h.: GleitPUNKTzahlen, **Zuweisung** mit <−

> objekt1 <− 3 # überschreibt die erste Definition

> Objekt1 <− 2 # nicht = "objekt1" (Groß-/Kleinschreibung beachten!)

Definierte Objekte werden (z.B.) unter /home/thales/user/MySwork/.Data gespeichert. Bereits belegte Variablennamen: z.B. pi, t, f, T, F, mean, var,...

- ls(): listet alle gespeicherten Objekte auf
- rm(objekt1): entfernt objekt1 (remove(ls()): entfernt alle gespeicherten Objekte!)

2 Datengenerierung und -strukturierung

2.1 Vektoren

Vektoren: mehrere Objekte **gleicher Art** zu einem Objekt verschmelzen. Die Objekte werden mit fortlaufender Nummer hintereinander geschrieben und können so auch erreicht werden. Systematisch werden Objekte durch den Befehl **c(element1,element2,...)** zu einem Vektor verbunden - das c = concatenate.

[1] Ohne Anspruch auf Vollständigkeit und Fehlerfreiheit.

Weitere Möglichkeiten:

- > rep(x, times=n) # wiederhole Objekt x times mal, z.B. rep(1,4) = c(1,1,1,1)
- > seq() # Syntax: seq(from=, to=, by=, length=)
  Beispiel: > seq(1,3, by=0.1) ergibt 1.0, 1.1, 1.2,..., 2.9, 3.0
- > from:to # entspricht seq(from,to,by=1.0), z.B. 1:3 ergibt 1,2,3
- > scan() # zeilenweise Einlesen von Std-Eingabe oder aus einer Datei
  Beispiele: > daten <- scan() oder daten <- scan("input.data")

Bestimme Länge des Vektors (Anzahl der Einträge): **length()**

Zugriff auf Einträge von Vektoren:
> x <- c(1.4, 3.7, 2.0, 4.6, 5.1)
> x[2]
[1] 3.7
> x[1:4]
[1] 1.4 3.7 2.0 4.6

2.2 Matrizen

Syntax: matrix(data, nrow=, ncol=, byrow=F) [Dimension von Matrizen: dim()]
Matrizen können nur Variablen eines Typs enthalten (Zahlen, Zeichenketten,...).

Beispiele:

- > m <- matrix(0, 4, 5) # eine 4 x 5 Null-Matrix
- > matrix(1:10, 5) # eine 5 x 2 Matrix
- > mm <- matrix(scan("mfile"), ncol=5, byrow=TRUE)
  # zeilenweise Einlesen aus Datei

siehe auch: rbind(vektor1, vektor2,...) bzw. cbind(vektor1, vektor2,...)

Zugriff auf Einträge von Matrizen:
> x <- c(1.4,3.7,2.0,4.6,5.1); m <- matrix(x,2,2,T)
> m[1,2]
[1] 3.7

i-te Zeile: m[i,] j-te Spalte: m[,j]

2.3 Data Frames

Verallgemeinerung des Typs Matrix: data.frame(Objekt1, Objekt2,...)
Beispiel:

- > x <- data.frame("Gewicht"=c(65,75),"Groesse"=c(168,175),"Geschlecht"=c("m","w"))

```
> print(x)
     Gewicht   Groesse   Geschlecht
1       65        168         m
2       75        175         w
```

Einlesen aus eine Datei: read.table(file="myfile", header=T/F)
header=TRUE, falls Spaltenbezeichnungen in der Datei *myfile* enthalten sind.

Zugriff: x$Gewicht ergibt [1] 65 75

2.4 Listen

Listenelemente können eine beliebige Datenstruktur aufweisen (in bezug auf Typ und Dimensionen).

> list(element1, element2,...)

Aufruf erstes Element:
> x[[1]] # ist dies ein Vektor, so erhält man mit x[[1]][1] dessen erstes Element

> list(Name1=element1, Name2=element2,...)
den Elementen werden Namen zugewiesen

Zugriff über: x$Name (wichtig bei: Zugriff auf Ausgaben von S-Plus Funktionen)

3 Funktionen und Operatoren

S-PLUS liefert sofort nach Aufruf des Programms eine sehr große Menge an vordefinierten Funktionen:

- +, -, *, / (nicht ":" !), ^
- min(), max(), sort()
- mean(), median(), var(), stdev()
- sqrt(), exp(), log(), log10(), logb(x,base=2), cos(), sin(), tan() etc.
- sum(), prod(), diff(), range()

  Beispiel: x <- c(1, 2.3, 4.2, 3.2)
  ```
  > sum(x)      [1] 10.7
  > diff(x)     [1] 1.3  1.9  -1.0
  > range(x)    [1] 1.0  4.2
  > sqrt(x)     [1] 1.0000  1.5166  2.0494  1.7889
  ```

