Handleiding R

3/21/2006

A.W. van der Vaart

nabewerkt door M.A. Jonker

Faculteit der Exacte Wetenschappen
Vrije Universiteit

Deze handleiding is volledig gebaseerd op de handleiding S-plus die is geschreven door A.W. van der Vaart. Op enkele punten heeft M.A. Jonker de S-plus handleiding gewijzigd, zodat het diktaat te gebruiken is voor R. Dit heeft ze gedaan door te controleren of de S-plus functies nog bestaan en ogenschijnlijk nog hetzelfde werken. Zowel A.W. van der Vaart als M.A. Jonker voelen zich niet verantwoordelijk voor eventuele fouten die ontstaan door het gebruiken van deze handleiding.

Inhoud

1. MANUAL

Dit is een korte handleiding voor het gebruik van R in een UNIX-omgeving. Alleen de belangrijkste mogelijkheden van de taal komen hier aan de orde. Voor uitgebreidere informatie over R verwijzen we naar de user's manuals. Hierin worden bijvoorbeeld de vele mogelijkheden beschreven voor het gebruik van karakter variabelen, geavanceerde grafieken, en gespecialiseerde statistische technieken, zoals variantieanalyse, glim, nonparametrische regressie, tijdreeksanalyse en survival analyse. Uitgebreide informatie over R is te vinden op www.r-project.org of met het commando

```
> help.start()
```

nadat R is opgestart.

2. STARTEN EN STOPPEN VAN R

Het R-pakket wordt bijna altijd interactief gebruikt. Dat betekent, dat de opdrachten één voor één worden ingetypt en steeds wordt gewacht op de uitvoering. Een *prompt* geeft aan dat uitvoering van de vorige opdracht voltooid is en dat R klaar is voor de volgende opdracht. In het volgende nemen we aan dat R gestart wordt binnen X-windows.

Aangenomen dat de `UNIX`-prompt gelijk is aan het dollar teken '$' wordt een R-sessie gestart met hoofdletter R.

```
$ R
```

De default R-prompt is het groter dan teken '>'. De R sessie wordt besloten met `q()`.

```
> q()
$
```

Gedurende de sessie heeft `CONTROL`-C de gebruikelijke betekenis van het onderbreken van de uitvoering van het laatste commando.

Ieder R-commando wordt afgesloten met een `RETURN`. Indien een commando bij het geven van een `RETURN` nog niet af is, antwoordt R met een *vervolg-prompt*: dit is bij default het plus teken '+'. Omgekeerd kunnen meerdere opdrachten op één regel gegeven worden door ze te scheiden met een puntkomma '`;`'.

```
> 3+3; 1+1
[1] 6
[1] 2
> q( # onvolledig commando
+  ) # vervolg prompt
$
```

Hoofdletters en kleine letters worden door R verschillend geïnterpreteerd. Derhalve zijn `c` en `C` verschillende commando's.

3. EEN VOORBEELD SESSIE

De volgende voorbeeld sessie toont enkele van de mogelijkheden van R. Hieronder wordt alleen de input gegeven, tesamen met een korte toelichting. Het aanschouwen van de resultaten (door de sessie na te bootsen) geeft een snelle introductie tot R. De gebruikte commando's worden later uitgebreider beschreven.

`$ R`	start R sessie vanaf `UNIX`-prompt.
`> x <- 1:20`	maak een vector `x` met waarden 1, 2, 3, . . ., 20.
`> x`	print de value van `x`.
`> y <- rnorm(20)`	genereer een random steekproef ter lengte 20
`>`	uit de normale verdeling.
`> y <- x+ 2*y`	transformeer `y`.
`> cbind(x,y)`	vorm een matrix met kolommen `x` en `y` en
	print het resultaat.
`> x11()`	open een graphics window.
`> plot(x,y)`	plot `y` tegen `x`.
`> lsfit(x,y)$coef`	bereken de coëfficiënten van
	de kleinste-kwadraten-lijn.
`> abline(lsfit(x,y))`	voeg de kleinste-kwadraten-lijn toe aan de plot.
`> lines(spline(x,y))`	voeg ook een exacte spline fit toe.
`> x <- rnorm(50)`	genereer een normale steekproef ter grootte 50.
`> y <- rnorm(50)`	idem.
`> plot(x,y)`	plot een scatter diagram.
`> cor(x,y)`	bereken de correlatie tussen `x` en `y`.
`> z <- x+y`	bereken de som van `x` en `y` coördinaatswijs.
`> cor(x,z)`	bereken de correlatie.
`> hist(z,prob=T)`	plot een histogram.
`> help(hist)`	geef documentatie over de functie `hist`.
`> var(z)`	bereken de variantie van z.
`> u <- seq(-5,5,0.1)`	vorm een rij punten tussen -5 en 5 met
	stapgrootte 0.1.
`> v <- dnorm(u,0,sqrt(2))`	bereken een normale dichtheid.
`> lines(u,v)`	voeg een plot van de normale dichtheid toe.
`> {hist(z,xlim=c(-5,5),prob=T)`	herhaal de plot van het histogram,
	maar voer de opdracht nog niet uit.
`+ lines(u,v)}`	herhaal en voer de opdracht uit.
`> x[x<0]`	selecteer de negatieve elementen in `x`.
`> sum(x<0)`	tel het aantal negatieve elementen in `x`.
`> u <- seq(-pi,pi,length=100)`	vorm een regelmatig rooster van 100 punten.

`> plot(cos(u),sin(u),type="l")`	teken een cirkel.
`> plot(sort(x),1:50/50,`	plot de empirische verdelingsfunctie van `x`.
`+ type="s",ylim=c(0,1))`	
`> lines(u,pnorm(u))`	voeg de echte verdelingsfunctie toe.
`> q()`	verlaat R.
$	de UNIX-prompt.

4. HULP

Met behulp van de functie `help` is het mogelijk vanachter de terminal uit te vinden wat een bepaald commando voor gevolgen heeft. Een beschrijving van de standaardfunctie `hist` wordt als volgt verkregen.

```
> help(hist)
```

Een functienaam die speciale karakters bevat moet omsloten worden door aanhalingstekens.

```
> help("%*%")
```

Dit roept informatie op over de operatie `%*%` (matrixproduct). Voor ieder standaard R commando kan op deze manier informatie worden opgevraagd.

5. VECTOREN

De simpelste data-structuur in R is de *vector*. Een vector is een enkel object bestaande uit een rij van getallen, tekst of logische symbolen. Een vector met de naam `x`, bestaande uit de vier getallen 10, 5, 3, 6, kan worden gemaakt met de functie `c`.

```
> x <- c(10, 5, 3, 6)
> x
[1] 10  5  3  6
```

De omgekeerde pijl is het *toekenningssymbool*; het wordt getypt als het kleiner dan teken gevolgd door een streepje. (Het symbool `=` heeft binnen R een andere rol.)

De functie `c` vormt uit een willekeurig aantal vectoren een enkele vector door concatenatie. Een enkel getal wordt door R beschouwd als een vector van lengte 1.

```
> y <- c(x,0.555,x,x) # een vector ter lengte 13 met 11-de coördinaat 5
> y
[1] 10.000  5.000  3.000  6.000  0.555 10.000  5.000  3.000  6.000 10.000
[11]  5.000  3.000  6.000
```

Zoals in de bovenstaande voorbeelden blijkt, kan de waarde van een vector worden geprint door zijn naam te typen. Hierbij verschijnt aan het begin van iedere regel binnen vierkante haakjes een nummer dat de positie van de eerst volgende coördinaat binnen de vector

aangeeft. De functie `round` geeft controle over het aantal decimalen. Het tweede argument is het gewenste aantal cijfers achter de komma.

```
> round(y,1)          # rond af op 1 cijfer achter de komma
[1] 10.0  5.0  3.0  6.0  0.6 10.0  5.0  3.0  6.0  10.0  5.0  3.0  6.0
```

Rekenen met Vectoren; Rekenkundige Functies

Rekenkundige bewerkingen op vectoren worden coördinaatsgewijs uitgevoerd. Bijvoorbeeld, `x * x` is een vector met dezelfde lengte als `x` waarvan de coördinaten de kwadraten van de corresponderende coördinaten van `x` zijn.

```
> x
[1] 10  5  3  6
> z <- x * x
> z
[1] 100  25   9  36
```

De symbolen voor de elementaire rekenkundige operaties zijn de gebruikelijke `+`, `-`, `*`, `/`. Machten worden verkregen met `^`. De meeste standaardfuncties zijn in R beschikbaar en werken eveneens coördinaatsgewijs op vectoren. Zie Tabel 1 voor de namen.

```
>                   # vervolg voorgaand display
> log(x)
[1] 2.302585 1.609438 1.098612 1.791759
> exp(log(x))^2     # equivalent aan x^2
[1] 100  25   9  36
```

functienaam	operatie
`sqrt`	wortel
`abs`	absolute waarde
`sin cos tan`	goniometrische functies
`asin acos atan`	inverse goniometrische functies
`sinh cosh tanh`	hyperbolische functies
`asinh acosh atanh`	inverse hyperbolische functies
`exp log`	exponentiële functie en natuurlijke logaritme
`log10`	logaritme met basis 10
`gamma lgamma`	gamma en log-gamma functie
`floor ceiling trunc`	entier, boven-entier en gehele deel
`round`	afronding
`sign`	teken

Tabel 1. Rekenkundige standaardfuncties

Vectoren in één expressie behoeven niet van dezelfde lengte te zijn. Als dit niet het geval is wordt de kortste vector herhaald en geconcateneerd aan zichzelf even vaak als nodig is om de lengte van de langste vector te matchen. Een eenvoudig voorbeeld is een binaire operatie tussen een vector en een constante.

```
> a <- sqrt(x) + 1
```

In dit geval wordt de constante 1 (een vector ter lengte 1) eerst vervangen door een vector met enen van dezelfde lengte als `x`, alvorens de optelling coördinaatsgewijs wordt uitgevoerd. In het volgende voorbeeld wordt de vector `x` met 4 coördinaten 3 $^1/_4$ keer herhaald om met de vector `y` ter lengte 13 in lengte overeen te komen.

```
> z <- x * y      # y is gelijk aan c(x,0.555,x,x) als tevoren
> z
Warning messages:
Length of longer object is not a multiple of the length of the shorter
object in:  x * y
> z
 [1] 100.00  25.00   9.00  36.00  5.55  50.00  15.00  18.00  60.00  50.00
[11]  15.00  18.00  60.00
```

Aangezien deze situatie nogal gekunsteld is, geeft R een waarschuwing. Mits zekerheid bestaat dat het beoogde effect bereikt is, kan een waarschuwing gewoon worden genegeerd.

Bij rekenkundige bewerkingen geldt de gebruikelijke prioriteitsvolgorde. Desgewenst kunnen ronde haakjes '`(`' en '`)`' worden gebruikt om deze volgorde te doorbreken. Ook in gevallen van twijfel is het gebruik van haakjes aanbevolen. Tabel 2 geeft de prioriteitsvolgorde voor alle operators in R.

operatie	naam	prioriteit
`$`	componentselectie	HOOG
`[ [[`	coördinaatselectie	
`^`	machtstransformatie	
`-`	unair min	
`:`	sequentie	.
`%`*naam*`%`	speciale operatie	.
`* /`	vermenigvuldiging en deling	.
`+ -`	plus en min	
`< > <= >= == !=`	logische vergelijking	
`!`	logische ontkenning	
`& | && ||`	en, of	
`<- _ ->`	toekenning	LAAG

Tabel 2. Prioriteitsvolgorde van R-operaties van hoog naar laag.

Genereren van Roosterpunten

Regelmatige rijen getallen spelen een belangrijke rol in het gebruik van talloze R-functies. Zulke *roosters* kunnen op meerdere manieren worden gegenereerd. De snelste operatie is de dubbele punt `:`.

```
> index <- 1:20
> index
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Omgekeerd geeft `20:1` dezelfde rij in dalende volgorde. De functie `seq` levert algemenere rijtjes. Hierin wordt ofwel de gewenste stapgrootte ofwel het aantal gewenste roosterpunten als argument opgegeven, tesamen met het eerste en laatste roosterpunt.

```
> u <- seq(-3,3,by=.5)      # een regelmatig rooster met stapgrootte .5
> u
[1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0
```

Precies hetzelfde effect hebben de volgende commando's.

```
> u <- seq(-3,3,length=13) # geeft 13 roosterpunten
> u <- (-6):6/2
```

De functie `rep` repliceert een gegeven vector. Als het tweede argument een enkel getal is, wordt het eerste argument evenzoveel malen herhaald.

```
> rep(1:4,4)
[1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
```

Het tweede argument van `rep` kan ook een vector zijn van dezelfde lengte als het eerste argument. In dat geval geeft iedere coördinaat aan hoe vaak de corresponderende coördinaat van de eerste vector moet worden herhaald.

```
> rep(1:4,rep(4,4))
[1] 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4
```

Logische en Karakter Vectoren

Naast numeriek kunnen de coördinaten van een vector ook de logische symbolen `FALSE` ('*false*') and `TRUE` ('*true*'), of *karakter strings* (rijtjes letters en/of symbolen) zijn. Iedere vector is echter ongemengd in de zin, dat alle coördinaten van hetzelfde type zijn. Derhalve zijn er numerieke, logische en karakter vectoren in R.

Logische vectoren worden meestal gecreëerd uit condities.

```
> x
[1] 10  5  3  6
```

```
> y <- x>9          # y[i] is T als x[i]>9
> y
[1] TRUE FALSE FALSE FALSE
```

Net als met rekenkundige bewerkingen wordt in condities de lengte van de kortste vector door recycling aangepast aan de lengte van de langste vector. De voorgaande toekenning is identiek aan het nodeloos lange `y <- x>rep(9, 4)`.

De *logische vergelijkingen* zijn `<`, `<=`, `>`, `>=`, `==` voor gelijkheid en `!=` voor ongelijkheid. (Het symbool `=` is geen logische operator!) De logische operatoren *en*, *of* en *ontkenning* worden gegeven door `&`, `|` en `!`. Analoog aan de rekenkundige bewerkingen kunnen deze operatoren op vectoren worden toegepast.

```
> (3<x) & (x<10)    # gebruik niet (3<x<10)!
[1] FALSE TRUE FALSE TRUE
> (!y)
[1] FALSE TRUE TRUE TRUE
```

De ronde haakjes rondom `!y` zijn niet altijd noodzakelijk, maar kunnen in dit geval niet worden gemist. Een voor de hand liggende fout is het afkorten van het eerste commando tot `(3<x<10)`. Deze uitdrukking is syntactisch correct, maar heeft niet het gewenste resultaat. (Het commando wordt geïnterpreteerd als `(3<x) <10` met het resultaat `TRUE TRUE TRUE TRUE`, gebruikmakend van 'coërcie' als beschreven in de volgende paragraaf.)

Op logische vectoren kunnen ook rekenkundige bewerkingen worden toegepast. In dat geval worden de logische waarden `FALSE` en `TRUE` eerst vervangen door 0 en 1 (zogenaamde *coërcie*). Bijvoorbeeld, de functie `sum(x)` berekent de som van de coördinaten van een vector `x`. Daarom geeft `sum(y)` het aantal keer `TRUE` in de logische vector `y`.

```
> sum(x>9)          # aantal coördinaten x[i] > 9
[1] 1
```

Een *karakter vector* ter lengte 1 wordt binnen R aangegeven door een rij letters of cijfers omgeven door aanhalingstekens, zoals `"x"` of `"x-waarden"`. In het volgende wordt een karakter vector ter lengte drie op twee manieren gecreëerd.

```
> x <- c("row a", "row b", "row c")
> x
[1] "row a" "row b" "row c"
> letters[1:3]       # een standaard karakter vector
[1] "a" "b" "c"
> paste("row",letters[1:3])
[1] "row a" "row b" "row c"
```

Een alternatief voor impliciete coërcie is om een logische vector permanent in een numerieke vector te veranderen met de functie `mode`.

```
> y
```

```
[1] TRUE FALSE FALSE FALSE
> mode(y) <- "numeric"
> y
[1] 1 0 0 0
> mode(y) <- "logical"
> y
[1] TRUE FALSE FALSE FALSE
```

Missing Data

Op de regel dat een vector ongemengd moet zijn, bestaat één uitzondering. De speciale waarde '`NA`' mag in vectoren van ieder type voorkomen. Deze waarde kan worden gebruikt voor missing data. ('`NA`' is een afkorting voor '*not available*'.) Bij het toepassen van berekeningen heeft iedere expressie waarin minstens één '`NA`' waarde voorkomt, de waarde '`NA`'. Met de functie `is.na` kan worden nagegaan welke coördinaten van een vector de waarde `NA` bezitten.

```
> x <- c(NA,1,4)
> x
[1] NA  1  4
> x+1
[1] NA  2  5
> is.na(x)
[1] TRUE FALSE FALSE
```

Subscripts

Een gedeelte van een vector kan worden geselecteerd volgens de algemene vorm

$$\texttt{x}[\textit{subscript}].$$

Het simpelste voorbeeld is het selecteren van de i-de coördinaat.

```
>  x[i]        # i-de coördinaat van x
```

Algemener kan *subscript* de volgende drie vormen nemen.

- Een vector van positieve natuurlijke getallen. Het resultaat is de vector van die coördinaten van `x` waarvan de nummers in *subscript* voorkomen. De coördinaten van een vector worden altijd gedacht als genummerd volgens 1, 2, 3, ...

```
> x
[1] 10  5  3  6
> x[1:3] ; x[c(4,1,4)] # de tweede geeft de vector (x[4], x[1], x[4])
[1] 10  5  3
 [1]  6 10  6
```

- Een logische vector van dezelfde lengte als `x`. Het resultaat is een vector met die coördinaten van `x` waarvoor de corresponderende coördinaten van de logische vector de waarde `T` bezitten.