Beachte bei Matrizen:
```
> m
     [,1]  [,2]
[1,]    1     2
[2,]    3     4
```

```
> sqrt(m)
          [,1]     [,2]
 [1,]  1.000000  1.414214
 [2,]  1.732051  2.000000

> sum(m)
[1] 10

> apply(m,1,sum)    #"1": zeilenweise, "2": spaltenweise
[1]  3  7
```
allgemein: apply(Matrix, Margin=1/2, FUN)

Definition einer Funktion:

```
> wurzel <- function(x,n=2){
+         out <- x^(1/n)
+         return(out)
+ }
```

Aufruf: > wurzel(x)

Beispiel zu Listen:

```
> meineFunktion <- function(x,n){
+ d <- x-n
+ m <- x*n
+ result <- list(differenz=d, multipl=m, x=x, n=n)
+ return(result)
+ }

> meineFunktion(2,3)
$differenz:
[1] -1

$multipl:
[1] 6

$x:
[1] 2

$n:
[1] 3
```

4 Konditionalabfragen

Oft benötigt man innerhalb einer Funktion die Möglichkeit, sich je nach Lage der Situation zu entscheiden und fortzufahren. In S-PLUS kann dies u.a. folgendermassen realisiert werden:

if (*test*) { Anweisungen für test==TRUE}
else { Anweisungen für test==FALSE}

Hinter *test* können sich zum Beispiel einfache Abfragen wie "n > 10" verbergen. Die Auswahl kann auch aufgrund meherer Testabfragen entschieden werden:

if (test1 && test2) { Anweisungen für test1 **und** test2==TRUE }
if (test1 || test2) { Anweisungen für test1 **oder** test2==TRUE }

Es gibt noch andere Möglichkeiten, Bedingungen abzufragen.

```
> x <- 1:10; y <- c(1,2,1,2,1,1,1,2,2,2)
> x > 5        [1] F F F F F T T T T T
> x[x>5]       [1] 6 7 8 9 10
> x[y==2]      [1] 2 4 8 9 10
```

5 Schleifen in S-PLUS

- **for (Variable in Werte) {S-PLUS-Ausdrücke}**
  Hierbei wird die Anzahl der Iterationen durch werte genau vorgegeben, welche die variable gerade annimmt. Die Summe der ersten 100 natürlichen Zahlen kann man also folgendermassen als Schleife darstellen:

  ```
  > z <- 0
  > for (i in 1:100){
  +  z <- z+i;  print(z)
  +  }
  ```

- **while (Bedingung) {S-PLUS-Ausdrücke}**
  Solange die Bedingung erfüllt ist, wird die Schleife nicht verlassen. Das folgende Programm addiert Zahlen, bis deren Summe grösser als 1000 ist:

  ```
  > n <- 0; summe <- 0
  > while(summe.bisher <= 1000){
  +  n <- n+1;
  +  summe <- summe+n
  +  }
  > print(summe)
  [1] 1035
  > print(n)
  [1] 45
  ```

6 Verteilungsmodelle

In S-PLUS sind eine ganze Reihe theoretischer Verteilungen implementiert, auf deren Dichte, Verteilungsfunktion etc. zugegriffen werden kann. Die Abfragen sind bei allen Verteilungen folgendermaßen aufgebaut:

d*verteilung*() (Zähl-)Dichte
p*verteilung*() Verteilungsfunktion
q*verteilung*() Quantil
r*verteilung*() Zufallszahl

verteilung: norm, unif, exp, pois, binom, t, f, chisq etc.
exakte Aufrufe siehe help()

7 Graphiken in S-PlUS

S-PLUS besitzt vielseitigen Graphik-Routinen. Sie können nur bei einem geöffneten graphischen Device aktiv werden. Unter Windows leistet dies die Anweisung win.graph(), unter UNIX oft motif().

motif()	Öffnen eines Graphikfensters (nicht immer notwendig!)
dev.off()	Schließen eines Graphikfensters; z.B. dev.off(2)
par():	ermöglicht die (allgemeine) Kontrolle über den Graphikbereich.

Um zum Beispiel eine 2x2-Matrix von Bildern in einem Bereich zu erzeugen, kann der Befehl par(mfrow=c(2,2)) benutzt werden - die Bilder werden zeilenweise erzeugt; vgl. mfcol.

Graphische High-Level-Routinen

plot(x)	erzeugt Scatterplot (isolierte Punkte!); z.B. plot(x,y)
barplot(x)	erzeugt Balkendiagramm
boxplot(x)	erzeugt Boxplot
hist(x)	erzeugt Histogramm
pairs(x)	erzeugt paarweise Scatterplots
qqplot(x)	erzeugt QQ-Plot

Aufruf von motif() ist für diese nicht notwendig, wenn nur ein Fenster geöffnet werden soll.