```
> y <- x>9    # y is (TRUE, FALSE, FALSE, FALSE)
> x[y]        # geeft x[1]
[1] 10
> x[x>9]
[1] 10
```

- Een vector van negatieve getallen. Alle coördinaten van `x` behalve de gespecifeerde worden geselecteerd.

```
> x[-(1:2)]   # geeft (x[3], x[4])
[1] 3 6
```

Andersom kunnen ook toekenningen aan gesubscripte vectoren worden gemaakt. Daarbij blijven ongeselecteerde coördinaten onveranderd.

```
> x[x<0] <- 0         # maakt alle negatieve coördinaten gelijk aan 0
> y[y<0] <- -y[y<0]   # hetzelfde resultaat als met y <- abs(y)
```

Een vector moet bestaan voordat je een deel ervan kunt selecteren. Een numerieke vector ter lengte `n` kun je creëren met de functie `numeric`.

```
> x <- numeric(5)
> x
[1] 0 0 0 0 0
```

Een andere mogelijkheid is gebruik te maken van de functie `rep`.

6. FUNCTIES

Alle bewerkingen in R worden uitgevoerd door *functies*. Een aantal daarvan, zoals `c` en `seq` is in het voorgaande al behandeld. In totaal kent R zo'n 500 standaardfuncties en een groeiend aantal locale functies. In deze handleiding wordt slechts een beperkt aantal functies besproken. Daarnaast kan iedere gebruikster haar eigen functies schrijven. Alle functies worden op dezelfde manier gebruikt.

Een functie wordt aangeroepen door het typen van zijn naam, tesamen met eventuele argumenten.

```
> functienaam(arg1,arg2,...)
```

De ronde haakjes zijn verplicht, zelfs als de functie geen argumenten behoeft. Zonder de haakjes reageert R met het listen van de definitie van de functie.

```
> functienaam() # voer de functie functienaam uit (zonder argumenten)
```

```
> functienaam   # list de definitie van de functie functienaam
```

De *argumenten* kunnen verplicht zijn of optioneel. Optionele argumenten mogen bij de aanroep van de functie worden weggelaten. In dat geval gebruikt R, indien nodig, zogenaamde default waarden. Ieder argument heeft een naam. Optionele argumenten worden vaak gespecifeerd door middel van de constructie `argumentnaam =` *waarde*. Bijvoorbeeld, in

```
> seq(-3,3,length=1000)
```

bezit het derde argument de naam '`length`'. Het voordeel van het specificeren van argumenten bij naam is, dat de definitievolgorde van de argumenten kan worden doorbroken en onbelangrijke argumenten kunnen worden weggelaten. Bijvoorbeeld, in de definitie van de functie `seq` is `by` het derde en `length` het vierde argument. Bij de aanroep `seq(-3,3,1000)` interpreteert R het getal 1000 als het derde argument in de definitie van `seq`. Daar dit `by` is heeft deze aanroep een ander effect dan `seq(-3,3,length=1000)`.

Argumentnamen mogen worden afgekort tot de eerste paar letters die de argumentnaam nog uniek bepalen.

```
> seq(-3,3,l=1000)
```

Sommige functies bezitten een variabel aantal argumenten. Een voorbeeld hiervan is de concatenatiefunctie `c`.

Data-Manipulatie

De functies

```
> length(x); sum(x); prod(x); max(x); min(x)
```

geven respectievelijk de lengte van de vector `x` en de som, het product, het maximum en het minimum van zijn coördinaten. De laatste vier functies kunnen ook op meerdere vectoren tegelijk worden toegepast.

```
> sum(x,y); prod(x,y); max(x,y), min(x,y)
```

Deze commando's geven som, product, maximum en minimum van alle coördinaten van de vectoren `x` en `y` tesamen. De eerste is bijvoorbeeld equivalent aan `sum(c(x,y))`.

De functie `cumsum(x)` produceert een vector met dezelfde lengte als `x`, waarvan de i-de coördinaat gelijk is aan de som van de eerste i coördinaten van `x`.

```
> cumsum(rep(1,10))
[1]  1  2  3  4  5  6  7  8  9 10
```

Een vector kan op grootte gesorteerd worden met behulp van `sort`.

```
> x <- c(2, 6, 4, 5, 5, 8, 8, 1, 3, 0)
> length(x)
[1] 10
```

```
> sort(x)
[1] 0 1 2 3 4 5 5 6 8 8
```

Met `order` krijg je de permutatie die nodig is om de vector op grootte geordend te krijgen. Deze kan worden gebruikt om een tweede vector met een eerste mee te permuteren.

```
> order(x)
[1] 10  8  1  9  3  4  5  2  6  7
```

In statistische termen is `sort(x)` de vector van *order statistics*; `order(x)` is iets anders! Uit de definities volgt, dat `x[ order(x) ]` eveneens de order statistics van `x` geeft.

De functie `rev` draait de volgorde van de coördinaten van een vector om. Derhalve is `rev(sort(x))` de vector van order statistics in dalende volgorde.

```
> rev(sort(x))
[1] 8 8 6 5 5 4 3 2 1 0
> rev(x)
[1] 0 3 1 8 8 5 5 4 6 2
```

De functie `rank(x)` geeft de *rangnummers* van `x`. In het geval van ties in `x` worden gemiddelde rangnummers gegeven, zoals gebruikelijk in de statistiek.

```
> rank(x)
[1] 3.0 8.0 5.0 6.5 6.5 9.5 9.5 2.0 4.0 1.0
```

Het commando `unique(x)` geeft de verschillende waarden in `x` en de logische vector `duplicated(x)` geeft voor iedere coördinaat aan of dezelfde waarde al eerder is voorgekomen.

```
> unique(x)
[1] 2 6 4 5 8 1 3 0
> duplicated(x)
[1] FALSE FALSE FALSE FALSE TRUE FALSE TRUE FALSE FALSE FALSE
```

Tenslotte geeft `diff(x)` de *spacings* van `x`, een vector met één coördinaat minder, die de verschillen van opeenvolgende coördinaten van `x` bevat. Met behulp van optionele argumenten kan `diff` ook hogere orde verschillen produceren, met een lag naar keuze.

```
> x
[1] 2 6 4 5 5 8 8 1 3 0
> diff(x)                        # de vector x[2]-x[1], x[3]-x[2], ...
[1]  4 -2  1  0  3  0 -7  2 -3
> diff(x, lag=2)                 # de vector x[3]-x[1], x[4]-x[2], ...
[1]  2 -1  1  3  3 -7 -5 -1
> diff(x,lag=1,differences=2)    # hetzelfde als  diff(diff(x))
[1] -6  3 -1  3 -3 -7  9 -5
```

Statistische Samenvattingen

Een aantal functies geeft statistische samenvattingen. De namen spreken meestal voor zich. Van de in Tabel 3 genoemde functies zijn sommige geen standaardfuncties, maar locale functies.

functienaam	operatie
`mean(x)`	gemiddelde
`mean(x, trim=`α`)`	getrimd gemiddelde
`median(x)`	mediaan
`var(x)`	variantie, covariantiematrix
`mad(x)`	mediaan absolute deviatie
`range(x)`	de vector `(min(x),max(x))`
`quantile(x,prob)`	kwantielen
`summary(x)`	gemiddelde, minimum, maximum, kwartielen
`stem(x)`	stem-en-leaf-plot
`var(x,y)`	covariantie
`cor(x,y)`	correlatiecoëfficiënt
`acf(x,plot=F)`	auto of partiële correlatie coefficiënten

Tabel 3. Statistische samenvattingen

De kwantielfunctie heeft twee vectoren als argumenten. Hij geeft de empirische kwantielen van de eerste vector voor de percentages gespecificeerd in de tweede.

```
> quantile(x,c(0.05,0.1)) # het 5% en 10% kwantiel van x
```

Een *stem-and-leaf-plot* wordt verkregen met de functie `stem`.

```
> stem(x)
Decimal point is at the |

  -2 | 221
  -1 | 87
  -1 | 443320
  -0 | 99888665
  -0 | 44433221
   0 | 01113334444
   0 | 88899
   1 | 002
   1 | 5669
```

Kansverdelingen en Toevalsgetallen

Voor ieder van de meest gebruikte kansverdelingen zijn vier functies beschikbaar, die respectievelijk de verdelingsfunctie, de kansdichtheid, de kwantielfunctie en toevalsgetallen berekenen. De namen van deze functies zijn steeds een code voor de kansverdelingen, voorafgegaan door de letters:

- `p` (voor probability): verdelingsfunctie.
- `d` (voor density): kansdichtheid.
- `q` (voor quantile): kwantielfunctie.
- `r` (voor random): toevalsgetallen.

Voor de normale verdeling zijn dit bijvoorbeeld de functies.

```
> pnorm(x,m,s); dnorm(x,m,s); qnorm(u,m,s); rnorm(n,m,s)
```

Hierin zijn `m` en `s` optionele argumenten die verwachting en standaardafwijking (niet variantie!) aangeven; de `n` in `rnorm` is het aantal te genereren toevalsgetallen en `x` en `u` zijn vectoren in het domein van de functies. (Dus de coördinaten van `u` altijd tussen 0 en 1.) Tabel 4 geeft een overzicht van kansverdelingen waarvoor soortgelijke functies beschikbaar zijn, tesamen met de argumenten. De cijfers in de kolom 'Defaults' zijn de default waarden voor de parameters.

Code	Verdeling	Parameters	Defaults
beta	beta	shape1, shape2	-, -
binom	binomiaal	size,prob	-, -
cauchy	Cauchy	location, scale	0, 1
chisq	chikwadraat	df	-
exp	exponentieel	rate	1
f	F	df1, df2	-, -
gamma	gamma	shape, rate	-, 1
geom	geometrisch	prob	-
hyper	hypergeometrisch	m,n,k	-,-,-
lnorm	lognormaal	meanlog, sdlog	0, 1
logis	logistiek	location, scale	0, 1
nbinom	negatief binomiaal	size,prob	-,-
norm	normaal	mean, sd	0, 1
pois	Poisson	lambda	1
t	Student's t	df	-
unif	homogeen	min, max	0, 1
weibull	Weibull	shape	-
wilcox	Wilcoxon	m,n	-, -

Tabel 4. Codes voor kansverdelingen

De functie `sample` doet aselecte trekkingen uit een gegeven populatie met of zonder teruglegging. Als alle elementen van de populatie worden getrokken zonder teruglegging (de

default), dan is het resultaat een random permutatie.

```
> sample(x,3) # 3 trekkingen uit coördinaten van x zonder teruglegging
> sample(x)   # random permutatie van de coördinaten van x
> sample(x,100,replace=T) # 100 trekkingen met teruglegging
> sample(c(0,1),100,replace=T,prob=c(0.3,0.7))
        # 100 trekkingen met teruglegging, waarbij 0 met kans 0.3
        # en 1 met kans 0.7 getrokken wordt
```

De 'toevalsgetallen' worden genereerd volgens een algoritme dat gestart wordt vanaf de waarde van de vector `.Random.seed`. De waarde van deze vector wordt voortdurend aangepast, zodat steeds nieuwe toevalsgetallen worden verkregen. Soms is het echter van belang om een voorgaand resultaat exact te reproduceren. Dit is mogelijk door de waarde van `.Random.seed` te bewaren.

```
> seed <- .Random.seed
> seed
[1] 61 60 42  6 28  3 12 22 32 29 39  2
> rnorm(5)
1] -1.4643258  1.6484454 -1.1360949 -0.8188030  0.4044431
> .Random.seed <- seed
> rnorm(5)
1] -1.4643258  1.6484454 -1.1360949 -0.8188030  0.4044431
```

Om betere ('toevalligere') resultaten te krijgen is het ook mogelijk de random number generator op een plaats van eigen keuze te starten. Dit kan ervoor zorgen dat twee simulatiestudies niet van dezelfde rij toevalsgetallen gebruik maken. Een gemakkelijke manier is gebruik te maken van `set.seed` met als argument een zelf gekozen natuurlijk getal.

```
> set.seed(i)
```

---

7. GRAFIEKEN

Het produceren van grafieken behoort tot de kern van het R-programma. Grafische output kan zowel op het scherm worden bekeken als op papier worden geprint. Wanneer R gestart is binnen het X-windows systeem zal een *graphics window* automatisch worden geopend wanneer een plotopdracht gegeven wordt. De output van plotopdrachten verschijnt automatisch in dit graphics window. Grootte en plaats van het graphics window kunnen op de gebruikelijke manier worden veranderd. Na verandering van de grootte wordt het plaatje opnieuw getekend, op zo'n manier dat de grafiek het gehele window vult.

Er zijn twee soorten plotopdrachten in R.

- *High level plotfuncties* produceren volledige plaatjes en wissen zonodig de output van een voorgaande plotopdracht uit.
- *Low level plotfuncties* zijn bedoeld om grafische output aan een eerder gemaakt plaatje toe te voegen.

Voor het maken van twee grafieken in één plaatje is derhalve één high level plotfunctie nodig, en één low level plotfunctie.

De meest gebruikte plotfunctie is `plot`. Zonder opties levert deze een scatterplot van de coördinaten van de ene vector tegen die van een tweede vector.

```
> x <- rnorm(50); y <- rnorm(50)
> plot(x,y,main="figuur 1")           # zie figuur 1
```

Met de optie `type="l"` worden opeenvolgende punten verbonden door lijnstukjes. Dit maakt `plot` geschikt voor het plotten van een grafiek.

```
> u <- seq(0,4*pi,by=0.05); v <- sin(u)
> plot(u,v,type="l",xlab="",ylab="sin",main="figuur 2")  # zie figuur 2
```

Andere high level plotting functies maken meer specifieke plots. Plots van statistische betekenis zijn het *histogram*, de *boxplot* en de *qq-plots*.

```
> x <- rnorm(50)
> hist(x, main="figuur 3")       # figuur 3
> text(0,5,"histogram")          # print "histogram" in het punt (0,5)
> boxplot(x, main="figuur 4")
> qqnorm(x, main="figuur 5")     # figuur 5
```

De range van de assen in een plot is gelijk aan de coördinaten van de vectoren `xlim` en `ylim`. Bij simpel gebruik is het handig om R default waarden voor deze vectoren te laten kiezen. Voor speciale effecten kunnen de waarden van `xlim` en `ylim` echter als optionele argumenten aan de plotfuncties worden meegegeven.

Hetzelfde geldt voor de labels op de assen, en boven- en ondertitels. Tabel 5 geeft een overzicht van enkele van zulke optionele argumenten. Vele andere aspecten van de plots kunnen worden gecontroleerd met `par`. Zie `help` voor nadere informatie.

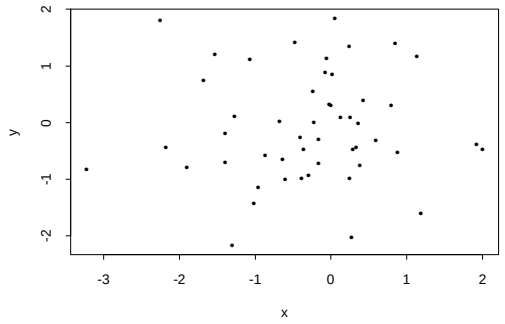
De functie `lines` is de low level versie van `plot(...,type="l")`. Deze functie kan worden gebruikt om een tweede grafiek over een eerste heen te tekenen. Beschouw bijvoorbeeld het plotten van een histogram en de ware dichtheid van een normale steekproef in één plaatje.

```
> x <- rnorm(100)
> range(x)
[1] -3.226397  2.001791    # default wordt de range in hist(x):  (-4,3)
> hist(x,xlim=c(-4,4), xlab="", prob=T, main="figuur 6")
> u <- seq(-4,4,by=0.1)
> lines(u,dnorm(u))
```

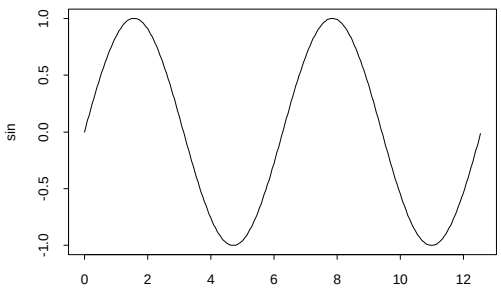
Meerdere plaatjes kunnen naast en onder elkaar in één plot getekend worden met behulp van de functie `par`.