Daneben können der Funktion eine Vielzahl von Parametern mitgegeben werden, die das endgültige Layout verbessern können. Hier ist eine kleine Auswahl:

type="p"	Daten als Punkte
type="l"	Daten als Linien
type="b"	Daten als Punkte und Linien
xlim=c(1,100)	Grenzen der x-Achse (z.B.: 1 und 100)
ylim=c(0,1)	Grenzen der y-Achse (z.B.: 0 und 1)
xlab="x-Achse"	Beschriftung der x-Achse
ylab="y-Achse"	Beschriftung der y-Achse
log="xy"	x- und y-Achse logarithmisch; auch log="x", log="y"
main="Testplot"	Überschrift zum Bild

Die aufgelisteten Parameter können mit den gewünschten Werten durch Kommata getrennt nach den Daten der Funktion plot() übergeben werden. Diese Parameter lassen sich, wenn sie dort Sinn machen, auch für die anderen aufgeführten High-Level-Plot-Funktionen verwenden. Wem das noch nicht reicht, kann über help(par) weitere Parameter ausfindig machen und deren Bedeutung ermitteln. Mit der Funktion par() lassen sich eine Reihe von Abfrage- und allgemeinen Änderungswünschen zu den graphischen Einstellungen realisieren.

Graphische Low-Level-Routinen

Bei Ergänzungswünschen zu einer Graphik helfen folgende Funktionen:

abline	abline(2,3) zeichnet Geraden (hier: y=2+3*x) abline(h=1) horizontale Linie f(x)=1 abline(v=5) vertikale Linie f(y)=5
lines(x,y)	Polygonzug durch die Punkte (x,y), x,y: Vektoren gleicher Länge
segments(x1,y1,x2,y2)	zeichnet Strecken von (x1,y1) nach (x2,y2)
points(x,y)	zeichnet die Punkte (x,y) x,y: Vektoren gleicher Länge
title("Entwicklung")	Überschriften
text(x,y,textxy)	zeichnet die Texte "textxy" wird an den Stellen (x,y)
symbols(x,y,circles=z)	Kreise um vom Umfang z (auch: squares, stars etc.)
legend	Legende legend(x=1,y=20, legend=c("DG","C"), lty=c(1,2))

Von der Graphik am Bildschirm zum Ausdruck

Ist eine Graphik im entsprechenden Bereich bereits erstellt worden, kann diese mit **dev.copy(postscript, 'name.ps')** als post-script Datei abgespeichert werden (der aktive Graphikbereich wird verwandt; vgl. dev.cur()). Es ist wichtig, dass nach dem Kopierbefehl ein **dev.off()** eingegeben wird, was den Vorgang sozusagen abschliesst. Das Bild steht nun als name.ps zur Verfügung.

Die zweite Möglichkeit ist die, das Bild frisch für die Datei zu erzeugen. Mit dem Befehl **postscript('name.ps')** wird eine leere Datei name.ps angelegt. Alle nun folgenden Graphikbefehle werden nicht an das Device motif() geschickt sondern in die Datei geschrieben. Ist der letzte nötige Graphikbefehl eingegeben worden, dann muss wie oben die Erzeugung mit **dev.off()** abgeschlossen werden.

8 Sonstige interessante/wichtige Kommandos

- sink() Umlenken der Ausgabe

  > sink("SplusAufgabe1.text")
  > 1 + 2
  > sink() # Zurück auf die Standardausgabe

- source()
  Sind syntaktisch richtige S-Plus-Anweisungen in einer Datei abgelegt worden, so lassen sich diese durch die Anweisung source(Dateiname) zur Ausführung bringen. Beispiel bei: source("myfunctions.src")

- cat('Die Wurzel von',x,'ist gleich',sqrt(x),'\n')
  schreibt den Text zwischen den Anführungszeichen, druckt das Ergebnis der Funktion und macht ganz am Ende einen Zeilenvorschub.

- print(paste('blabla',sqrt(x),'blabla')) macht im Prinzip das gleiche.
  Die Funktion paste() klebt Textstrings zusammen; nützlich z.B. für die Funktion title().

  > print(paste('blabla', x<-1:3 ,'blabla'))
  [1] "blabla 1 blabla" "blabla 2 blabla" "blabla 3 blabla"

  > cat('blabla',x<-1:3 ,'blabla')
  blabla 1 2 3 blabla>

- summary(objekt)
  gibt eine Zusammenfassung von objekt. Der Befehl ist generisch und reagiert je nach Beschaffenheit von objekt anders. Ist z.B. x ein numerischer Vektor, werden das Minimum, das Maximum, der Mittelwert sowie die 3 Quartile (in Vektorform) ausgegeben.

  > summary(x) # x <- 1:4

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.00	1.75	2.50	2.50	3.25	4.00

- quantile(x)

0%	25%	50%	75%	100%
1	1.75	2.5	3.25	4

- names(objekt) Namen von Teilobjekten von x ausgeben
  # interessant z.B. bei read.table()

  > x <− data.frame("Gewicht"=c(65,75),"Groesse"=c(168,175))
  > names(x)
  [1] "Gewicht" "Groesse"

  oder bei S-Plus eigenen Funktionen:
  > names(t.test(x))
  [1] "statistic" "parameters" "p.value" "conf.int" "estimate"
  [6] "null.value" "alternative" "method" "data.name"