```
> par(mfrow=c(r,k))
```

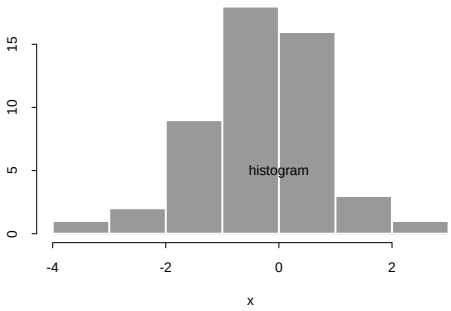
figuur 1



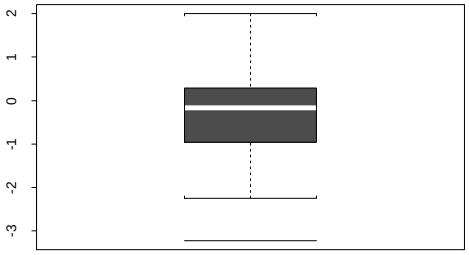
figuur 2



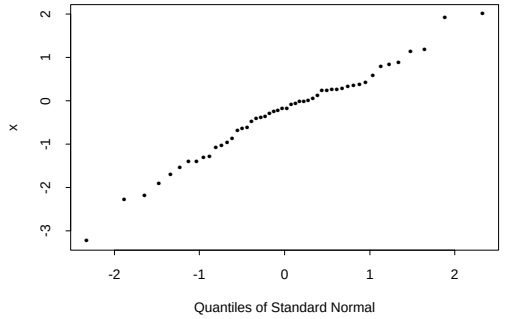
figuur 3



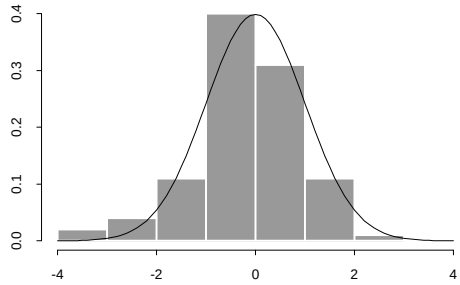
figuur 4



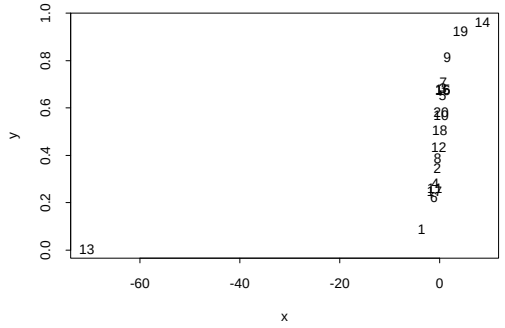
figuur 5



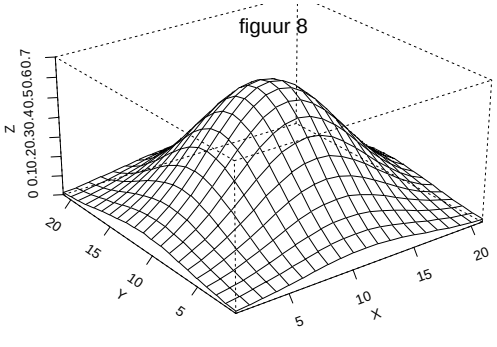
figuur 6



figuur 7



figuur 8



De eerst volgende $(r \times k)$ plots worden dan in één plaatje gezet, in respectievelijk r rijen en k kolommen.

Een extra *graphics window* kan worden geopend met het commando

```
> x11()
```

De output van de plotopdrachten verschijnt nu in het extra window.

Om plaatjes te printen of te bewaren als een postscript-file moet er een postscript 'graphics device' geopend worden. Het algemene schema om de plaatjes te printen op de Oce-printer is als volgt.

```
> postscript("|lpr -Poce")
> ...
>   plotopdrachten
> ...
> dev.off()                # stuurt alle plaatjes naar de printer
```

De plaatjes kunnen worden verzameld en in één keer naar de printer worden gestuurd. Het `postscript` commando maakt het *current graphics device* gelijk aan postscript. Hierna wordt alle uitvoer van grafische commando's omgezet in postscript-code en in een file geschreven. (Het schrijven vindt pas plaats nadat een nieuw graphics commando is gegeven.) Het postscript graphics device wordt gesloten met behulp van `dev.off`. Op dat moment wordt de postscript file automatisch naar de (postscript) printer gestuurd (mits een default printer commando is ingesteld). Bij default wordt het papier (een A4 pagina) volledig gevuld. Dit geeft vaak geen mooie verhoudingen van de x-as en y-as. Met de opties `height` en `width` kan dit worden verbeterd. Bijvoorbeeld,

```
> postscript("|lpr -Poce",he=8,wi=8)     # een plaatje van 8 x 8 inch
```

Het algemene shema om een plaatje als postscript-file op te slaan is

```
> postscript("filenaam.ps")
>   plotopdrachten
> dev.off()
```

Opnieuw kunnen de afmetingen van het plaatje worden gewijzigd met de opties `height` en `width`.

Het is mogelijk afwisselend output naar het graphics window en naar de postscript file te sturen. Het is bovendien mogelijk meerdere graphics windows tegelijk te openen. Eén van de graphics devices (een window of postscript) is steeds het current device en ontvangt de output voor de volgende plotopdracht.

```
> x11()                    # opent graphics window
> x11()                    # opent tweede graphics window
> postscript()             # opent postscript device
> dev.list()               # geeft alle open devices
  X11   X11 postscript
    2     3          4
> dev.cur()                # het current device
 postscript
          4
```

```
> dev.set(2)              # maakt het graphics device 2 current
  X11
    2
```

optienaam	betekenis
`xlab="`*xlabel*`"`	Label op de x-as
`ylab="`*ylabel*`"`	Label op de y-as
`xlim=c(`*l*`,`*r*`)`	Range van de x-as van *l* tot *r*
`ylim=c(`*l*`,`*h*`)`	Range van de y-as van *l* tot *h*
`main="`*titel*`"`	Titel boven de plot
`sub="`*subtitel*`"`	Titel onder de plot
`lty=`*n*	Lijntype: *n*=1: normaal; *n*=2, 3,. . .: gebroken

Tabel 5. Enkele plot-opties

Enkele High Level Plotfuncties

`> plot(x,y)`	Scatterplot de punten `(x[i],y[i])`.
`> plot(x,y,type="l")`	Plot de punten `(x[i],y[i])` met opeenvolgende punten verbonden door een rechte.
`> plot(x,y,type="n")`	Plot de assen, maar niet de punten.
`> plot(x,y,type="s")`	Plot een trapfunctie.
`> plot(mat)`	Plot de punten `(mat[i,1],mat[i,2])`.
`> matplot(matx,maty)`	Plot kolommen van matrices tegen elkaar.
`> hist(x)`	Histogram van vector `x`.
`> boxplot(x)`	Boxplot van vector `x`.
`> qqnorm(x)`	qq-plot tegen de normale verdeling.
`> qq`*verd*	qq-plot tegen de kansverdeling met code *verd*.
`> qqplot(x,y)`	Empirische qqplot van `x` tegen `y`.
`> persp(z)`	Quasi drie-dimensionale plot.

Enkele Low Level Plotfuncties

`> lines(x,y)`	Plot van de punten `(x[i],y[i])` met opeenvolgende punten verbonden door een rechte.
`> title("`*titel*`","`*ondertitel*`")`	Print titels.
`> text(x,y)`	Plot coördinaatnummers op de punten `(x[i],y[i])`.
`> text(x,y,label)`	Plot de karakter string `label[i]` op `(x[i],y[i])`.
`> abline(a,b)`	Voeg de lijn `y=a+bx` toe.
`> points(x,y)`	Voeg de punten `(x[i],y[i])` toe.
`> polygon(x,y)`	Veelhoek met hoekpunten `(x[i],y[i])`.
`> mtext("`*tekst*`",side,line)`	Plot *tekst* in een kantlijn.
`> legend()`	Voeg een legenda toe.

Figuren 7 en 8 illustreren nog twee andere grafische mogelijkheden. Ze werden op de volgende manier verkregen.

```
> x <- rcauchy(20); y <- runif(20);
> plot(x,y,type="n", main="figuur 7"); text(x,y)
>
> u<- dnorm(seq(-1,1,0.1),0,0.5)
> z <- outer(u,u)
> persp(z, main="figuur 8")              # een bivariate dichtheid
```

Sommige R-functies maken gebruik van de muis.

Met de functie `identify(x,y,labels)` kun je erachter komen welk punt een gegeven punt in een puntenwolk is, bijvoorbeeld een uitbijter.

```
> plot(x,y)
> identify(x,y,labels)
```

Klik vervolgens met de linker knop van de muis op de plaats van een punt. Als dit het punt `(x[i],y[i])` is, verschijnt vervolgens op de plaats van het punt de `i`-de coördinaat van de character vector `labels`. Het default label is `i`. Je kunt dit herhalen. Een druk op de rechter knop van de muis beëindigt `identify()`.

Het R-commando `locator` is handig als je de precieze coördinaten van een aantal punten op het scherm wilt weten, bijvoorbeeld om daar tekst af te drukken. Start met het typen van:

```
> z <- locator(n)
```

Klik vervolgens op n punten in het graphics window. De coördinaten van deze punten worden opgeslagen in de list `z`. Het is extra handig om `locator` te combineren met `text`.

```
> text(locator(1),"Tekst op de juiste plaats")
```

De tekst verschijnt nu op de plaats in de grafiek waar met de muis geklikt wordt.

Door na `locator(n=2,type="l")` op twee punten in het graphics window te klikken kan een lijnstuk in de grafiek worden getekend.

8. LISTS

Een *list* is ruwweg een vector waarvan de componenten verschillende typen R-objecten mogen zijn. Een voorbeeld is een list bestaande uit een numerieke vector, een karakter vector, een andere list (sublist) en een functie. Bij elementair gebruik van R is de list vooral belangrijk, omdat de resultaten van standaard functies vaak uit een list bestaan. Beschouw bijvoorbeeld het resultaat van de functie `lsfit`. In zijn eenvoudigste vorm berekent deze de kleinste-kwadraten-lijn van regressie van `y` op `x`.

```
> x <- 1:5; y <- x +rnorm(5,0,0.25)
> z <- lsfit(x,y)
> z
$coef:
  Intercept         X
 -0.3286285 1.059901
```

```
$residuals:
[1]  1.1562909 -2.1479019  1.4187857 -1.0190293  0.5918546


$intercept:
[1] TRUE
```

In dit voorbeeld is de waarde van `lsfit(x,y)` toegekend aan `z`, een list met als eerste component een (1x2)-matrix bestaande uit intercept en helling van de kleinste-kwadraten-lijn; de tweede component is een vector met de waarden van de residuen en de derde component is een logische vector ter lengte 1 die aangeeft of al dan niet een intercept in het model is opgenomen. (In werkelijkheid heeft de uitvoer van `lsfit` nog meer componenten, maar deze zijn hier weggelaten.) De drie componenten hebben de namen: `coef`, `residuals` en `intercept`.

De componenten van een list kunnen op twee manieren geselecteerd worden.

- Per componentnummer: `z[[i]]` is de i-de component. Gebruik dubbele vierkante haken!
- Per componentnaam: `z$`*naam* is de component met naam *naam*.

```
>                    # vervolg van het voorgaande display
>                    # de volgende 4 opdrachten hebben identieke uitvoer
> z[[2]]
> z$residuals
> z$r
[1]  1.1562909 -2.1479019  1.4187857 -1.0190293  0.5918546
```

In dit geval is de geslecteerde component van de list een vector ter lengte 5. Volgens het format voor het selecteren van coördinaten van vectoren kan vervolgens een coördinaat van deze vector worden geselecteerd.

```
> z$r[4]             # vierde coördinaat van z$r = het vierde residu
[1] -1.019029
```

Een list kan worden gevormd met behulp van het commando `list`. Hierbij kunnen de namen van de componenten onmiddelijk worden meegegeven met behulp van het = symbool.

```
> x <- 1:5
> y <- list(getallen=x, waarofniet=T)
> y
$getallen:
[1] 1 2 3 4 5


$waarofniet:
[1] TRUE
```

De functie `names` bevat de namen van de componenten van een list. Deze functie kan ook worden gebruikt om de namen te veranderen.

```
>                    # vervolg voorgaand display
> names(y)
1] "getallen"   "waarofniet"
>
> names(y) <- c("nummers", "waar")
```

Aan een list kunnen extra componenten worden toegevoegd door toekenning aan een geselecteerde component.

```
>                   # vervolg voorgaand display
> y[[3]] <- 1:3; y$vier <- "c"
> y
$nummers:
[1] 1 2 3 4 5

$waar:
[1] TRUE

[[3]]:
[1] 1 2 3

$vier:
[1] "c"
```

9. MATRICES EN ARRAYS

Een *matrix* kan op twee manieren uit een vector worden gecreëerd.

```
>      # de twee opdrachten ..
> x <- 1:8
> dim(x) <- c(2,4)
>      # ..  hebben hetzelfde resultaat als de ene opdracht
> x <- matrix(1:8,2,4, byrow=F)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

Bij het gebruik van zowel de functie `dim` als de functie `matrix` wordt een matrix bij default kolomsgewijs opgevuld met de coördinaten van de vector. Bij gebruik van `matrix` kan deze default worden doorbroken door het argument `byrow=T` mee te geven.

Een matrix kan ook worden gemaakt door een aantal vectoren van gelijke lengte koloms- of rijsgewijs te combineren met de functies `cbind` en `rbind`.

```
> cbind(c(1,2),c(3,4))
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Een matrix mag worden opgevat als een bijzonder soort vector: een vector met een extra attribuut `dim`. Alle numerieke bewerkingen die coördinaatsgewijs op vectoren kunnen worden uitgevoerd, kunnen ook op matrices worden toegepast.

```
> max(x)
[1] 8
> x * x^2
     [,1] [,2] [,3] [,4]
[1,]    1   27  125  343
[2,]    8   64  216  512
```

Daarnaast zijn er talloze functies die specifiek op matrices werken. Matrix vermenigvuldiging is `%*%` en de getransponeerde is `t(x)`.

```
> matrix(1:4,2,2, byrow=T) %*% x
     [,1] [,2] [,3] [,4]
[1,]    5   11   17   23
[2,]   11   25   39   53
>
> t(x)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
```

De functie `apply` past een als argument gegeven functie toe op alle rijen of kolommen van een matrix.

```
> apply(x,1,mean) # vormt de gemiddelden van de rijen van x
> apply(x,2,max)  # vormt de maxima van de kolommen van x
```

Matrix-Subscripts

Delen van een matrix `x` kunnen worden geselecteerd volgens het schema `x`[*subscript*]. Hierin heeft *subscript* één van drie vormen.

- Een paar (*rijen*, *kolommen*), waarin *rijen* de gewenste rijnummers en *kolommen* de gewenste kolomnummers zijn. 'Lege' collecties *rijen* en/of *kolommen* betekenen alle *rijen* of *kolommen*.

```
> x[i,j]  # het (i,j)-de element
> x[i,]   # de i-de rij
> x[3,c(1,5)] # de elementen in de 3-de  rij én 1-ste en 5-de kolom
> x[,-j]  # alles behalve de j-de kolom
```

- Een logische matrix van dezelfde vorm als `x`. Het resultaat is een vector.

```
> x[x>5]
[1] 6 7 8
> x>5
        [,1]    [,2]    [,3]   [,4]
[1,]   FALSE   FALSE   FALSE   TRUE
[2,]   FALSE   FALSE   TRUE    TRUE
```

functienaam	operatie
`ncol(x)`	aantal kolommen van een matrix
`nrow(x)`	aantal rijen van een matrix
`t(x)`	getransponeerde van een matrix
`diag(x)`	maak een diagonaalmatrix uit een vector of vorm de diagonaalvector uit een matrix
`col(x)`	matrix van kolomnummers
`row(x)`	matrix van rijnummers
`cbind(...)`	combineer kolommen tot een matrix
`rbind(...)`	combineer rijen tot een matrix
`apply(x,1,functie)`	pas `functie` toe op de rijen van de matrix
`apply(x,2,functie)`	pas `functie` toe op de kolommen van de matrix
`%*%`	matrix vermenigvuldiging
`solve(x)`	inverse van een matrix
`solve(A,b)`	oplossing van het stelsel $Ax = b$
`eigen(x)`	eigenwaarden
`chol(x)`	choleski decompositie
`qr(x)`	q-r decompositie
`svd(x)`	singuliere waarden decompositie
`cancor(x,y)`	canonische correlaties
`var(x)`	covariantiematrix van de kolommen

Tabel 6. Enkele matrix functies

Arrays

Matrices zijn tweedimensionale arrays. De elementaire operaties op algemene arrays zijn analoog aan die voor matrices.

De functie `dimnames` bevat de namen van de rijen en kolommen van een matrix.

```
> a                              # een karakter matrix
          sex height weight
   Jansen "M" "1.79" "75"
Pietersen "V" "1.61" "58"
 Klaassen "V" "0.80" "23"
> dimnames(a)                    # een list met twee karakter vectoren
[[1]]:                                          als componenten
[1] "Jansen"    "Pietersen" "Klaassen"

[[2]]:
[1] "sex"    "height" "weight"
> dim(a)
[1] 3 3
```

---

10. DATA-FRAMES

De elementen van een matrix als besproken in de vorige sectie moeten allemaal dezelfde `mode` bezitten: bijvoorbeeld numeriek, karakter of logisch. Een `data.frame` is in essentie een matrix waarvan de kolommen van verschillende `mode` mogen zijn. Deze datastructuur is vooral interessant in verband met het `read.table` commando om tabellen in te lezen. (Zie verderop.) Een `data.frame` kan ook worden gemaakt met behulp van het commando van dezelfde naam.

```
> x <- c("M","V","V")
> y <- c(1.79,1.61,0.80)
> z <- c(T,F,T)
> data.frame(x,y,z)
  x    y     z
1 M 1.79  TRUE
2 V 1.61 FALSE
3 V 0.80  TRUE
```

11. INPUT EN OUTPUT

Voor het invoeren van grote data-sets is het gebruik van de functie `c` niet aanbevolen. De beste manier is om de data eerst in te typen in een `UNIX`-file met behulp van een editor naar keuze en vervolgens binnen R gebruik te maken van de functie `scan` of de functie `read.table`. Het voordeel is, dat typfouten met behulp van de editor gemakkelijk kunnen worden verbeterd.

Inlezen van Vectoren

De snelste manier om numerieke data in te voeren is met de functie `scan`.

```
> x <- scan("filenaam")
```

Hierin is *filenaam* de betreffende `UNIX` tekstfile. De aanhalingstekens zijn verplicht. In de bovenstaande vorm leest de functie `scan` de getallen regel per regel, waarbij de getallen gescheiden moeten zijn door witte ruimte: een spatie, tab of nieuwe regel. Met een optie kan desgewenst een andere separator worden gebruikt. In deze basisvorm mag de 'tekst'file alleen getallen bevatten. Met een andere optie kunnen karakter strings in plaats van getallen worden gelezen. Het resultaat is steeds een vector, hoewel van een vector natuurlijk meteen een matrix kan worden gemaakt.

```
> x <- matrix(scan(" filenaam"),10,3) # een file met 30 getallen
```

Ook voor het invoeren van kleine datasets is `scan` soms handig. Als geen file als argument wordt gegeven, verwacht `scan` de input van het toetsenbord.

```
> x <- scan()
1:  1 2.5 3.14159     (ingetypt op het toetsenbord)
4:                                  (RETURN)
3 items read
> x
[1] 1.00000 2.50000 3.14159
```

De input wordt afgesloten met een `RETURN` of een `CONTROL`-D op een lege regel.

Inlezen van Tabellen

In veel gevallen bestaat een dataset uit een tabel waarin de getallen of tekst op iedere regel de gegevens aangaande één object betreft. Bijvoorbeeld, een `UNIX`-file kan de volgende structuur bezitten.

```
          sex height weight
Jansen    M   1.79   75
Pietersen V   1.61   58
Klaassen  V   0.80   23
```

De verschillende velden van de tabel zijn hier netjes onder elkaar geplaatst, maar van belang is alleen dat zij door witte ruimte zijn gescheiden. Het commando `read.table` zet zo'n tabel om in de data-structuur `data.frame`.

```
> z <- read.table(" filenaam")
> z
          sex height weight
   Jansen   M   1.79     75
Pietersen   V   1.61     58
 Klaassen   V   0.80     23
> dim(z)
[1] 3 3                        # z is een 3x3 matrix!
> dimnames(z)
[[1]]:
[1] "Jansen"    "Pietersen" "Klaassen"

[[2]]:
[1] "sex"    "height" "weight"
```

Bij een file met de gegeven structuur beschouwt `read.table` de eerste rij en kolom als de namen van respectievelijk de kolommen en rijen. Dit verklaart dat in het bovenstaande voorbeeld de gevormde `data.frame` dimensie 3×3 heeft, niet 4×4. Met behulp van een optioneel argument is dit te voorkomen.

In de meeste gevallen wordt R interactief gebruikt, maar het is soms handig een kleinere of grotere groep opdrachten tegelijk aan te bieden. Dit kan met behulp van `source`. Creëer eerst een `UNIX`-tekstfile met daarin de gewenste opdrachten op dezelfde manier getypt als waarop dat tijdens een interactieve R-sessie zou zijn gebeurd (zonder de R-prompts). Save deze file onder een willekeurige naam, bijvoorbeeld *filenaam*, en geef het R-commando.

```
> source("filenaam")
```

De opdrachten worden nu één voor één door R uitgevoerd en de R-prompt verschijnt weer in beeld nadat de uitvoering van de laatste opdracht is afgerond. Het onderstaande is een voorbeeld van een source file.

```
m <- numeric(1000)
for (i in 1:1000) {x <- rnorm(50)
                   m[i] <- mean(x)}
s <- var(m)
```

Data Editing

Een fout in een bestaande vector, matrix of data.frame `x` kan worden verbeterd door de

data-set met het `write` commando in een `UNIX`-file te schrijven (in ASCII-format), de data te veranderen met een editor (zoals `vi` of `emacs`) en vervolgens de file weer in te lezen volgens één van de bovenstaande methoden.

```
> x                                   # een 3x4 matrix
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> write(t(x),"file.naam",ncol=4)
```

12. ZELF FUNCTIES SCHRIJVEN

Het schrijven van R functies is eenvoudig. Door goed gebruik te maken van de al in R aanwezige functies en de data-structuren van R kan de definitie vaak kort zijn. De algemene definitie van een functie is:

```
> functienaam <- function(arg1, arg2, ...)   definitie
```

Hierin is `functienaam` een zelf gekozen naam, `function` een vast sleutelwoord dat aangeeft, dat het hier om een functiedefinitie gaat, `arg1, arg2, ...` zijn zelf gekozen namen van argumenten en *definitie* is een groep van R statements. De waarde van de laatste expressie in deze groep wordt de (uitvoer)waarde van de functie. Dit mag een vector zijn, een list of iedere andere R-datastructuur. Een groep R statements is ofwel een enkele opdracht, ofwel een rij opdrachten omsloten door accoladen. Bijvoorbeeld, de functie `sdev`

```
> sdev <- function(x) sqrt(var(x))
```

berekent de steekproef-standaarddeviatie. Het *definitie* gedeelte bestaat hier uit een enkele statement en roept twee andere functies aan. Een ingewikkelder voorbeeld is de steekproef kurtosis.

```
> kurtosis <- function(x){
+  vierdemoment <- sum((x - mean(x))^4)/(length(x)-1)
+  tweedemoment <- var(x)
+  vierdemoment/(tweedemoment^2)
}
```

Wijzigen van de definitie kan op verschillende manieren. Allereerst is het mogelijk een nieuwe definitie van voren af aan in te typen. Gemakkelijker is om gebruik te maken van een editor. Na

```
> kurtosis <- vi(kurtosis)
```

neemt de UNIX-editor `vi` het tijdelijk over van R en de functiedefinitie kan binnen `vi` worden veranderd. Save de uiteindelijke tekst binnen `vi` en verlaat `vi`. In de hervatte R sessie geldt vervolgens de nieuwe definitie.

Een andere manier om gebruik te maken van een editor is door de functie eerst op een UNIX-file te dumpen, de UNIX-file te bewerken met een editor en vervolgens de file binnen R te sourcen. Dit is vooral handig binnen een window systeem, omdat R en de editor dan tegelijkertijd in verschillende windows kunnen worden gebruikt. Het algemene schema is als volgt.

```
> dump("functienaam", "filenaam")
  ...
 (edit de file  filenaam met je favoriete editor, buiten R)


  ...
> source("filenaam")
```

Volgens hetzelfde idee is het ook mogelijk een nieuwe functie binnen R in te voeren. Creëer eerst met behulp van een editor buiten R een tekst file met daarin de definite van de functie. Gebruik vervolgens `source` om de functie binnen R te creëren. Het 'dumpen' blijft nu dus achterwege en voor het aanbrengen van veranderingen is `dump` ook onnodig zolang de oorspronkelijke tekst file blijft bestaan.

Bij de aanroep van een functie is ieder argument ofwel verplicht, ofwel optioneel. Dit verschil wordt zowel bij de uitvoering als bij de definitie van de functie bepaald. Ten eerste behoeft een argument dat bij een bepaalde uitvoering van de functie niet gebruikt wordt, ook niet te worden meegegeven. Bovendien is een een argument permanent optioneel als in de definitie van de functie een default waarde is meegegeven. Dit gebeurt met behulp van de constructie `arg =` *defaultwaarde* in de definitie.

```
> macht <- function(x, k=2) x^k   # de defaultwaarde voor k is 2
>
> macht(5)                        # de defaultwaarde wordt gebruikt
[1] 25
> macht(5,3)                      # de defaultwaarde wordt genegeerd
[1] 125
```

Toekenningen binnen een functie zijn locaal voor het '*frame*' van die functie. Dit betekent, dat de directe resultaten van zulke toekenningen na de volledige uitvoering van de functie verloren zijn. Alleen de waarde van de laatste regel van de functie wordt doorgegeven aan het hoofdprogramma. Hulpvariabelen binnen de functie kunnen derhalve zonder bezwaar dezelfde namen als variabelen in het hoofdprogramma bezitten. Andersom dient alle relevante informatie van de functie op de laatste regel te worden gezet.

```
> x <- 0
> verloren <- function(x) {x <- 3; x}
> verloren(x)
```

```
[1] 3
> x
[1] 0
```

In het zeldzame geval dat een globale en permanente toekenning binnen een functie de bedoeling is, kan gebruik worden gemaakt van de `<<-` operator.

```
> bewaard <- function(x) x<<-3
> bewaard(x)
[1] 3
> x
[1] 3
```

De argumenten van een functie mogen van ieder type R-object zijn; derhalve ook functies. Bijvoorbeeld, als `qverdeling` de kwantielfunctie van een generieke kansverdeling geeft, dan geeft de volgende functie een generator van toevalsgetallen uit die verdeling.

```
> rverdeling <- function(n,qverdeling){
+ u <- runif(n)
+ qverdeling(u)}
>
```

Voor de zinvolheid van deze definitie hoeft een functie met de naam `qverdeling` niet te bestaan, daar `qverdeling` alleen de formele naam is van het tweede argument van `rverdeling`. Bij de aanroep van `rverdeling` moet voor `qverdeling` echter een echte functie worden ingevuld.

```
> rverdeling(5, qnorm)
[1]  0.9332848  2.2774360 -0.3208976  0.1994874 -0.5723007
> rverdeling(1000, qeinstein)
Error:  Object "qeinstein" not found
Dumped
```

(Overigens is `rnorm` een standaard R-functie.) Als binnen een functie andere functies worden aangeroepen, dan kunnen de argumenten voor die secundaire functies worden meegegeven aan de hoofdfunctie. Een algemenere vorm van de toevalsgetal generator is als volgt.

```
> rverdeling <- function(n,qverdeling,mean=0,sdev=1){
+ u <- runif(n)
+ qverdeling(u, mean, sdev)}
```

Deze functie is echter alleen geschikt voor gebruik tesamen met kwantielfuncties `qverdeling` die precies twee argumenten behoeven. Een betere mogelijkheid is om de argumenten impliciet door te geven.

```
> rverdeling <- function(n,qverdeling,...){
+ u <- runif(n)
+ qverdeling(u,...)}
```

De betekenis van de (precies) drie puntjes is, dat ieder argument, dat bij de feitelijke aanroep van `rverdeling` op de plaats van ... verschijnt, letterlijk wordt doorgegeven aan `qverdeling`. Gegeven de laatste definitie van `rverdeling` zijn de volgende twee aanroepen nu beide geoorloofd.

```
> rverdeling(5, qnorm,10,1)   # qnorm verwacht 2 argumenten
> rverdeling(5,qpois,2)       # qpois verwacht 1 argument
```

13. IF, FOR, WHILE, REPEAT

De R-taal kent een voorwaardelijke constructie en mogelijkheden voor loops analoog aan andere programmeertalen. De `if` functie in R kent twee vormen.

> `if (`*conditie*`)` *expr1*

> `if (`*conditie*`)` *expr1* `else` *expr2*

In beide gevallen wordt eerst de *conditie* geëvalueerd; als de waarde `TRUE` is wordt *expr1* uitgerekend. Is de waarde van de *conditie* `FALSE`, dan gebeurt er niets in de eerste `if` statement; in de tweede statement wordt *expr2* geëvalueerd. Vergeet de ronde haakjes rond *conditie* niet!

Als de *conditie* tot een logische vector van lengte groter dan één evalueert, dan geeft R een waarschuwing, maar voert toch de `if` statement uit met de eerste coördinaat van de logische vector.

Bij het schrijven van een functie kan de constructie `if (!missing(`*argument*`))` worden gebruikt om te testen of bij de aanroep van de functie een bepaald argument is meegegeven of niet.

Uit de voorgaande secties is duidelijk, dat R veel loop-iteraties impliciet uitvoert. Bijvoorbeeld, het commando `sqrt(x)` past automatisch de wortelfunctie toe op alle coördi-naten van de vector `x`. Door hiervan handig gebruik te maken is het vaak mogelijk expliciete loop constructies te vermijden. Naast tot besparing van typwerk leidt dit in veel gevallen ook tot aanzienlijk sneller rekenwerk. De nu te bespreken expliciete loop constructies zijn in R relatief langzaam.

De belangrijkste loop constructie is de `for` constructie.

> `for (name in` *values*`)` *expr*

Het resultaat is het één voor één toekennen van de elementen van *values* aan `name` en het steeds evalueren van *expr*. Hierbij is *values* meestal een vector of een list. De grootheid `name` is een dummy, die aan het eind van de `for` statement niet meer bestaat.

```
> x <- y <- NULL        # NULL is een leeg object
```

```
> for (i in 1:10) {x <- c(x,i); y <-c(y,0)}
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> y
 [1] 0 0 0 0 0 0 0 0 0 0
```

Een alternatief is de `while` statement.

```
> while (conditie) expr
```

De uitvoering begint met het testen van de *conditie*. Als de waarde `FALSE` is, stopt de uitvoering; als de waarde `TRUE` is, dan wordt *expr* geëvalueerd en begint de uitvoering van voren af aan.

```
> x <- numeric(10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
> i <- 0
> while (i < 10) { i <- i+1
+                  x[i]<- i}
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

Het zij opgemerkt, dat de voorgaande voorbeelden geen goed gebruik van R zijn. Vergelijk het commando.

```
> x <- 1:10
```

De `repeat` constructie kan worden gebruikt in samenhang met `break`. De algemene vorm is

```
> repeat {
    ...
    if ( conditie) break
    ...
}
```

Wanneer R `break` evalueert, verlaat het de binnenste `for`, `while`, of `repeat` loop waarin `break` bevat is. Naast `break` is het ook mogelijk `next` te gebruiken. Dit zorgt ervoor, dat de volgende iteratie van de loop onmiddelijk wordt begonnen, zonder de huidige af te ronden.

---

14. DATA DIRECTORIES

Bij het afsluiten van R wordt gevraagd of de workspace bewaard moet blijven. Indien je y (van yes) intypt worden alle objecten die je tijdens je R-sessie gecreëerd hebt, bewaard in de file '.RData' in de directory waar je R had opgestart. Ze zijn dus tijdens je volgende R-sessie weer aanwezig als je R in dezelfde directory opstart. Technisch gesproken correspondeert met ieder R-object een gewone `UNIX`-file met dezelfde naam als het R-object. Dit betekent, dat de file kan worden weggegooid, gecopiëerd, van naam veranderd, etcetera, met de desbetreffende `UNIX`-commandos. De files in de `.Data`-directory zijn echter geen tekstfiles en kunnen niet buiten R worden gelezen: om de werking van R te versnellen worden de R-objecten in een speciaal format gecodeerd. Je kunt leesbare afdrukken van R-objecten maken met behulp van de R-commando's `write` en `cat` (voor data-objecten) en `dump` (vooral geschikt voor functies).

Binnen R krijg je een overzicht van alle objecten in je `.Data`-directory met het `ls` commando. Onnodige objecten kunnen worden verwijderd met `rm`.

```
> ls()
> ls("x*")       # print alle objecten waarvan de naam begint met x
> rm(x,y,object)     # verwijdert x, y en object
```

Beide operaties kunnen ook met de gelijkluidende `UNIX`-commando's buiten R worden uitgevoerd. Wild cards zijn toegestaan.

Naast de door jezelf gecreëerde R-objecten bevindt zich een groot aantal standaard R-objecten, voornamelijk functies, in directories ergens in het systeem. R doorzoekt relevante directories voortdurend teneinde de gevraagde objecten te vinden. Dit zoeken vindt plaats in een vaste volgorde, de *search list*. Bij default bestaat deze uit de .Data directory van je home directory

R stopt met zoeken zodra het genoemde object gevonden is. Een gevolg is, dat een standaardfunctie met een bepaalde naam (in de R-functies directory) onbruikbaar is vanaf het moment, dat een functie van dezelfde naam in de `.Data`-directory bestaat. De remedie bij dubbel gebruik van een naam, is om het zelf gecreëerde object te verwijderen. Je behoeft niet bang te zijn een standaardfunctie te verwijderen: R zal een dergelijke opdracht altijd weigeren.

15. CATEGORISCHE DATA

Een *category* is een data-structuur in R, die kan worden gebruikt bij het analyseren van categorische data: data waarvan het waardenbereik een verzameling codes is. De functie `cut(x, breaks)` creëert een category uit een numerieke vector `x` door voor ieder van de coördinaten van `x` na te gaan in welk van de door `breaks` gedefinieerde intervallen het ligt.

```
> x
[1] 3 3 5 1 5 5
> breaks
[1] 0 2 4 6 8
```

```
>               # de intervallen zijn (0,2], (2,4], etc.
>               # x[1] is in het 2e interval, x[3] in het derde, etc
>  cut(x,breaks)
[1] (2,4] (2,4] (4,6] (0,2] (4,6] (4,6]
Levels:  (0,2] (2,4] (4,6] (6,8]
```

De functie `cut` kan worden gecombineerd met `table` om het aantal coördinaten van `x` dat in ieder interval valt te tellen.

```
> table(cut(x,breaks))
(0,2] (2,4] (4,6] (6,8]
    1     2     3     0
```

Een category kan worden opgevat als een vector met een extra attribuut: *levels*. Dit betekent, dat met een category kan worden gerekend als met vectoren.

```
> table(cut(x,breaks))+10
(0,2] (2,4] (4,6] (6,8]
   11    12    13    10
```

De functie `table` kan ook worden gebruikt voor het creëren van hoger dimensionale tabellen. Gegeven twee vectoren of categories `x` en `y` van gelijke lengte, produceert `table(x,y)` een $k \times r$ matrix met als (i, j) element het aantal paren `(x[l],y[l])` dat level i in `x` en level j in `y` heeft. Hierbij zijn de levels van een vector gelijk aan de geordende verschillende waarden.

```
> x
[1] 3 3 5 1 5 5
> y
[1] 0 1 0 0 1 1
>
> table(x,y)
  0 1
1 1 0
3 1 1
5 1 2
>
>  table(cut(x,breaks),y)
     y
      0 1
(0,2] 1 0
(2,4] 1 1
(4,6] 1 2
```

```
(6,8] 0 0
```

De functie `tapply(x,category,functie)` kan worden gebruikt om een gegeven functie `functie` toe te passen op alle coördinaten van een vector `x` van hetzelfde level, voor ieder level. De levels worden gegeven in het tweede argument `category`: `x[i]` wordt gedacht level `category[i]` te bezitten. De werking is het gemakkelijkst in twee stappen voor te stellen. Eerst worden alle coördinaten van `x` per level gesorteerd. Vervolgens wordt op ieder groepje coördinaten van hetzelfde level (opgevat als een subvector van `x`) de functie toegepast. Het resultaat is een vector ter lengte van het totaal aantal levels.

```
> x
[1] 3 3 5 1 5 5
> breaks
[1] 0 2 4 6 8
>
> tapply(x,cut(x,breaks),sum)
(0,2] (2,4] (4,6] (6,8]
    1     6    15    NA
>
> y
[1] 0 1 0 0 1 1
> tapply(x,y,sum)
 0  1
 9 13
```

INDEX